Rogue Wave Standard C++ Library

Documentation

Copyright 1999 Rogue Wave Software, Inc.

Table of Contents

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

If you are accessing this for the first time, please read the licensing statement.

Standard C++ Library Class Reference

Welcome to the Standard C++ Library Class Reference. Click on the "Chapter 2: Reference" link below to see a hypertext list of class descriptions.

For member functions and data types (exclusive of constructors and destructors), you can consult the comprehensive index. Each class description has an index specific to the class.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 1: Introduction

Sections in This Chapter

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Class Reference Overview

This reference guide is an alphabetical listing of the C++ classes in the **Standard C++ Library**. The entry for each class includes:

- Either a general category to which the class belongs or an illustration showing the inheritance hierarchy
- A brief summary of the class functionality
- A synopsis indicating the header file(s) associated with the class
- A description of the class
- The C++ code that describes the class interface
- All methods associated with the class, including constructors, operators, member functions, etc.

  The constructors, operators, member functions, etc. are grouped together and listed alphabetically within each group. These groups are not a part of the C++ language, but they are a useful way to organize information.
- Examples and warnings

Entries that are not classes include different information. In general, there is still a brief summary of the entry's function, followed by a more detailed description, and in many cases an example.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

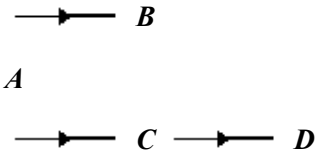
Conventions

Function names, examples, operating system commands, mathematical symbols, and code fragments are shown in a `Courier` font. Ellipses are used in code examples to indicate that some part of the code is missing.

Throughout the documentation, there are frequent references to "self," which should be understood to mean `*this`.

The following convention is used to show that class ***A*** inherits from class ***B***:

When a class inherits from more than one class, or there are multiple levels of inheritance, all of its inheritance relationships are shown. For example, the following illustration indicates that class ***A*** inherits from class ***B*** and from class ***C***, which inherits from class ***D***.



The notation system used in the inheritance hierarchies is based on the Object Modeling Technique (OMT) developed by Rumbaugh and others.1

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Standards Conformance

The information presented in this reference conforms with the requirements of the ANSI X3J16/ISO WG21 Joint C++ Committee.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top

SINGLE USER SOFTWARE LICENSE

PLEASE READ THIS AGREEMENT BEFORE OPENING THIS SOFTWARE PACKAGE. IF YOU OPEN THIS PACKAGE OR KEEP IT FOR MORE THAN THIRTY (30) DAYS, YOU ACCEPT ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, DO NOT OPEN THIS SOFTWARE PACKAGE, AND PROMPTLY RETURN THIS SOFTWARE PACKAGE ALONG WITH DATED PROOF OF PAYMENT FOR A REFUND. YOU MAY ONLY UNLOCK AND/OR USE THE SOFTWARE FOR WHICH YOU HAVE A PAID-UP LICENSE AND FOR WHICH YOU HAVE LEGALLY RECEIVED AN UNLOCK KEY.

1. DEFINITION OF TERMS

(a) "Documentation": any explanatory written or on-line material including, but not limited to, user guides, reference manuals and HTML files.

(b) "Software": all material in this distribution including, but not limited to, source code, object code, byte code, dynamic-link libraries, shared libraries, static libraries, header files, executables, scripts and Documentation.

(c) "Licensed Software": the Software for which you have paid the applicable license fee and received an authorized unlock key.

(d) "Software Application Programming Interface ("API")": the set of access methods, whether provided by Rogue Wave Software, third parties, or developed by the licensee, through which the programmatic services provided by the Licensed Software are made available. The aforementioned access methods may use technology including, but not limited to, header files, events, network communications, macro and scripting languages, component technologies such as COM or JavaBeans, or distributed technologies such as CORBA, DCOM, or Java RMI.

(e) "Value-Added Interface": a programmatic interface that has added significant value to the Software Application Programming Interface and makes use of the Licensed Software in its implementation, but without exposing any part of the Software Application Programming Interface through any means either directly or indirectly.

(f) "User Software Product": an application developed by the licensee intended for execution on a computer, that makes use of the Licensed Software in its implementation, but without exposing any part of the API through any means either directly or indirectly.

(g) "Mainframe Class Computers": such computers as are marketed or commonly considered to be mainframe computers in the computer industry.

2. GENERAL

The Software is owned by Rogue Wave Software, Inc. ("Rogue Wave") and is protected by U.S. copyright laws and other laws and by international treaties. It is intended for use by a software programmer who has experience using development tools and class libraries. The Software is not intended for use by consumers for household use.

This Rogue Wave Single User Software License Agreement ("Agreement") accompanies the Licensed Software. This copy of the Licensed Software is licensed to you as the end user or to your employer ("Company") for your exclusive use as an end user. "You" as used in the remainder of this Agreement shall refer to the individual licensee, whether as an individual programmer or as a Company. If you do not accept this Agreement, you may return this package along with dated proof of purchase to the place you obtained it within thirty (30) days and your money will be refunded, and you will not be licensed to unlock or to use the Software.

3. LICENSE GRANTS

Subject to the terms and conditions of this Agreement, Rogue Wave grants to you the non-exclusive, non-transferable, non-concurrent right to use the Software Application Programming Interface on a single computer by a single individual user, except that this license does not grant you the right to use the Licensed Software on any Mainframe Class Computer. If multiple copies have been purchased, you may load one copy of the Licensed Software on a server and permit those individuals (who purchased the Licensed Software or for whom the Licensed

©Copyright 1999 Rogue Wave Software, Inc.

Software was purchased) to access and execute the Licensed Software. This is not a concurrent user license; the rights granted are not for any individual(s) up to the number of licenses purchased. The rights are granted only to the specified individual(s) in your purchase order for whom the Licensed Software was purchased. You may also:

(a) Make one backup copy of the Licensed Software solely for archival and disaster-recovery purposes, or

Transfer the Licensed Software to a hard disk and keep the original copy solely for archival and disaster-recovery purposes.

You may use the Licensed Software to develop Value-Added Interfaces and User Software Products only for the single platform (operating system environment) or platforms for which you have purchased the license. If you wish to develop User Software Products for use on other platforms for which you do not have a license, you must purchase additional licenses for each additional platform. Your purchase of a license does not entitle you to use the Software licensed for a platform different than the platform(s) for which your copy is licensed.

(b) You may distribute User Software Products for internal use only, including those portions of the Licensed Software used solely for purposes of supporting execution of said User Software Product where reasonable steps have been taken to ensure that no parts of the Software Application Programming Interface or any internally used Value-Added Interfaces (if any) have been exposed directly or indirectly, and where the license for said User Software Products explicitly prohibits the use of the User Software Product for software development use.

YOU HAVE NO RIGHTS TO USE THE LICENSED SOFTWARE BEYOND THOSE SPECIFICALLY GRANTED IN THIS SECTION.

4. LICENSE RESTRICTIONS

Notwithstanding any provisions in this Agreement to the contrary, you may not distribute:

(a)
- any portion of the Software Application Programming Interface,
- any Value-Added Interface,
- any executable delivered with the Licensed Software, or
- any portion of the Documentation.

In addition, you may not:

(b)
- decompile, disassemble, or reverse engineer any object code form of any portion of the Licensed Software,
- export from the United States any portion of the Licensed Software without obtaining the prior written consent of Rogue Wave and all applicable export licenses and governmental permits,
- disclose any source code of the Licensed Software to any person or entity, or
- copy the Documentation, including any Documentation available in on-line form (except that, with respect to Documentation available only in on-line form, you may make a single tangible copy solely for use with the Licensed Software).

5. TITLE

You acknowledge and agree that all right, title and interest in and to the Licensed Software, including all intellectual property rights therein, are the property of Rogue Wave, subject only to the licenses granted to you under this Agreement. This Agreement is not a sale and does not transfer to you any title or ownership in or to the Licensed Software or any patent, copyright, trade secret, trade name, trademark or other proprietary or intellectual property rights related thereto.

6. NON-TRANSFERABILITY

You may not rent, transfer, assign, sublicense or grant any rights in the Licensed Software, in full or in part, to any other person or entity without Rogue Wave's written consent.

7. LIMITED WARRANTIES

©Copyright 1999 Rogue Wave Software, Inc.

Rogue Wave warrants to you that the Licensed Software will substantially perform the functions described in the Documentation for a period of thirty (30) days after the date of delivery of the Licensed Software to you. Rogue Wave's sole and exclusive obligation, and your sole and exclusive remedy, under this warranty is limited to Rogue Wave's using reasonable efforts to correct material, documented, reproducible defects in the Licensed Software that you describe and document to Rogue Wave during the thirty (30) day warranty period. In the event that Rogue Wave fails to correct a material, documented, reproducible defect within a reasonable period, Rogue Wave may, at Rogue Wave's discretion, replace the defective Licensed Software or refund to you the amount that you paid Rogue Wave for the defective Licensed Software and cancel this Agreement and the licenses granted herein. In such event, you agree to return to Rogue Wave all copies of the Licensed Software (including the original).

Rogue Wave also warrants that the physical media on which the Licensed Software is delivered to you (unless the Licensed Software is downloaded electronically by you and no media is supplied by Rogue Wave) will be free from defects in manufacturing and workmanship for a period of thirty (30) days from the date of delivery. Rogue Wave will, at Rogue Wave's expense, replace any defective media returned to Rogue Wave within thirty (30) days of the date of delivery of the media to you.

EXCEPT AS EXPRESSLY SET FORTH ABOVE, ROGUE WAVE EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. LIMITATION OF LIABILITY

IN NO EVENT SHALL ROGUE WAVE BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS, REVENUES, DATA OR OTHER ECONOMIC ADVANTAGE) WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY, EVEN IF ROGUE WAVE WAS ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

9. AUDIT

Rogue Wave may, no more than once during any twelve (12) month period, unless there is evidence of violation, request you to conduct an internal audit of your use of the Licensed Software, or Rogue Wave may request an audit by an independent party, to ensure compliance with the provisions of this Agreement. Rogue Wave shall provide you with no less than fifteen (15) days prior notice of each such audit. If Rogue Wave requests an independent auditor, the cost of such audit will be paid by Rogue Wave, unless said audit shows you to be in violation of this Agreement.

10. TERMINATION

Rogue Wave reserves the right, at its sole discretion, to terminate this Agreement upon written notice if you have breached the terms and conditions hereof. You may terminate this Agreement at any time by ceasing to use the Licensed Software and by returning all copies of the Licensed Software (including the original) to Rogue Wave or by destroying all copies of the Licensed Software (including the original). Sections 4, 5, 6, 7, 8 and 10 survive any termination of this Agreement and apply fully to any termination. Unless terminated by either party, this Agreement shall remain in effect.

11. MISCELLANEOUS

(a) Applicable Law and Jurisdiction. This Agreement will be governed by and construed in accordance with the laws of the State of California without regard to conflict of laws principles and without regard to the 1980 U.N. Convention on Contracts for the International Sale of Goods. The federal and state courts of California shall have exclusive jurisdiction and venue to adjudicate any dispute arising out of this Agreement, and you expressly consent to (i) the personal jurisdiction of the state and federal courts of California, and (ii) service of process being effected upon you by registered mail.

(b) Limitation of Actions. No action or proceeding against Rogue Wave, whether for breach, indemnification, contribution, or otherwise, shall be commenced more than one year after delivery of the Licensed Software, and no such claim may be brought unless Rogue Wave has first been given commercially reasonable notice, a full written explanation of all pertinent details (including copies of all materials), and a good faith opportunity to resolve the matter.

(c) Invalidity and Waiver. Should any provision of this Agreement be held by a court of law to be illegal, invalid,

©Copyright 1999 Rogue Wave Software, Inc.

or unenforceable, the legality, validity, and enforceability of the remaining provisions of this Agreement will not be affected or impaired thereby. The failure of any party to enforce any of the terms or conditions of this Agreement, unless waived in writing, will not constitute a waiver of that party's right to enforce each and every term and condition of this Agreement.

(d) U.S. Government Restricted Rights. The Licensed Software is provided with Restricted Rights. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Rogue Wave Software, Inc., 5500 Flatiron Parkway, Boulder, Colorado 80301 USA.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND ROGUE WAVE WHICH SUPERSEDES ANY PROPOSAL OR PRIOR OR CONTEMPORANEOUS AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

Index

Click on one of the letters below to jump immediately to that section of the index. If you get no response, that letter has no entries.

Symbols A B C D E F G H I J K L M N O P

Q R S T U V W X Y Z

This index does not include constructors or destructors. Refer to the applicable class description for these.

Symbols

a

b

©Copyright 1999 Rogue Wave Software, Inc.

c

©Copyright 1999 Rogue Wave Software, Inc.

d

©Copyright 1999 Rogue Wave Software, Inc.

e

©Copyright 1999 Rogue Wave Software, Inc.

f

©Copyright 1999 Rogue Wave Software, Inc.

g

h

i

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

k

l

©Copyright 1999 Rogue Wave Software, Inc.

m

n

o

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

in gslice_array
in indirect_array
in mask_array
in slice_array
in valarray
operator>()
in basic_string
in deque
in list
in map
in multimap
in multiset
in pair
in queue
in set
in stack
in valarray
in vector
operator>=()
in basic_string
in deque
in list
in map
in multimap
in multiset
in pair
in set
in stack
in valarray
in vector
operator<<()
in basic_ostream
in basic_string
in bitset
in complex
in smanip
in valarray
operator<<=()
in bitset
in gslice_array
in indirect_array
in mask_array
in slice_array
in valarray
operator<()
in basic_string
in deque
in list
in map
in multimap
in multiset
in pair
in queue
in set
in stack
in valarray
in vector
operator<=()
in basic_string
in deque
in list
in map
in multimap
in multiset

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

operator/=()
- in complex
- in gslice_array
- in indirect_array
- in mask_array
- in slice_array
- in valarray

operator=()
- in auto_ptr
- in back_insert_iterator
- in basic_string
- in bitset
- in complex
- in deque
- in exception
- in front_insert_iterator
- in gslice_array
- in indirect_array
- in insert_iterator
- in list
- in locale
- in map
- in mask_array
- in multimap
- in multiset
- in ostreambuf_iterator
- in ostream_iterator
- in raw_storage_iterator
- in set
- in slice_array
- in valarray
- in vector

operator==()
- in basic_string
- in bitset
- in complex
- in deque
- in istreambuf_iterator
- in istream_iterator
- in list
- in locale
- in map
- in multimap
- in multiset
- in pair
- in queue
- in set
- in stack
- in valarray
- in vector

operator[]()
- in basic_string
- in deque
- in map
- in valarray
- in vector

operator^()
- in bitset
- in bitset
- in valarray
- in valarray
- in valarray
- in valarray
- in valarray

©Copyright 1999 Rogue Wave Software, Inc.

p

©Copyright 1999 Rogue Wave Software, Inc.

q

r

©Copyright 1999 Rogue Wave Software, Inc.

s

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

t

©Copyright 1999 Rogue Wave Software, Inc.

u

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Chapter 2: Reference

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

accumulate

`Generalized Numeric Operation`

Summary

Accumulates all elements within a range into a single value.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <numeric>
template <class InputIterator, class T>
T accumulate (InputIterator first,
              InputIterator last,
              T init);

template <class InputIterator,
          class T,
          class BinaryOperation>
T accumulate (InputIterator first,
              InputIterator last,
              T init,
              BinaryOperation binary_op);
```

Description

accumulate applies a binary operation to `init` and each value in the range `[first,last)`. The result of each operation is returned in `init`. This process aggregates the result of performing the operation on every element of the sequence into a single value.

Accumulation is done by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order. If the sequence is empty, ***accumulate*** returns `init`.

`binary_op` should not have side effects.

Complexity

accumulate performs exactly `last-first` applications of the binary operation (`operator+` by default).

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// accum.cpp
//
 #include <numeric>   //for accumulate
 #include <vector>    //for vector
 #include <functional> //for times
 #include <iostream>
 using namespace std;

 int main()
 {
   //
   //Typedef for vector iterators
   //
   typedef vector<int>::iterator iterator;
   //
   //Initialize a vector using an array of ints
   //
   int d1[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v1(d1, d1+10);
   //
   //Accumulate sums and products
   //
   int sum = accumulate(v1.begin(), v1.end(), 0);
   int prod = accumulate(v1.begin(), v1.end(),
               1, times<int>());
   //
   //Output the results
   //
   cout << "For the series: ";
   for(iterator i = v1.begin(); i != v1.end(); i++)
       cout << *i << " ";

   cout << " where N = 10." << endl;
   cout << "The sum = (N*N + N)/2 = " << sum << endl;
   cout << "The product = N! = " << prod << endl;
   return 0;
 }
```

Program Output

```
For the series: 1 2 3 4 5 6 7 8 9 10  where N = 10.
The sum = (N*N + N)/2 = 55
The product = N! = 3628800
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

adjacent_difference

`Generalized Numeric Operation`

Summary

Outputs a sequence of the differences between each adjacent pair of elements in a range.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <numeric>
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference (InputIterator first,
                                    InputIterator last,
                                    OutputIterator result);
template <class InputIterator,
          class OutputIterator,
          class BinaryOperation>
OutputIterator adjacent_difference (InputIterator first,
                                    InputIterator last,
                                    OutputIterator result,
                                    BinaryOperation bin_op);
```

Description

Informally, ***adjacent_difference*** fills a sequence with the differences between successive elements in a container. The result is a sequence in which the first element is equal to the first element of the sequence being processed, and the remaining elements are equal to the calculated differences between adjacent elements. For instance, applying ***adjacent_difference*** to {1,2,3,5} produces a result of {1,1,1,2}.

By default, subtraction is used to compute the difference, but you can supply any binary operator. The binary operator is then applied to adjacent elements. For example, by supplying the plus (+) operator, the result of applying ***adjacent_difference*** to {1,2,3,5} is the sequence {1,3,5,8}.

Formally, ***adjacent_difference*** assigns to every element referred to by iterator `i` in the range `[result + 1, result + (last - first))` a value equal to the appropriate one of the following:

```
*(first + (i - result)) - *(first + (i - result) - 1)
```

or

```
binary_op (*(first + (i - result)),
           *(first + (i - result) - 1))
```

©Copyright 1999 Rogue Wave Software, Inc.

`result` is assigned the value of `*first`.

`binary_op` should not have side effects.

`adjacent_difference` returns `result + (last - first)`.

`result` can be equal to `first`. This allows you to place the results of applying ***adjacent_difference*** into the original sequence.

Complexity

This algorithm performs exactly `(last-first) - 1` applications of the default operation (`-`) or `binary_op`.

Example

```
//
// adj_diff.cpp
//
 #include<numeric>       //For adjacent_difference
 #include<vector>        //For vector
 #include<functional>    //For times
 #include <iostream>
 using namespace std;
 int main()
 {
   //
   //Initialize a vector of ints from an array
   //
   int arr[10] = {1,1,2,3,5,8,13,21,34,55};
   vector<int> v(arr,arr+10);
   //
   //Two uninitialized vectors for storing results
   //
   vector<int> diffs(10), prods(10);
   //
   //Calculate difference(s) using default operator (minus)
   //
   adjacent_difference(v.begin(),v.end(),diffs.begin());
   //
   //Calculate difference(s) using the times operator
   //
   adjacent_difference(v.begin(), v.end(), prods.begin(),
         times<int>());
   //
   //Output the results
   //
   cout << "For the vector: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   cout << "The differences between adjacent elements are: "
        << endl << "     ";
   copy(diffs.begin(),diffs.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   cout << "The products of adjacent elements are: "
        << endl << "     ";
   copy(prods.begin(),prods.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   return 0;
```

Program Output

```
For the vector:
     1 1 2 3 5 8 13 21 34 55
The differences between adjacent elements are:
    1 0 1 1 2 3 5 8 13 21
The products of adjacent elements are:
     1 1 2 6 15 40 104 273 714 1870
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

adjacent_find

`Algorithm`

Summary

Find the first adjacent pair of elements in a sequence that are equivalent.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator>
  ForwardIterator
  adjacent_find(ForwardIterator first,
                ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
 ForwardIterator
  adjacent_find(ForwardIterator first, ForwardIterator last,
                BinaryPredicate pred);
```

Description

There are two versions of the ***adjacent_find*** algorithm. The first finds equal adjacent elements in the sequence defined by iterators `first` and `last` and returns an iterator `i` pointing to the first of the equal elements. The second version lets you specify your own binary function to test for a condition. It returns an iterator `i` pointing to the first of the pair of elements that meet the conditions of the binary function. In other words, ***adjacent_find*** returns the first iterator `i` such that both `i` and `i + 1` are in the range `[first, last)` for which one of the following conditions holds:

```
*i == *(i + 1)
```

or

```
pred(*i,*(i + 1)) == true
```

If ***adjacent_find*** does not find a match, it returns `last`.

Complexity

adjacent_find performs exactly `find(first,last,value) - first` applications of the corresponding predicate.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// find.cpp
//
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[10] = {0,1,2,2,3,4,2,2,6,7};

   // Set up a vector
   vector<int> v1(d1,d1 + 10);

   // Try find
   iterator it1 = find(v1.begin(),v1.end(),3);

   // Try find_if
   iterator it2 =
    find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));

   // Try both adjacent_find variants
   iterator it3 = adjacent_find(v1.begin(),v1.end());

   iterator it4 =
      adjacent_find(v1.begin(),v1.end(),equal_to<int>());

   // Output results
   cout << *it1 << " " << *it2 << " " << *it3 << " "
        << *it4 << endl;

   return 0;
 }
```

Program Output :

```
3 3 2 2
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

find

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

advance

`Iterator Operation`

Summary

Moves an iterator forward or backward (if available) by a certain distance.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class InputIterator, class Distance>
void advance (InputIterator& i, Distance n);
```

Description

The ***advance*** template function allows an iterator to be advanced through a container by some arbitrary distance. For bidirectional and random access iterators, this distance may be negative. For random access iterators, this function uses `operator+` and `operator-` for constant time implementations. For input, forward, and bidirectional iterators, ***advance*** uses `operator++` for linear time implementations. ***advance*** also uses `operator--` with bidirectional iterators for linear time implementations of negative distances.

If `n` is positive, ***advance*** increments iterator reference `i` by `n`. For negative `n`, ***advance*** decrements reference `i`. Remember that ***advance*** accepts a negative argument `n` for random access and bidirectional iterators only.

Example

```
//
// advance.cpp
//
 #include<iterator>
 #include<list>
 #include<iostream>
 using namespace std;

 int main()
 {

   //
   //Initialize a list using an array
   //
   int arr[6] = {3,4,5,6,7,8};
   list<int> l(arr,arr+6);
   //
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   //Declare a list iterator, s.b. a ForwardIterator
   //
   list<int>::iterator itr = l.begin();
   //
   //Output the original list
   //
   cout << "For the list: ";
   copy(l.begin(),l.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   cout << "When the iterator is initialized to l.begin(),"
        << endl << "it points to " << *itr << endl << endl;
   //
   // operator+ is not available for a ForwardIterator,
   // so use advance.
   //

   advance(itr, 4);
   cout << "After advance(itr,4), the iterator points to "
        << *itr << endl;
   return 0;
 }
```

Program Output :

```
For the list: 3 4 5 6 7 8
When the iterator is initialized to l.begin(),
it points to 3
After advance(itr,4), the iterator points to 7
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Sequences, ***Random Access Iterators***, ***distance***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Algorithms

Summary

Generic algorithms for performing various operations on containers and sequences.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
```

The synopsis of each algorithm appears in its entry in the reference guide.

Description

The Standard C++ Library allows you to apply generic algorithms to containers, and it supplies a set of these algorithms for searching, sorting, merging, transforming, scanning, and more.

Each algorithm can be applied to a variety of containers, including those defined by a user of the library. The following design features make algorithms generic:

- Generic algorithms access the collection through iterators
- Algorithms are templatized on iterator types
- Each algorithm is designed to require the least number of services from the iterators it uses

In addition to requiring certain iterator capabilities, algorithms may require a container to be in a specific state. For example, some algorithms can only work on previously sorted containers.

Because most algorithms rely on iterators to gain access to data, they can be grouped according to the type of iterator they require, as is done in the *Algorithms by Iterator* section below. They can also be grouped according to the type of operation they perform.

Algorithms by Mutating/Non-mutating Function

The broadest categorization groups algorithms into two main types: mutating and non-mutating. Algorithms that alter (or mutate) the contents of a container fall into the mutating group. All others are considered non-mutating. For example, both ***fill*** and ***sort*** are mutating algorithms, while ***find*** and ***for_each*** are non-mutating.

Non-mutating operations

©Copyright 1999 Rogue Wave Software, Inc.

```
accumulate              find_end                max_element
adjacent_find           find_first_of           min
binary_search           find_if                 min_element
count_min               for_each                mismatch
count_if                includes                nth_element
equal                   lexicographical_compare search
equal_range             lower_bound             search_n
find                    max
```

Mutating operations

```
copy                    remove_if
copy_backward           replace
fill                    replace_copy
fill_n                  replace_copy_if
generate                replace_if
generate_n              reverse
inplace_merge           reverse_copy
iter_swap               rotate
make_heap               rotate_copy
merge                   set_difference
nth_element             set_symmetric_difference
next_permutation        set_intersection
partial_sort            set_union
partial_sort_copy       sort
partition               sort_heap
prev_permutation        stable_partition
push_heap               stable_sort
pop_heap                swap
random_shuffle          swap_ranges
remove                  transform
remove_copy             unique
remove_copy_if          unique_copy
```

Note that the library has place and copy versions of many algorithms, such as ***replace*** and ***replace_copy***. The library also has versions of algorithms that allow the use of default comparators and comparators supplied by the user. Often these functions are overloaded, but in some cases (where overloading proved impractical or impossible) the names differ (for example, ***replace***, which uses equality to determine replacement, and ***replace_if***, which accesses a user-provided compare function).

Algorithms by Operation

We can further distinguish algorithms by the kind of operations they perform. The following lists all algorithms by loosely grouping them into similar operations.

Initializing operations

```
fill                    generate
fill_n                  generate_n
```

Search operations

```
adjacent_find           find_end            search_n
count                   find_if
count_if                find_first_of
find                    search
```

Binary search operations (Elements must be sorted)

```
binary_search           lower_bound
equal_range             upper_bound
```

Compare operations

```
equal                   mismatch
lexicographical_compare
```

Copy operations

```
copy                    copy_backward
```

Transforming operations

©Copyright 1999 Rogue Wave Software, Inc.

```
partition                       reverse
random_shuffle                  reverse_copy
replace                         rotate
replace_copy                    rotate_copy
replace_copy_if                 stable_partition
replace_if                      transform

Swap operations

swap                            swap_ranges
```

Scanning operations

```
accumulate                      for_each
```

Remove operations

```
remove                          remove_if
remove_copy                     unique
remove_copy_if                  unique_copy
```

Sorting operations

```
nth_element                     sort
partial_sort                    stable_sort
partial_sort_copy
```

Merge operations (Elements must be sorted)

```
inplace_merge                   merge
```

Set operations (Elements must be sorted)

```
includes                        set_symmetric_difference
set_difference                  set_union
set_intersection
```

Heap operations

```
make_heap                       push_heap
pop_heap                        sort_heap
```

Minimum and maximum

```
max                             min
max_element                     min_element
```

Permutation generators

```
next_permutation                prev_permutation
```

Algorithms by Category

Each algorithm requires certain kinds of iterators (for a description of the iterators and their capabilities see the ***Iterator*** entry in this manual). The following set of lists groups the algorithms according to the types of iterators they require.

Algorithms that use no iterators:

```
max                     min                 swap
```

Algorithms that require only input iterators:

```
accumulate              find                mismatch
count                   find_if
count_if                includes
equal                   inner_product
for_each                lexicographical_compare
```

Algorithms that require only output iterators:

```
fill_n                  generate_n
```

©Copyright 1999 Rogue Wave Software, Inc.

Algorithms that read from input iterators and write to output iterators:

```
adjacent_difference     replace_copy        transform
copy                    replace_copy_if     unique_copy
merge                   set_difference
partial_sum             set_intersection
remove_copy             set_symmetric_difference
remove_copy_if          set_union
```

Algorithms that require forward iterators:

```
adjacent_find           iter_swap           replace_if
binary_search           lower_bound         rotate
equal_range             max_element         search
fill                    min_element         search_n
find_end                remove              swap_ranges
find_first_of           remove_if           unique
generate                replace             upper_bound
```

Algorithms that read from forward iterators and write to output iterators:

```
rotate_copy
```

Algorithms that require bidirectional iterators

```
copy_backward           partition           stable_permutation
inplace_merge           prev_permutation
next_permutation        reverse
```

Algorithms that read from bidirectional iterators and write to output iterators:

```
reverse_copy
```

Algorithms that require random access iterators:

```
make_heap               pop_heap            sort
nth_element             push_heap           sort_heap
partial_sort            random_shuffle      stable_sort
```

Algorithms that read from input iterators and write to random access iterators:

```
partial_sort_copy
```

Complexity

The complexity for each of these algorithms is given in the manual page for that algorithm.

See Also

Manual pages for each of the algorithms named in the lists above.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

allocator

Summary

The default allocator object for storage management in Standard Library containers.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

const_pointer	reference
const_reference	size_type
difference_type	template <class U> struct rebind;
pointer	value_type

Member Functions

address()
allocate()
construct()
deallocate()
destroy()
max_size()

Synopsis

```
#include <memory>
template <class T>
class allocator;
```

Description

Containers in the Standard Library allow you control of storage management through the use of allocator objects. Each container has an allocator template parameter specifying the type of allocator to be used. Every constructor, except the copy constructor, has an allocator parameter, allowing you to pass in a specific allocator. A container uses that allocator for all storage management.

The library has a default allocator, called `allocator`. This allocator uses the global `new` and `delete` operators. By default, all containers use this allocator. You can also design your own allocator, but if you do so it must have an appropriate interface. The standard interface and an alternate interface are specified below. The alternate interface works on all supported compilers.

©Copyright 1999 Rogue Wave Software, Inc.

The Alternate Allocator

As of this writing, very few compilers support the full range of features needed by the standard allocator. If your compiler does not support member templates, both classes and functions, then you must use the alternate allocator interface. This alternate interface requires no special features of a compiler and offers most of the functionality of the standard allocator interface. The only thing missing is the ability to use special pointer and reference types. The alternate allocator fixes these as `T*` and `T&`. If your compiler supports partial specialization, then even this restriction is removed.

From outside a container, use of the alternate allocator is transparent. Simply pass the allocator as a template or function parameter exactly as you would pass the standard allocator.

Within a container, the alternate allocator interface is more complicated to use because it requires two separate classes, rather than one class with another class nested inside. If you plan to write your own containers and need to use the alternate allocator interface, we recommend that you support the default interface as well, since that is the only way to ensure long-term portability. See the User's Guide section on building containers for an explanation of how to support both the standard and the alternate allocator interfaces.

A generic allocator must be able to allocate space for objects of arbitrary type, and it must be able to construct those objects on that space. For this reason, the allocator must be type aware, but it must be aware on any arbitrary number of different types, since there is no way to predict the storage needs of any given container.

Consider an ordinary template. Although you may be able to instantiate on any fixed number of types, the resulting object is aware of only those types and any other types that can be built up from them (`T*`, for instance), as well as any types you specify up front. This won't work for an allocator, because you can't make any assumptions about the types a container needs to construct. It may well need to construct `T`s (or it may not), but it may also need to allocate node objects and other data structures necessary to manage the contents of the container. Clearly there is no way to predict what an arbitrary container might need to construct. As with everything else within the Standard Library, it is absolutely essential to be fully generic.

The Standard allocator interface solves the problem with member templates. The precise type you are going to construct is not specified when you create an allocator, but when you actually go to allocate space or construct an object on existing space.

The alternate allocator interface uses a different technique. The alternate interface breaks the allocator into two pieces: an interface and an implementation. The interface is a template class containing a pointer to an implementation. The implementation is a simple class providing raw un-typed storage. Anything can be constructed on it. The interface template types the raw storage based on the template parameter. Only the implementation object is passed into a container. The container constructs interface objects as necessary, using the implementation to manage the storage of data.

Since all interface objects use the one copy of the implementation object to allocate space, that one implementation object manages all storage acquisition for the container. The container makes calls to the ***allocator_interface*** objects in the same way it would make calls to a standard allocator object.

For example, if your container needs to allocate `T` objects and node objects, you need to have two ***allocator_interface*** objects in your container:

```
allocator_interface<Allocator,T> value_allocator;
allocator_interface<Allocator,node> node_allocator;
```

You then use the `value_allocator` for all allocation, construction, etc. of values (`T`s), and use the `node_allocator` object to allocate and deallocate nodes.

The only significant drawback is the lack of special pointer types and the inability to alter the behavior of the `construct` and `destroy` functions, since these must reside in the interface class. If your compiler has partial specialization, then this restriction goes away, since you can provide specialized interfaces along with your implementation.

Standard Interface

```
template <class T>
class allocator {
  typedef size_t           size_type;
  typedef ptrdiff_t        difference_type;
  typedef T*               pointer;
  typedef const T*         const_pointer;
  typedef T&               reference;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef const T&          const_reference;
  typedef T                 value_type;

  template <class U> struct rebind {
    typedef allocator<U> other;
  };
  allocator () throw();
  allocator (const allocator&) throw ();
  template <class U> allocator(const allocator<U>&) throw();
  template <class U>
    allocator& operator=(const allocator<U>&) throw();
  ~allocator () throw();
  pointer address (reference) const;
  const_pointer address (const_reference) const;
  pointer allocate (size_type,
     typename allocator<void::const_pointer = 0);
  void deallocate(pointer p, size_type n);
  size_type max_size () const;
  void construct (pointer, const T&);
  void destroy (pointer);
};
// specialize for void:
  template <> class allocator<void> {
  public:

    typedef void*       pointer;
    typedef const void* const_pointer;
    //  reference-to-void members are impossible.
    typedef void value_type;
    template <class U>
      struct rebind { typedef allocator<U> other; };
  };

// globals
template <class T, class U>
  bool operator==(const allocator<T>&,
                  const allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const allocator<T>&,
                  const allocator<U>&) throw();
```

Types

`size_type`

Type used to hold the size of an allocated block of storage.

`difference_type`

Type used to hold values representing distances between storage addresses.

`pointer`

Type of pointer returned by allocator.

`const_pointer`

`Const` version of `pointer`.

`reference`

Type of reference to allocated objects.

`const_reference`

`Const` version of `reference`.

`value_type`

Type of allocated object.

`template <class U> struct rebind;`

©Copyright 1999 Rogue Wave Software, Inc.

Converts an allocator templatized on one type to an allocator templatized on another type. This struct contains a single type member:

```
typedef allocator<U> other
```

Constructors

```
allocator()
```

Default constructor.

```
template <class U>
allocator(const allocator<U>&)
```

Copy constructor.

Destructors

```
~allocator()
```

Destructor.

Member Functions

```
pointer
address(reference x) const;
```

Returns the address of the reference `x` as a pointer.

```
const_pointer
address(const_reference x) const;
```

Returns the address of the reference `x` as a `const_pointer`.

```
pointer
allocate(size_type n,
         typename allocator<void>::const_pointer p = 0)
```

Allocates storage. Returns a pointer to the first element in a block of storage `n*sizeof(T)` bytes in size. The block is aligned appropriately for objects of type `T`. Throws the exception `bad_alloc` if the storage is unavailable. This function uses operator `new(size_t)`. The second parameter `p` can be used by an allocator to localize memory allocation, but the default allocator does not use it.

```
void
deallocate(pointer p, size_type n)
```

Deallocates the storage obtained by a call to `allocate` with arguments `n` and `p`.

```
size_type
max_size() const;
```

Returns the largest size for which a call to `allocate` might succeed.

```
void
construct(pointer p, const T& val);
```

Constructs an object of type `T2` with the initial value of `val` at the location specified by `p`. This function calls the `placement new` operator.

```
void
destroy(pointer p)
```

Calls the destructor on the object pointed to by `p`, but does not delete.

Alternate Interface

```
class allocator
{
public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
typedef size_t               size_type ;
typedef ptrdiff_t            difference_type ;
 allocator ();
  ~allocator ();
void * allocate (size_type, void * = 0);
void deallocate (void*);
};
template <class Allocator,class T>
class allocator_interface .
{
   public:
   typedef Allocator        allocator_type ;
   typedef T*               pointer ; .
   typedef const T*         const_pointer ;
   typedef T&               reference ; .
   typedef const T&         const_reference ;
   typedef T                value_type ; .
   typedef typename Allocator::size_type    size_type ;
   typedef typename Allocator::size_type    difference_type ;

protected:
   allocator_type*     alloc_;

public:
   allocator_interface ();
   allocator_interface (Allocator*);
   pointer address (T& x);
   size_type max_size () const;
   pointer allocate (size_type, pointer = 0);
   void deallocate (pointer);
   void construct (pointer, const T&);
   void destroy (T*);
};

//
// Specialization
//
class allocator_interface <allocator,void>
 {
 typedef void*                pointer ;
 typedef const void*          const_pointer ;
 };
```

Alternate Allocator Description

The description for the operations of ***allocator_interface<T>*** are generally the same as for corresponding operations of the standard allocator. The exception is that ***allocator_interface*** members `allocate` and `deallocate` call respective functions in ***allocator***, which are in turn implemented like the standard allocator functions.

See the ***container*** section of the Class Reference for a further description of how to use the alternate allocator within a user-defined container.

See Also

Containers

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Associative Containers

Summary

Associative containers are ordered containers. These containers include member functions that allow key insertion, retrieval, and manipulation. The standard library has the ***map***, ***multimap***, ***set***, and ***multiset*** associative containers. ***map*** and ***multimap*** associate values with the keys and allow for fast retrieval of the value, based upon fast retrieval of the key. ***set*** and ***multiset*** store only keys, allowing fast retrieval of the key itself.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

See Also

For more information about associative containers, see the ***Containers*** section of this reference guide, or see the section on the specific container.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

auto_ptr

`Memory Management`

Summary

A simple, smart pointer class.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

auto_ptr_ref
operator

Member Functions

get()
operator*()
operator->()
operator=()
release()
reset()

Synopsis

```
#include <memory>
template <class X> class auto_ptr;
```

Description

The template class ***auto_ptr*** holds onto a pointer obtained via `new` and deletes that object when the ***auto_ptr*** object itself is destroyed (such as when leaving block scope). ***auto_ptr*** can be used to make calls to operator new exception-safe. The ***auto_ptr*** class has semantics of strict ownership: an object may be safely pointed to by only one ***auto_ptr***, so copying an ***auto_ptr*** copies the pointer and transfers ownership to the destination if the source had already had ownership.

Interface

```
template <class X> class auto_ptr {
template <class Y> class auto_ptr_ref {
   public:
     const auto_ptr<Y>& p;
     auto_ptr_ref (const auto_ptr<Y>&);
};
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   public:
     typedef X element_type;
     // constructor/copy/destroy

     explicit auto_ptr (X* = 0) throw();
     auto_ptr (const auto_ptr<X>&) throw ();
     template <class Y>
       auto_ptr (const auto_ptr<Y>&) throw();
     void operator=(const auto_ptr<X>&) throw():
     template <class Y>
       void operator= (const auto_ptr<Y>&) throw();
     ~auto_ptr ();

     // members

     X& operator* () const throw();
     X* operator-> () const throw();
     X* get () const throw();
     X* release () throw();
     void reset (X*=0) throw();
     auto_ptr(auto_ptr_ref<X>) throw();
     template <class Y>
     operator auto_ptr_ref<Y>() throw();
     template <class Y>
     operator auto_ptr<Y>() throw();
 };
```

Types

```
template <class Y>
class auto_ptr_ref;
```

A private class template that holds a reference to an `auto_ptr`. It can only be constructed within an `auto_ptr` using a reference to an `auto_ptr`. It prevents unsafe copying.

Constructors

```
explicit
auto_ptr (X* p = 0);
```

Constructs an object of class `auto_ptr<X>`, initializing the held pointer to `p`, and acquiring ownership of that pointer. `p` must point to an object of class `X` or a class derived from `X` for which `delete p` is defined and accessible, or `p` must be a null pointer.

```
auto_ptr (const auto_ptr<X>& a);
template <class Y>
auto_ptr (const auto_ptr<Y>& a);
```

Constructs an object of class `auto_ptr<X>`, and copies the argument `a` to `*this`. If `a` owned the underlying pointer, then `*this` becomes the new owner of that pointer.

```
auto_ptr (const auto_ptr_ref<X> r);
```

Constructs an `auto_ptr` from an `auto_ptr_ref`.

Destructors

```
~auto_ptr ();
```

Deletes the underlying pointer.

Operators

```
void operator= (const auto_ptr<X>& a);
template <class Y>
void operator= (const auto_ptr<Y>& a);
```

©Copyright 1999 Rogue Wave Software, Inc.

Copies the argument `a` to `*this`. If `a` owned the underlying pointer, then `*this` becomes the new owner of that pointer. If `*this` already owned a pointer, then that pointer is deleted first.

```
X&
operator* () const;
```

Returns a reference to the object to which the underlying pointer points.

```
X*
operator-> () const;
```

Returns the underlying pointer.

```
template <class Y>
operator auto_ptr_ref<Y> ();
```

Constructs an `auto_ptr_ref` from `*this` and returns it.

```
template <class Y>
operator auto_ptr<Y> ();
```

Constructs a new `auto_ptr` using the underlying pointer held by `*this`. Calls `release()` on `*this`, so `*this` no longer possesses the pointer. Returns the new `auto_ptr`.

Member Functions

```
X*
get () const;
```

Returns the underlying pointer.

```
X*
release();
```

Releases ownership of the underlying pointer. Returns that pointer.

```
void
reset(X* p)
```

Sets the underlying pointer to `p`. If non-null, deletes the old underlying pointer.

Example

```
//
// auto_ptr.cpp
//
#include <iostream>
#include <memory>
using namespace std;

//
// A simple structure.
//
struct X
{
    X (int i = 0) : m_i(i) { }
    int get() const { return m_i; }
    int m_i;
};

int main ()
{
   //
   // b will hold a pointer to an X.
   //
   auto_ptr<X> b(new X(12345));
   //
   // a will now be the owner of the underlying pointer.
   //
   auto_ptr<X> a = b;
   //
   // Output the value contained by
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    // the underlying pointer.
    //
    cout << a->get() << endl;
    //
    // The pointer will be deleted when a is destroyed on
    // leaving scope.
    //
    return 0;
}
```

Program Output

```
12345
```

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

back_insert_iterator, back_inserter

`Insert Iterator`

Summary

An insert iterator used to insert items at the end of a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

container_type

Member Functions

back_inserter()
operator*()
operator++()
operator=()

Synopsis

```
#include <iterator>
template <class Container>
class back_insert_iterator;
```

Description

Insert iterators let you insert new elements into a collection rather than copy a new element's value over the value of an existing element. The class ***back_insert_iterator*** is used to insert items at the end of a collection. The function `back_inserter` creates an instance of a ***back_insert_iterator*** for a particular collection type. A ***back_insert_iterator*** can be used with ***vector***s, ***deque***s, and ***list***s, but not with ***map***s or ***set***s.

Interface

```
template <class Container>
 class back_insert_iterator
   : public iterator <output_iterator_tag, void, void, void,
     void> {

protected:
   Container* container;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
public:
   typedef Container container_type;
   explicit back_insert_iterator (Container&);
   back_insert_iterator<Container>&
    operator=
          (const typename Container::const_reference value);
   back_insert_iterator<Container>& operator* ();
   back_insert_iterator<Container>& operator++ ();
   back_insert_iterator<Container> operator++ (int);
};

template <class Container>
 back_insert_iterator<Container> back_inserter (Container&);
```

Types

container_type

The type of container acted on by this iterator.

Constructors

```
explicit
back_insert_iterator (Container& x);
```

Constructor. Creates an instance of a ***back_insert_iterator*** associated with container `x`.

Operators

```
back_insert_iterator<Container>&
operator= (const typename Container::value_type& value);
```

Inserts a copy of `value` on the end of the container, and returns `*this`.

```
back_insert_iterator<Container>&
operator* ();
```

Returns `*this`.

```
back_insert_iterator<Container>&
operator++ ();
back_insert_iterator<Container>
operator++ (int);
```

Increments the input iterator and returns `*this`.

Helper Functions

```
template <class Container>
back_insert_iterator<Container>
back_inserter (Container& x)
```

Returns a ***back_insert_iterator*** that inserts elements at the end of container `x`. This function allows you to create insert iterators inline.

Example

```
//
// ins_itr.cpp
//
 #include <iterator>
 #include <deque>
 #include <iostream>
 using namespace std;

 int main ()
 {
   //
   // Initialize a deque using an array.
   //
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   int arr[4] = { 3,4,7,8 };
   deque<int> d(arr+0, arr+4);
   //
   // Output the original deque.
   //
   cout << "Start with a deque: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // Insert into the middle.
   //
   insert_iterator<deque<int> > ins(d, d.begin()+2);
   *ins = 5; *ins = 6;
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use an insert_iterator: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // A deque of four 1s.
   //
   deque<int> d2(4, 1);
   //
   // Insert d2 at front of d.
   //
   copy(d2.begin(), d2.end(), front_inserter(d));
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use a front_inserter: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // Insert d2 at back of d.
   //
   copy(d2.begin(), d2.end(), back_inserter(d));
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use a back_inserter: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
Start with a deque:
     3 4 7 8
Use an insert_iterator:
     3 4 5 6 7 8
Use a front_inserter:
     1 1 1 1 3 4 5 6 7 8
Use a back_inserter:
     1 1 1 1 3 4 5 6 7 8 1 1 1 1
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

See Also

Insert Iterators

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_filebuf

Summary

Class that associates the input or output sequence with a file.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
filebuf
int_type
off_type
pos_type
traits_type
wfilebuf

Member Functions

close()
is_open()
open()
overflow()
pbackfail()
seekoff()
seekpos()
setbuf()
sync()
underflow()
xsputn()

Synopsis

```
#include <fstream>
template<class charT, class traits = char_traits<charT> >
class basic_filebuf
: public basic_streambuf<charT, traits>
```

Description

The template class ***basic_filebuf*** is derived from ***basic_streambuf***. It associates the input or output sequence with a file. Each object of type ***basic_filebuf<charT, traits>*** controls two character sequences:

- A character input sequence
- A character output sequence

©Copyright 1999 Rogue Wave Software, Inc.

The restrictions on reading and writing a sequence controlled by an object of class ***basic_filebuf<charT,traits>*** are the same as for reading and writing with the Standard C library files.

If the file is not open for reading, the input sequence cannot be read. If the file is not open for writing, the output sequence cannot be written. A joint file position is maintained for both the input and output sequences.

A file has byte sequences, so the ***basic_filebuf*** class treats a file as the external source (or sink) byte sequence. In order to provide the contents of a file as wide character sequences, a wide-oriented file buffer called wfilebuf converts wide character sequences to multibytes character sequences (and vice versa) according to the current locale being used in the stream buffer.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_filebuf
: public basic_streambuf<charT, traits> {

 public:

  typedef traits                      traits_type;
  typedef charT                       char_type;
  typedef typename traits::int_type   int_type;
  typedef typename traits::pos_type   pos_type;
  typedef typename traits::off_type   off_type;

  basic_filebuf();
  basic_filebuf(int fd);

  virtual ~basic_filebuf();

  bool is_open() const;
  basic_filebuf<charT, traits>* open(const char *s,
                                     ios_base::openmode,
                                     long protection = 0666);

  basic_filebuf<charT, traits>* open(int fd);
  basic_filebuf<charT, traits>* close();

 protected:

  virtual int      showmanyc();
  virtual int_type underflow();
  virtual int_type overflow(int_type c = traits::eof());
  virtual int_type pbackfail(int_type c = traits::eof());

  virtual basic_streambuf<charT,traits>*
    setbuf(char_type *s,streamsize n);

  virtual pos_type seekoff(off_type off,
                           ios_base::seekdir way,
                           ios_base::openmode which =
                           ios_base::in | ios_base::out);

  virtual pos_type seekpos(pos_type sp,
                           ios_base::openmode which =
                           ios_base::in | ios_base::out);
  virtual int sync();
  virtual streamsize xsputn(const char_type* s,
                            streamsize n);

};
```

Types

`char_type`

The type `char_type` is a synonym for the template parameter `charT`.

`filebuf`

The type `filebuf` is an instantiation of class `basic_filebuf` on type `char`:

©Copyright 1999 Rogue Wave Software, Inc.

```
typedef basic_filebuf<char> filebuf;
```

int_type

The type `int_type` is a synonym of type `traits::in_type`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wfilebuf

The type `wfilebuf` is an instantiation of class `basic_filebuf` on type `wchar_t`:

```
typedef basic_filebuf<wchar_t> wfilebuf;
```

Constructors

```
basic_filebuf();
```

Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()`.

```
basic_filebuf(int fd);
```

Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()`. It then calls `open(fd)` with file descriptor `fd`. This function is not described in the C++ standard, and is included as an extension to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptor.

Destructors

```
virtual ~basic_filebuf();
```

Calls `close()` and destroys the object.

Member Functions

```
basic_filebuf<charT,traits>*
close();
```

If `is_open() == false`, returns a null pointer. Otherwise, closes the file, and returns `*this`.

```
bool
is_open() const;
```

Returns `true` if the associated file is open.

```
basic_filebuf<charT,traits>*
open(const char* s, ios_base::openmode mode,
   long protection = 0666);
```

If `is_open() == true`, returns a null pointer. Otherwise opens the file, whose name is stored in the null-terminated byte-string `s`. The file open modes are given by their C-equivalent description (see the C function `fopen`):

```
in                      "w"
in|binary               "rb"
out                     "w"
out|app                 "a"
out|binary              "wb"
out|binary|app          "ab"
```

©Copyright 1999 Rogue Wave Software, Inc.

```
out|in                  "r+"
out|in|app              "a+"
out|in|binary           "r+b"
out|in|binary|app       "a+b"
trunc|out               "w"
trunc|out|binary        "wb"
trunc|out|in            "w+"
trunc|out|in|binary     "w+b"
```

The third argument, `protection`, is used as the file permission. It does not appear in the Standard C++ description of the function `open` and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS, since files are always readable and do not have special execute permission. If the `open` function fails, it returns a null pointer.

```
basic_filebuf<charT,traits>*
open(int fd);
```

Attaches the previously opened file, which is identified by its file descriptor `fd`, to the `basic_filebuf` object. This function is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors.

```
int_type
overflow(int_type c = traits::eof() );
```

If the output sequence has a put position available, and `c` is not `traits::eof()`, then writes `c` into it. If there is no position available, the function outputs the contents of the buffer to the associated file and then writes `c` at the new current put position. If the operation fails, the function returns `traits::eof()`. Otherwise it returns `traits::not_eof(c)`. In wide characters file buffer, `overflow` converts the internal wide characters to their external multibytes representation by using the `locale::codecvt` facet located in the locale object imbued in the stream buffer.

```
int_type
pbackfail(int_type c = traits::eof() );
```

Puts back the character designated by `c` into the input sequence. If `traits::eq_int_type(c,traits::eof())` returns `true`, moves the input sequence one position backward. If the operation fails, the function returns `traits::eof()`. Otherwise it returns `traits::not_eof(c)`.

```
pos_type
seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in |
        ios_base::out);
```

If the open mode is `in | out`, alters the stream position of both the input and the output sequence. If the open mode is `in`, alters the stream position of only the input sequence, and if it is `out`, alters the stream position of only the output sequence. The new position is calculated by combining the two parameters `off` (displacement) and `way` (reference point). If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position. File buffers using `locale::codecvt` facet and performing state dependent conversion support only seeking to the beginning of the file, to the current position, or to a position previously obtained by a call to one of the iostreams seeking functions.

```
pos_type
seekpos(pos_type sp,ios_base::openmode      which = ios_base::in | ios_base::out);
```

If the open mode is `in | out`, alters the stream position of both the input and the output sequence. If the open mode is `in`, alters the stream position of only the input sequence, and if it is `out`, alters the stream position of only the output sequence. If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position. File buffers using `locale::codecvt` facet performing state dependent conversion, only support seeking to the beginning of the file, to the current position, or to a position previously obtained by a call to one of the iostreams seeking functions.

```
basic_filebuf<charT,traits>*
setbuf(char_type*s, streamsize n);
```

If `s` is not a null pointer, outputs the content of the current buffer to the associated file, then deletes the current buffer and replaces it by `s`. Otherwise resizes the current buffer to size `n` after outputting its contents to the associated file, if necessary.

```
int
sync();
```

©Copyright 1999 Rogue Wave Software, Inc.

Synchronizes the contents of the external file, with its image maintained in memory by the file buffer. If the function fails, it returns `-1`; otherwise it returns `0`.

```
int_type
underflow();
```

If the input sequence has a read position available, returns the contents of this position. Otherwise fills up the buffer by reading characters from the associated file, and if it succeeds, returns the contents of the new current position. The function returns `traits::eof()` to indicate failure. In wide characters file buffer, `underflow` converts the external multibytes characters to their wide character representation by using the `locale::codecvt` facet located in the locale object imbued in the stream buffer.

```
streamsize
xsputn(const char_type* s, streamsize n);
```

Writes up to `n` characters to the output sequence. The characters written are obtained from successive elements of the array whose first element is designated by `s`. The function returns the number of characters written.

Example

```
//
// stdlib/examples/manual/filebuf.cpp
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // create a read/write file-stream object on tiny char
  // and attach it to the file "filebuf.out"
  ofstream out("filebuf.out",ios_base::in |
               ios_base::out);

  // tie the istream object to the ofstream object
  istream in(out.rdbuf());

  // output to out
  out << "Il errait comme un ame en peine";

  // seek to the beginning of the file
  in.seekg(0);

  // output in to the standard output
  cout << in.rdbuf() << endl;

  // close the file "filebuf.out"
  out.close();

  // open the existing file "filebuf.out"
  // and truncate it
  out.open("filebuf.out",ios_base::in | ios_base::out |
           ios_base::trunc);

  // set the buffer size
  out.rdbuf()->pubsetbuf(0,4096);

  // open the source code file
  ifstream ins("filebuf.cpp");

  //output it to filebuf.out
  out << ins.rdbuf();

  // seek to the beginning of the file
  out.seekp(0);

  // output the all file to the standard output
  cout << out.rdbuf();
}
```

See Also

©Copyright 1999 Rogue Wave Software, Inc.

char_traits(3C++),***ios_base***(3C++), ***basic_ios***(3C++), ***basic_streambuf***(3C++), ***basic_ifstream***(3C++), ***basic_ofstream***(3C++), ***basic_fstream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.8.1.1

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

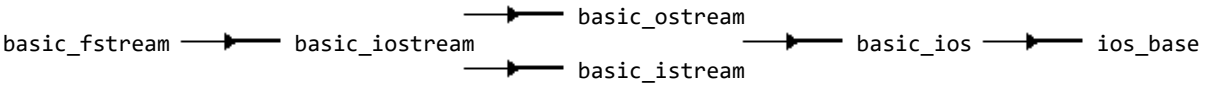
Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

basic_fstream



Summary

Supports reading and writing of named files or devices associated with a file descriptor.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	off_type
fstream	pos_type
int_type	traits_type
ios_type	wfstream

Member Functions

close()
is_open()
open()
rdbuf()

Synopsis

```
#include <fstream>
template<class charT, class traits = char_traits<charT> >
class basic_fstream
: public basic_iostream<charT, traits>
```

Description

The template class ***basic_fstream<charT,traits>*** supports reading and writing to named files or other devices associated with a file descriptor. It uses a `basic_filebuf` object to control the associated sequences. It inherits from ***basic_iostream*** and can therefore use all the formatted and unformatted input and output functions.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_fstream
```

©Copyright 1999 Rogue Wave Software, Inc.

```
: public basic_iostream<charT, traits> {

 public:

  typedef basic_ios<charT, traits>   ios_type;

  typedef charT                      char_type;
  typedef traits                     traits_type;
  typedef typename traits::int_type  int_type;
  typedef typename traits::pos_type  pos_type;
  typedef typename traits::off_type  off_type;

  basic_fstream();
  explicit basic_fstream(const char *s, ios_base::openmode
                         mode = ios_base::in |
                         ios_base::out,
                         long protection = 0666);

  explicit basic_fstream(int fd);
  basic_fstream(int fd, char_type *buf, int len);

  virtual ~basic_fstream();

  basic_filebuf<charT, traits> *rdbuf() const;
  bool is_open();
  void open(const char *s, ios_base::openmode mode =
            ios_base::in | ios_base::out,
            long protection = 0666);

  void close();

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

fstream

The type `fstream` is an instantiation of class `basic_fstream` on type `char`:

```
typedef basic_fstream<char> fstream;
```

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wfstream

The type `wfstream` is an instantiation of class `basic_fstream` on type `wchar_t`:

```
typedef basic_fstream<wchar_t> wfstream;
```

Constructors

©Copyright 1999 Rogue Wave Software, Inc.

```
basic_fstream();
```

Constructs an object of class ***basic_fstream<charT,traits>***, initializing the base class ***basic_iostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. After construction, a file can be attached to the ***basic_fstream*** object by using the `open()` member function.

```
basic_fstream(const char* s,
              ios_base::openmode mode=
              ios_base::in | iosw_base::out,
              long protection= 0666);
```

Constructs an object of class ***basic_fstream<charT,traits>***, initializing the base class ***basic_iostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the open function `open(s,mode,protection)` in order to attach the file, whose name is pointed to by `s`, to the ***basic_fstream*** object. The third argument, `protection`, is used as the file permission. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS since files are always readable and do not have special execute permission.

```
explicit basic_fstream(int fd);
```

Constructs an object of class ***basic_fstream<charT,traits>***, initializing the base class ***basic_iostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_fstream*** object. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. If the function fails, it sets `ios_base::failbit.`

```
basic_fstream(int fd, char_type* buf,int len);
```

Constructs an object of class ***basic_fstream<charT,traits>***, initializing the base class ***basic_iostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_fstream*** object. The underlying buffer is then replaced by calling the `basic_filebuf` member function, `setbuf()`, with parameters `buf` and `len`. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. It also maintains compatibility with the old iostreams library. If the function fails, it sets `ios_base::failbit.`

Destructors

```
virtual ~basic_fstream();
```

Destroys an object of class `basic_fstream`.

Member Functions

```
void
close();
```

Calls the associated `basic_filebuf` function `close()` and if this function fails, it calls the `basic_ios` member function `setstate(failbit)`.

```
bool
is_open();
```

Calls the associated `basic_filebuf` function `is_open()` and return its result.

```
void
open(const char* s,ios_base::openmode =
     ios_base::out | ios_base::in,
     long protection = 0666);
```

Calls the associated `basic_filebuf` function `open(s,mode,protection)` and, if this function fails at opening the file, calls the `basic_ios` member function `setstate(failbit)`. The third argument `protection` is used as the file permissions. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS since files are always readable and do not have special execute permission.

©Copyright 1999 Rogue Wave Software, Inc.

```
basic_filebuf<charT,traits>*
rdbuf() const;
```

Returns a pointer to the `basic_filebuf` associated with the stream.

Example

```
//
// stdlib/examples/manual/fstream.cpp
//
#include<iostream>
#include<fstream>
void main ( )
{
    using namespace std;

    // create a bi-directional fstream object
    fstream inout("fstream.out");

    // output characters
    inout << "Das ist die rede von einem man" << endl;
    inout << "C'est l'histoire d'un home" << endl;
    inout << "This is the story of a man" << endl;

    char p[100];

    // seek back to the beginning of the file
    inout.seekg(0);

    // extract the first line
    inout.getline(p,100);

    // output the first line to stdout
    cout << endl << "Deutch :" << endl;
    cout << p;

    fstream::pos_type pos = inout.tellg();

    // extract the second line
    inout.getline(p,100);

    // output the second line to stdout
    cout << endl << "Francais :" << endl;
    cout << p;

    // extract the third line
    inout.getline(p,100);

    // output the third line to stdout
    cout << endl << "English :" << endl;
    cout << p;

    // move the put sequence before the second line
    inout.seekp(pos);

    // replace the second line
    inout << "This is the story of a man" << endl;

    // replace the third line
    inout << "C'est l'histoire d'un home";

    // seek to the beginning of the file
    inout.seekg(0);

    // output the all content of the fstream
    // object to stdout
    cout << endl << endl << inout.rdbuf();
}
```

See Also

char_traits(3C++),*ios_base*(3C++), *basic_ios*(3C++), *basic_filebuf*(3C++), *basic_ifstream*(3C++), *basic_ofstream*(3C++)

©Copyright 1999 Rogue Wave Software, Inc.

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.8.1.11

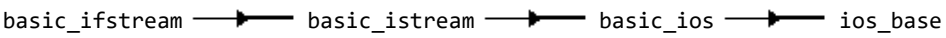
Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_ifstream



Summary

Supports reading from named files or other devices associated with a file descriptor.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	off_type
ifstream	pos_type
int_type	traits_type
ios_type	wifstream

Member Functions

close()
is_open()
open()
rdbuf()

Synopsis

```
#include <fstream>
template<class charT, class traits = char_traits<charT> >
class basic_ifstream
: public basic_istream<charT, traits>
```

Description

The template class ***basic_ifstream<charT,traits>*** supports reading from named files or other devices associated with a file descriptor. It uses a ***basic_filebuf*** object to control the associated sequences. It inherits from ***basic_istream*** and can therefore use all the formatted and unformatted input functions.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_ifstream
: public basic_istream<charT, traits> {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 public:

  typedef basic_ios<charT, traits>    ios_type;

  typedef traits                      traits_type;
  typedef charT                       char_type;
  typedef typename traits::int_type   int_type;
  typedef typename traits::pos_type   pos_type;
  typedef typename traits::off_type   off_type;

  basic_ifstream();

  explicit basic_ifstream(const char *s,
                          ios_base::openmode mode =
                          ios_base::in,
                          long protection = 0666);

  explicit basic_ifstream(int fd);
  basic_ifstream(int fd, char_type* buf, int len);

  virtual ~basic_ifstream();

  basic_filebuf<charT, traits> *rdbuf() const;
  bool is_open();
  void open(const char *s, ios_base::openmode mode =
            ios_base::in, long protection = 0666);

  void close();

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

ifstream

The type `ifstream` is an instantiation of class `basic_ifstream` on type `char`:

```
typedef basic_ifstream<char> ifstream;
```

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wifstream

The type `wifstream` is an instantiation of class `basic_ifstream` on type `wchar_t`:

```
typedef basic_ifstream<wchar_t> wifstream;
```

Constructors

basic_ifstream();

©Copyright 1999 Rogue Wave Software, Inc.

Constructs an object of class ***basic_ifstream<charT,traits>***, initializing the base class ***basic_istream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. After construction, a file can be attached to the ***basic_ifstream*** object by using the `open` member function.

```
basic_ifstream(const char* s,
               ios_base::openmode mode= ios_base::in,
               long protection= 0666);
```

Constructs an object of class ***basic_ifstream<charT,traits>***, initializing the base class ***basic_istream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the open function `open(s,mode,protection)` in order to attach the file, whose name is pointed to by `s`, to the ***basic_ifstream*** object. The third argument `protection` holds file permissions. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS since files are always readable and do not have special execute permission.

```
explicit basic_ifstream(int fd);
```

Constructs an object of class ***basic_ifstream<charT,traits>***, initializing the base class ***basic_istream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_ifstream*** object. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. If the function fails, it sets `ios_base::failbit.`

```
basic_ifstream(int fd, char_type* buf,int len);
```

Constructs an object of class ***basic_ifstream<charT,traits>***, initializing the base class ***basic_istream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_ifstream*** object. The underlying buffer is then replaced by calling the `basic_filebuf` member function `setbuf` with parameters `buf` and `len`. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. It also maintains compatibility with the old iostreams library. If the function fails, it sets `ios_base::failbit.`

Destructors

```
virtual ~basic_ifstream();
```

Destroys an object of class `basic_ifstream`.

Member Functions

```
void
close();
```

Calls the associated `basic_filebuf` function `close()` and if this function fails, it calls the `basic_ios` member function `setstate(failbit)`.

```
bool
is_open();
```

Calls the associated `basic_filebuf` function `is_open()` and returns its result.

```
void
open(const char* s,ios_base::openmode =
     ios_base::in, long protection = 0666);
```

Calls the associated `basic_filebuf` function `open(s,mode,protection)`. If this function fails opening the file, it calls the `basic_ios` member function `setstate(failbit)`. The third argument `protection` holds file permissions. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS since files are always readable and do not have special execute permission.

```
basic_filebuf<charT,traits>*
rdbuf() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a pointer to the `basic_filebuf` associated with the stream.

Example

```
//
// stdlib/examples/manual/ifstream.cpp
//
#include<iostream>
#include<fstream>
#include<iomanip>

void main ( )
{
  using namespace std;

  long   l= 20;
  const char *ntbs="Le minot passait la piece a frotter";
  char   c;
  char   buf[50];

try {

  // create a read/write file-stream object on char
  // and attach it to an ifstream object
  ifstream in("ifstream.out",ios_base::in |
              ios_base::out | ios_base::trunc);

  // tie the ostream object to the ifstream object
  ostream out(in.rdbuf());

  // output ntbs in out
  out << ntbs << endl;

  // seek to the beginning of the file
  in.seekg(0);

  // output each word on a separate line
  while ( in.get(c) )
   {
     if ( char_traits<char>::eq(c,' ') )
      cout << endl;
     else
      cout << c;
   }
  cout << endl << endl;

  // move back to the beginning of the file
  in.seekg(0);

  // clear the state flags
  in.clear();

  // does the same thing as the previous code
  // output each word on a separate line
  while ( in >> buf )
   cout << buf << endl;

  cout << endl << endl;

  // output the base info before each integer
  out << showbase;

  ostream::pos_type pos= out.tellp();

  // output l in hex with a field with of 20
  out << hex << setw(20) << l << endl;

  // output l in oct with a field with of 20
  out << oct << setw(20) << l << endl;

  // output l in dec with a field with of 20
  out << dec << setw(20) << l << endl;

  // move back to the beginning of the file
  in.seekg(0);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // output the all file
  cout << in.rdbuf();

  // clear the flags
  in.clear();

  // seek the input sequence to pos
  in.seekg(pos);

  int a,b,d;

  // read the previous outputted integer
  in >> a >> b >> d;

  // output 3 times 20
  cout << a << endl << b << endl << d << endl;

}
  catch( ios_base::failure& var )
   {
      cout << var.what();
   }

}
```

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***basic_filebuf***(3C++), ***basic_ofstream***(3C++), ***basic_fstream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.8.1.5

Standards Conformance

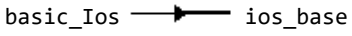
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_ios



Summary

A base class that includes the common functions required by all streams.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	off_type	traits_type
int_type	ostream_type	void*
ios	pos_type	wios
ios_type	streambuf_type	

Member Functions

bad()	fill()	rdbuf()
clear()	good()	rdstate()
copyfmt()	imbue()	setstate()
eof()	init()	tie()
exceptions()	narrow()	widen()
fail()	operator!()	

Synopsis

```
#include <ios>
template<class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base
```

Description

The class ***basic_ios*** is a base class that includes the common functions required by all streams. It maintains state information that reflects the integrity of the stream and stream buffer. It also maintains the link between the stream classes and the stream buffer classes via the `rdbuf` member functions. Classes derived from ***basic_ios*** specialize operations for input and output.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   public:

    typedef basic_ios<charT, traits>         ios_type;
    typedef basic_streambuf<charT, traits>   streambuf_type;
    typedef basic_ostream<charT, traits>     ostream_type;

    typedef typename traits::char_type       char_type;
    typedef traits                           traits_type;

    typedef typename traits::int_type        int_type;
    typedef typename traits::off_type        off_type;
    typedef typename traits::pos_type        pos_type;

    operator void*() const;
    bool operator!() const;

    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;

    void exceptions(iostate except);
    iostate exceptions() const;


    explicit basic_ios(basic_streambuf<charT, traits>
                       *sb_arg);
    virtual ~basic_ios();

    ostream_type *tie() const;
    ostream_type *tie(ostream_type *tie_arg);

    streambuf_type *rdbuf() const;
    streambuf_type *rdbuf( streambuf_type *sb);

    ios_type& copyfmt(const ios_type& rhs);

    char_type fill() const;
    char_type fill(char_type ch);

    locale imbue(const locale& loc);

    char narrow(charT, char) const;
    charT widen(char) const;

   protected:

    basic_ios();
    void init(basic_streambuf<charT, traits> *sb);
};
```

Types

char_type

The type `char_type` is a synonym of type `traits::char_type`.

ios

The type `ios` is an instantiation of `basic_ios` on `char`:

```
typedef basic_ios<char> ios;
```

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is a synonym for `basic_ios<charT, traits>` .

©Copyright 1999 Rogue Wave Software, Inc.

off_type

The type off_type is a synonym of type traits::off_type.

ostream_type

The type ostream_type is a synonym for basic_ostream<charT, traits> .

pos_type

The type pos_type is a synonym of type traits::pos_type.

streambuf_type

The type streambuf_type is a synonym for basic_streambuf<charT, traits> .

traits_type

The type traits_type is a synonym for the template parameter traits.

wios

The type wios is an instantiation of basic_ios on wchar_t:

```
typedef basic_ios<wchar_t> wios;
```

Public Constructors

```
explicit basic_ios(basic_streambuf<charT, traits>* sb);
```

Constructs an object of class basic_ios, assigning initial values to its member objects by calling init(sb). If sb is a null pointer, the stream is positioned in error state by triggering its badbit.

```
basic_ios();
```

Constructs an object of class basic_ios, leaving its member objects uninitialized. The object must be initialized by calling the init member function before using it.

Public Destructors

```
virtual ~basic_ios();
```

Destroys an object of class basic_ios.

Public Member Functions

```
bool
bad() const;
```

Returns true if badbit is set in rdstate().

```
void
clear(iostate state = goodbit);
```

If (state & exception()) == 0, returns. Otherwise, the function throws an object of class ios_base::failure. After the call returns state == rdstate() if rdbuf() !=0 otherwise state == state|ios_base::badbit.

```
basic_ios&
copyfmt(const basic_ios& rhs);
```

Assigns to the member objects of *this the corresponding member objects of rhs, except the following:

- rdstate() and rdbuf() are left unchanged
- Calls ios_base::copyfmt
- exceptions() is altered last by calling exceptions(rhs.exceptions())

©Copyright 1999 Rogue Wave Software, Inc.

```
bool
eof() const;
```

Returns `true` if `eofbit` is set in `rdstate()`.

```
iostate
exceptions() const;
```

Returns a mask that determines what elements set in `rdstate()` cause exceptions to be thrown.

```
void
exceptions(iostate except);
```

Sets the exception mask to `except`, then calls `clear(rdstate())`.

```
bool
fail() const;
```

Returns `true` if failbit or badbit is set in `rdstate()`.

```
char_type
fill() const;
```

Returns the character used to pad (fill) an output conversion to the specified field width.

```
char_type
fill(char_type fillch);
```

Saves the field width value, then replaces it by `fillch` and returns the previously saved value.

```
bool
good() const;
```

Returns `rdstate() == 0`.

```
locale
imbue(const locale& loc);
```

Saves the value returned by `getloc()`, then assigns `loc` to a private variable. If `rdbuf() != 0` calls `rdbuf()->pubimbue(loc)` and returns the previously saved value.

```
void
init(basic_streambuf<charT,traits>* sb);
```

Performs the following initialization:

```
rdbuf()      sb
tie()        0
rdstate()    goodbit if sb is not null otherwise badbit
exceptions() goodbit
flags()      skipws | dec
width()      0
precision()  6
fill()       the space character
getloc()     a copy of the locale returned by locale::locale()
```

```
char
narrow(charT c, char dfault) const;
```

Uses the stream's locale to convert the wide character `c` to a tiny character, and then returns it. If no conversion exists, it returns the character `dfault`.

```
bool
operator!() const;
```

Returns `fail() ? 1 : 0;`

```
streambuf_type*
rdbuf() const;
```

Returns a pointer to the stream buffer associated with the stream.

©Copyright 1999 Rogue Wave Software, Inc.

```
streambuf_type*
rdbuf(streambuf_type* sb);
```

Associates a stream buffer with the stream. All the input and output is directed to this stream buffer. If `sb` is a null pointer, the stream is positioned in error state, by triggering its `badbit`.

```
iostate
rdstate() const;
```

Returns the control state of the stream.

```
void
setstate(iostate state);
```

Calls `clear(rdstate() | state)`.

```
ostream_type*
tie() const;
```

Returns an output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

```
ostream_type*
tie(ostream_type* tiestr);
```

Saves the `tie()` value, then replaces it by `tiestr` and returns the value previously saved.

```
operator
void*() const;
```

Returns `fail() ? 0 : 1;`

```
charT
widen(char c) const;
```

Uses the stream's locale to convert the tiny character `c` to a wide character, then returns it.

See Also

ios_base(3C++), ***basic_istream***(3C++), ***basic_ostream***(3C++), ***basic_streambuf***(3C++), ***char_traits***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, section 27.4.5.

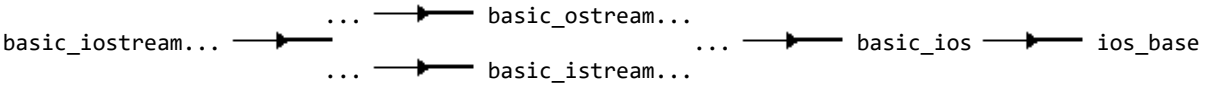
Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

basic_iostream



Summary

Assists in formatting and interpreting sequences of characters controlled by a stream buffer.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <istream>
template<class charT, class traits = char_traits<charT> >
class basic_iostream
: public basic_istream<charT, traits>,
public basic_ostream<charT, traits>
```

Description

The class ***basic_iostream*** inherits a number of functions from classes ***basic_ostream<charT, traits>*** and ***basic_istream<charT, traits>.*** They assist in formatting and interpreting sequences of characters controlled by a stream buffer. Two groups of functions share common properties, the formatted functions and the unformatted functions.

Interface

```
template<class charT, class traits>
class basic_iostream
: public basic_istream<charT, traits>,
  public basic_ostream<charT, traits>

{

 public:

  explicit basic_iostream(basic_streambuf<charT, traits>
                          *sb);
  virtual ~basic_iostream();
};
```

Public Constructors

©Copyright 1999 Rogue Wave Software, Inc.

```
explicit basic_iostream(basic_streambuf<charT, traits>
                        *sb);
```

Constructs an object of class `basic_iostream`, assigning initial values to the base class by calling `basic_istream<charT, traits>(sb)` and `basic_ostream<charT, traits>(sb)`.

Destructors

```
virtual ~basic_iostream();
```

Destroys an object of class `basic_iostream`.

Example

See ***basic_istream*** and ***basic_ostream*** examples.

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***basic_streambuf***(3C++), ***basic_istream***(3C++), ***basic_ostream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.6.1.5.

Standards Conformance

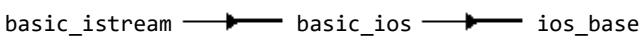
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_istream



Summary

Assists in reading and interpreting input from sequences controlled by a stream buffer.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	istream_type	
int_type	off_type	traits_type
ios_type	pos_type	wistream
istream	streambuf_type	

Member Functions

bool()	peek()	sync()
gcount()	putback()	tellg()
get()	read()	unget()
getline()	readsome()	ws()
ignore()	seekg()	~sentry()
operator>>()	sentry()	

Synopsis

```
#include <istream>
template<class charT, class traits = char_traits<charT> >
class basic_istream
: virtual public basic_ios<charT, traits>
```

Description

The class ***basic_istream*** defines member functions that assist in reading and interpreting input from sequences controlled by a stream buffer.

Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions obtain (or extract) input characters from

©Copyright 1999 Rogue Wave Software, Inc.

basic_streambuf. They both begin by constructing an object of class ***basic_istream::sentry*** and, if this object is in `good` state after construction, the function obtains the requested input. The `sentry` object performs exception safe initialization, such as controlling the status of the stream or locking it in a multithread environment.

Some formatted input functions parse characters extracted from the input sequence, converting the result to a value of some scalar data type, and storing the converted value in an object of that scalar type. The conversion behavior depends directly on the locale object being imbued in the stream.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_istream
: virtual public basic_ios<charT, traits> {

public:

 typedef basic_istream<charT, traits>    istream_type;
 typedef basic_ios<charT, traits>        ios_type;
 typedef basic_streambuf<charT, traits>  streambuf_type;

 typedef traits                      traits_type;
 typedef charT                       char_type;
 typedef typename traits::int_type   int_type;
 typedef typename traits::pos_type   pos_type;
 typedef typename traits::off_type   off_type;

 explicit basic_istream(basic_streambuf<charT, traits> *sb);
 virtual ~basic_istream();

 class sentry
  {
     public:
      inline explicit sentry(basic_istream<charT,traits>&,
                             bool noskipws = 0);
       ~sentry();
       operator bool ();
  };

 istream_type& operator>>(istream_type&
                          (*pf)(istream_type&));
 istream_type& operator>>(ios_base& (*pf)(ios_base&));
 istream_type& operator>>(ios_type& (*pf)(ios_type&));

 istream_type& operator>>(bool& n);
 istream_type& operator>>(short& n);
 istream_type& operator>>(unsigned short& n);
 istream_type& operator>>(int& n);
 istream_type& operator>>(unsigned int& n);
 istream_type& operator>>(long& n);
 istream_type& operator>>(unsigned long& n);
 istream_type& operator>>(float& f);
 istream_type& operator>>(double& f);
 istream_type& operator>>(long double& f);

 istream_type& operator>>(void*& p);

 istream_type& operator>>(streambuf_type& sb);
 istream_type& operator>>(streambuf_type *sb);

 streamsize gcount() const;
 int_type get();
 istream_type& get(char_type& c);
 istream_type& get(char_type *s, streamsize n);
 istream_type& get(char_type *s, streamsize n,
                   char_type delim);

 istream_type& get(streambuf_type& sb);
 istream_type& get(streambuf_type& sb,char_type delim);

 istream_type& getline(char_type *s, streamsize n);
 istream_type& getline(char_type *s, streamsize n,
                       char_type delim);

 istream_type& ignore(streamsize n = 1,
                      int_type delim = traits::eof());
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 int peek();
 istream_type& read(char_type *s, streamsize n);
 streamsize readsome(char_type *s, streamsize n);


 istream_type& putback(char_type c);
 istream_type& unget();
 int sync();

 pos_type tellg();
 istream_type& seekg(pos_type&);
 istream_type& seekg(off_type&, ios_base::seekdir);

};

//global character extraction templates

  template<class charT, class traits>
  basic_istream<charT, traits>&
  ws(basic_istream<charT, traits>& is);

template<class charT, class traits>
basic_istream<charT, traits>&
operator>> (basic_istream<charT, traits>&, charT&);

template<class charT, class traits>
basic_istream<charT, traits>&
operator>> (basic_istream<charT, traits>&, charT*);

template<class traits>
basic_istream<char, traits>&
operator>> (basic_istream<char, traits>&, unsigned char&);

template<class traits>
basic_istream<char, traits>&
operator>> (basic_istream<char, traits>&, signed char&);

template<class traits>
basic_istream<char, traits>&
operator>> (basic_istream<char, traits>&, unsigned char*);

template<class traits>
basic_istream<char, traits>&
operator>> (basic_istream<char, traits>&, signed char*);
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is a synonym for `basic_ios<charT, traits>` .

istream

The type `istream` is an instantiation of class `basic_istream` on type `char`:

```
typedef basic_istream<char> istream;
```

istream_type

The type `istream_type` is a synonym for `basic_istream<charT, traits>`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

©Copyright 1999 Rogue Wave Software, Inc.

pos_type

The type pos_type is a synonym of type traits::pos_type.

streambuf_type

The type streambuf_type is a synonym for basic_streambuf<charT, traits> .

traits_type

The type traits_type is a synonym for the template parameter traits.

wistream

The type wistream is an instantiation of class basic_istream on type wchar_t:

```
typedef basic_istream<wchar_t> wistream;
```

Public Constructors

```
explicit basic_istream(basic_streambuf<charT, traits>* sb);
```

Constructs an object of class basic_istream, assigning initial values to the base class by calling basic_ios::init(sb).

Public Destructors

```
virtual ~basic_istream();
```

Destroys an object of class basic_istream.

Sentry Classes

```
explicit sentry(basic_istream<charT,traits>&,
                bool noskipws=0);
```

Prepares for formatted or unformatted input. If the basic_ios member function tie() is not a null pointer, the function synchronizes the output sequence with any associated stream. If noskipws is zero and the ios_base member function flags() & skipws is nonzero, the function extracts and discards each character as long as the next available input character is a white space character. If after any preparation is completed, the basic_ios member function good() is true, the sentry conversion function operator bool() returns true. Otherwise it returns false. In a multithread environment the sentry object constructor is responsible for locking the stream and the stream buffer associated with the stream.

```
~sentry();
```

Destroys an object of class ***sentry***. In a multithread environment, the sentry object destructor is responsible for unlocking the stream and the stream buffer associated with the stream.

```
operator bool();
```

If after any preparation is completed, the basic_ios member function good() is true, the sentry conversion function operator bool() returns true else it returns false.

Extractors

```
istream_type&
operator>>(istream_type&
          (*pf) (istream_type&));
```

Calls pf(*this), then returns *this.

```
istream_type&
operator>>(ios_type& (*pf) (ios_type&));
```

Calls pf(*this), then returns *this.

©Copyright 1999 Rogue Wave Software, Inc.

```
istream_type&
operator>>(ios_base& (*pf) (ios_base&));
```

Calls `pf(*this)`, then returns `*this`.

```
istream_type&
operator>>(bool& n);
```

Converts a Boolean value, if one is available, and stores it in `n`. If the `ios_base` member function `flag() & ios_base::boolalpha` is `false`, it tries to read an integer value, which if found must be `0` or `1`. If the `boolalpha` flag is `true`, it reads characters until it determines whether the characters read are correct according to the locale function `numpunct<>::truename()` or `numpunct<>::falsename()`. If no match is found, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`.

```
istream_type&
operator>>(short& n);
```

Converts a `signed short` integer, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(unsigned short& n);
```

Converts an `unsigned short` integer, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(int& n);
```

Converts a `signed integer`, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(unsigned int& n);
```

Converts an `unsigned integer`, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(long& n);
```

Converts a `signed long` integer, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(unsigned long& n);
```

Converts an `unsigned long` integer, if one is available, and stores it in `n`, then returns `*this`.

```
istream_type&
operator>>(float& f);
```

Converts a `float`, if one is available, and stores it in `f`, then returns `*this`.

```
istream_type&
operator>>(double& f);
```

Converts a `double`, if one is available, and stores it in `f`, then returns `*this`.

```
istream_type&
operator>>(long double& f);
```

Converts a `long double`, if one is available, and stores it in `f`, then returns `*this`.

```
istream_type&
operator>>(void*& p);
```

Extracts a `void` pointer, if one is available, and stores it in `p`, then returns `*this`.

```
istream_type&
operator>>(streambuf_type* sb);
```

If `sb` is null, calls the `basic_ios` member function `setstate(badbit)`, which may throw `ios_base::failure`. Otherwise extracts characters from `*this` and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

©Copyright 1999 Rogue Wave Software, Inc.

- An end-of-file on the input sequence
- A failure when inserting in the output sequence
- An exception

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. If failure was due to catching an exception thrown while extracting characters from `sb` and `failbit` is on in `exception()`, then the caught exception is rethrown.

```
istream_type&
operator>>(streambuf_type& sb);
```

Extracts characters from `*this` and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

- An end-of-file on the input sequence
- A failure when inserting in the output sequence
- An exception

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. If failure was due to catching an exception thrown while extracting characters from `sb` and `failbit` is on in `exception()`, then the caught exception is rethrown.

Unformatted Functions

```
streamsize
gcount() const;
```

Returns the number of characters extracted by the last unformatted input member function called.

```
int_type
get();
```

Extracts a character, if one is available. Otherwise, the function calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. Returns the character extracted or returns `traits::eof()`, if none is available.

```
istream_type&
get(char_type& c);
```

Extracts a character, if one is available, and assigns it to `c`. Otherwise, the function calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`.

```
istream_type&
get(char_type* s, streamsize n,char_type delim);
```

Extracts characters and stores them into successive locations of an array whose first element is designated by `s`. Characters are extracted and stored until any of the following occurs:

- `n`-1 characters are stored
- An end-of-file on the input sequence
- The next available input character `== delim`.

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. In any case, it stores a null character into the next successive location of the array.

```
istream_type&
get(char_type* s, streamsize n);
```

Calls `get(s,n,widen("\n"))`.

```
istream_type&
get(streambuf_type& sb,char_type delim);
```

©Copyright 1999 Rogue Wave Software, Inc.

Extracts characters and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

- An end-of-file on the input sequence
- A failure when inserting in the output sequence
- The next available input character `== delim`.
- An exception

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. If failure was due to catching an exception thrown while extracting characters from `sb` and `failbit` is on in `exception()`, then the caught exception is rethrown.

```
istream_type&
get(streambuf_type& sb);
```

Calls `get(sb,widen("\n"))`.

```
istream_type&
getline(char_type* s, streamsize n, char_type delim);
```

Extracts characters and stores them into successive locations of an array whose first element is designated by `s`. Characters are extracted and stored until any of the following occurs:

- `n`-1 characters are stored
- An end-of-file on the input sequence
- The next available input character `== delim`.

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. In any case, it stores a null character into the next successive location of the array.

```
istream_type&
getline(char_type* s, streamsize n);
```

Calls `getline(s,n,widen("\n"))`.

```
istream_type&
ignore(streamsize n=1, int_type delim=traits::eof());
```

Extracts characters and discards them. Characters are extracted until any of the following occurs:

- `n` characters are extracted
- An end-of-file on the input sequence
- The next available input character `== delim`.

```
int_type
peek();
```

Returns `traits::eof()` if the `basic_ios` member function `good()` returns `false`. Otherwise, returns the next available character. Does not increment the current get pointer.

```
istream_type&
putback(char_type c);
```

Inserts `c` in the putback sequence.

```
istream_type&
read(char_type* s, streamsize n);
```

Extracts characters and stores them into successive locations of an array whose first element is designated by `s`. Characters are extracted and stored until any of the following occurs:

- `n` characters are stored

©Copyright 1999 Rogue Wave Software, Inc.

- An end-of-file on the input sequence

If the function does not store `n` characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`.

```
streamsize
readsome(char_type* s, streamsize n);
```

Extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `rdbuf()->in_avail() == -1`, calls the `basic_ios` member function `setstate(eofbit)`.

- If `rdbuf()->in_avail() == 0`, extracts no characters
- If `rdbuf()->in_avail() > 0`, extracts

  ```
  min( rdbuf()->in_avail(), n)
  ```

In any case the function returns the number of characters extracted.

```
istream_type&
seekg(pos_type& pos);
```

If the `basic_ios` member function `fail()`, returns `false`, executes `rdbuf()->pubseekpos(pos)`, which positions the current pointer of the input sequence at the position designated by `pos`.

```
istream_type&
seekg(off_type& off, ios_base::seekdir dir);
```

If the `basic_ios` member function `fail()` returns `false`, executes `rdbuf()->pubseekpos(off,dir)`, which positions the current pointer of the input sequence at the position designated by `off` and `dir`.

```
int
sync();
```

If `rdbuf()` is a null pointer, return `-1`. Otherwise, calls `rdbuf()->pubsync()` and if that function returns `-1` calls the `basic_ios` member function `setstate(badbit)`. The purpose of this function is to synchronize the internal input buffer, with the external sequence of characters.

```
pos_type
tellg();
```

If the `basic_ios` member function `fail()` returns `true`, `tellg()` returns `pos_type(off_type(-1))` to indicate failure. Otherwise it returns the current position of the input sequence by calling `rdbuf()->pubseekoff(0,cur,in)`.

```
istream_type&
unget();
```

If `rdbuf()` is not null, calls `rdbuf()->sungetc()`. If `rdbuf()` is null or if `sungetc()` returns `traits::eof()`, calls the `basic_ios` member function `setstate(badbit)`.

Non-member Functions

```
template<class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, charT& c);
```

Extracts a character if one is available, and stores it in `c`. Otherwise the function calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`.

```
template<class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, charT* s);
```

Extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If the `ios_base` member function `is.width()` is greater than zero, then `is.width()` is the maximum number of characters stored. Characters are extracted and stored until any of the following occurs:

- If `is.width()>0`, `is.width()-1` characters are extracted

©Copyright 1999 Rogue Wave Software, Inc.

- An end-of-file on the input sequence
- The next available input character is a white space

If the function stores no characters, it calls the `basic_ios` member function `setstate(failbit)`, which may throw `ios_base::failure`. In any case, it then stores a null character into the next successive location of the array and calls `width(0)`.

```
template<class traits>
basic_istream<char, traits>&
operator>>(basic_istream<char, traits>& is,
            unsigned char& c);
```

Returns `is >> (char&)c`.

```
template<class traits>
basic_istream<char, traits>&
operator>>(basic_istream<char, traits>& is,
           signed char& c);
```

Returns `is >> (char&)c`.

```
template<class traits>
basic_istream<char, traits>&
operator>>(basic_istream<char, traits>& is,
           unsigned char* c);
```

Returns `is >> (char*)c`.

```
template<class traits>
basic_istream<char, traits>&
operator>>(basic_istream<char, traits>& is,
           signed char* c);
```

Returns `is >> (char*)c`.

```
template<class charT, class traits>
basic_istream<charT, traits>&
ws(basic_istream<charT, traits>& is);
```

Skips any white space in the input sequence and returns `is`.

Example

```
//
// stdlib/examples/manual/istream1.cpp
//
#include<iostream>
#include<istream>
#include<fstream>

void main ( )
{
  using namespace std;

  float f= 3.14159;
  int   i= 3;
  char  s[200];

  // open a file for read and write operations
  ofstream out("example", ios_base::in | ios_base::out
               | ios_base::trunc);

  // tie the istream object to the ofstream filebuf
  istream in (out.rdbuf());

  // output to the file
  out << "Annie is the Queen of porting" << endl;
  out << f << endl;
  out << i << endl;

  // seek to the beginning of the file
  in.seekg(0);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  f = i = 0;

  // read from the file using formatted functions
  in >> s >> f >> i;

  // seek to the beginning of the file
  in.seekg(0,ios_base::beg);

  // output the all file to the standard output
  cout << in.rdbuf();

  // seek to the beginning of the file
  in.seekg(0);

  // read the first line in the file
  // "Annie is the Queen of porting"
  in.getline(s,100);

  cout << s << endl;

  // read the second line in the file
  // 3.14159
  in.getline(s,100);

  cout << s << endl;

  // seek to the beginning of the file
  in.seekg(0);

  // read the first line in the file
  // "Annie is the Queen of porting"
  in.get(s,100);

  // remove the newline character
  in.ignore();

  cout << s << endl;

  // read the second line in the file
  // 3.14159
  in.get(s,100);

  cout << s << endl;

  // remove the newline character
  in.ignore();

  // store the current file position
  istream::pos_type position = in.tellg();

  out << "replace the int" << endl;

  // move back to the previous saved position
  in.seekg(position);

  // output the remain of the file
  // "replace the int"
  // this is equivalent to
  // cout << in.rdbuf();
  while( !char_traits<char>::eq_int_type(in.peek(),
          char_traits<char>::eof()) )
   cout << char_traits<char>::to_char_type(in.get());

  cout << "\n\n\n" << flush;
}
//
// istream example #2
//
#include <iostream>

void main ( )
{
  using namespace std;

  char p[50];
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // remove all the white spaces
  cin >> ws;

  // read characters from stdin until a newline
  // or 49 characters have been read
  cin.getline(p,50);

  // output the result to stdout
  cout << p;
}
```

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***basic_streambuf***(3C++), ***basic_iostream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.6.1

Standards Conformance

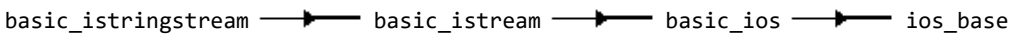
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

basic_istringstream



Summary

Supports reading objects of class ***basic_string<charT,traits,Allocator>*** from an array in memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	off_type	
int_type	pos_type	traits_type
ios_type	sb_type	wistringstream
istringstream	string_type	

Member Functions

rdbuf()
str()

Synopsis

```
#include <sstream>
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_istringstream
: public basic_istream<charT, traits>
```

Description

The template class ***basic_istringstream<charT,traits,Allocator>*** reads from an array in memory. It supports reading objects of class ***basic_string<charT,traits,Allocator>***. It uses a `basic_stringbuf` object to control the associated storage. It inherits from ***basic_istream*** and therefore can use all the formatted and unformatted input functions.

Interface

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<void> >
class basic_istringstream
: public basic_istream<charT, traits> {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 public:

  typedef basic_stringbuf<charT, traits, Allocator> sb_type;
  typedef basic_ios<charT, traits> ios_type;
  typedef basic_string<charT, traits, Allocator>
                       string_type;

  typedef traits                            traits_type;
  typedef charT                             char_type;
  typedef typename traits::int_type         int_type;
  typedef typename traits::pos_type         pos_type;
  typedef typename traits::off_type         off_type;

  explicit basic_istringstream(ios_base::openmode which =
                               ios_base::in);

  explicit basic_istringstream(const string_type& str,
                               ios_base::openmode which =
                               ios_base::in);

  virtual ~basic_istringstream();
  basic_stringbuf<charT,traits,Allocator> *rdbuf() const;
  string_type str() const;
  void str(const string_type& str);

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

istringstream

The type `istringstream` is an instantiation of class `basic_istringstream` on type `char`:

```
typedef basic_istringstream<char> istringstream;
```

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

sb_type

The type `sb_type` is an instantiation of class `basic_stringbuf` on type `charT`.

string_type

The type `string_type` is an instantiation of class `basic_string` on type `charT`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wistringstream

The type `wistringstream` is an instantiation of class `basic_istringstream` on type `wchar_t`:

```
typedef basic_istringstream<wchar_t> wistringstream;
```

©Copyright 1999 Rogue Wave Software, Inc.

Constructors

```
explicit basic_istringstream(ios_base::openmode which =
                             ios_base::in);
```

Constructs an object of class `basic_istringstream`, initializing the base class `basic_istream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(which)`.

```
explicit basic_istringstream(const string_type& str,
                             ios_base::openmode which =
                             ios_base::in);
```

Constructs an object of class `basic_istringstream`, initializing the base class `basic_istream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(str,which)`.

Destructors

```
virtual ~basic_istringstream();
```

Destroys an object of class `basic_istringstream`.

Member Functions

```
basic_stringbuf<charT,traits,Allocator>*
rdbuf() const;
```

Returns a pointer to the `basic_stringbuf` associated with the stream.

```
string_type
str() const;
```

Returns a string object of type `string_type`, which contains a copy of the underlying buffer contents.

```
void
str(const string_type& str);
```

Clears the string buffer and copies the string object `str` into it. If the opening mode is `in`, initializes the input sequence to point to the first character of the buffer. If the opening mode is `out`, initializes the output sequence to point to the first character of the buffer. If the opening mode is `out | app`, initializes the output sequence to point to the last character of the buffer.

Example

```
//
// stdlib/examples/manual/istringstream.cpp
//
#include<iostream>
#include<sstream>
#include<string>
#include<iomanip>

void main ( )
{
  using namespace std;

  long   l= 20;
  wchar_t *ntbs=L"Il avait l'air heureux";
  wchar_t c;
  wchar_t buf[50];

  // create a read/write string-stream object on wide char
  // and attach it to an wistringstream object
  wistringstream in(ios_base::in | ios_base::out);

  // tie the ostream object to the wistringstream object
  wostream out(in.rdbuf());

  // output ntbs in out
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  out << ntbs;

  // output each word on a separate line
  while ( in.get(c) )
   {
     if ( c == L' ' )
      wcout << endl;
     else
      wcout << c;
   }
  wcout << endl << endl;

  // move back the input sequence to the beginning
  in.seekg(0);

  // clear the state flags
  in.clear();

  // does the same thing as the previous code
  // output each word on a separate line
  while ( in >> buf )
   wcout << buf << endl;

  wcout << endl << endl;

  // create a tiny string object
  string test_string("Il dormait pour l'eternite");

  // create a read/write string-stream object on char
  // and attach it to an istringstream object
  istringstream in_bis(ios_base:: in | ios_base::out |
                       ios_base::app );

  // create an ostream object
  ostream out_bis(in_bis.rdbuf());

  // initialize the string buffer with test_string
  in_bis.str(test_string);

  out_bis << endl;

  // output the base info before each integer
  out_bis << showbase;

  ostream::pos_type pos= out_bis.tellp();

  // output l in hex with a field with of 20
  out_bis << hex << setw(20) << l << endl;

  // output l in oct with a field with of 20
  out_bis << oct << setw(20) << l << endl;

  // output l in dec with a field with of 20
  out_bis << dec << setw(20) << l << endl;

  // output the all buffer
  cout << in_bis.rdbuf();

  // seek the input sequence to pos
  in_bis.seekg(pos);

  int a,b,d;

  // read the previous outputted integer
  in_bis >> a >> b >> d;

  // output 3 times 20
  cout << a << endl << b << endl << d << endl;

}
```

See Also

char_traits(3C++), *ios_base*(3C++), *basic_ios*(3C++), *basic_stringbuf*(3C++), *basic_string*(3C++), *basic_ostringstream*(3C++), *basic_stringstream*(3C++)

©Copyright 1999 Rogue Wave Software, Inc.

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.7.2

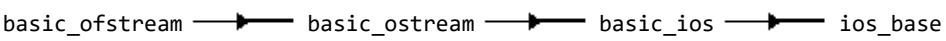
Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_ofstream



Summary

Supports writing into named files or other devices associated with a file descriptor.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	ofstream
int_type	pos_type
ios_type	traits_type
off_type	wofstream

Member Functions

close()
is_open()
open()
rdbuf()

Synopsis

```
#include <fstream>
template<class charT, class traits = char_traits<charT> >
class basic_ofstream
: public basic_ostream<charT, traits>
```

Description

The template class ***basic_ofstream<charT,traits>*** supports writing into named files or other devices associated with a file descriptor. It uses a `basic_filebuf` object to control the associated sequences. It inherits from ***basic_ostream*** and can therefore use all the formatted and unformatted output functions.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_ofstream
: public basic_ostream<charT, traits> {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 public:

  typedef charT                       char_type;
  typedef traits                      traits_type;
  typedef typename traits::int_type   int_type;
  typedef typename traits::pos_type   pos_type;
  typedef typename traits::off_type   off_type;

  typedef basic_ios<charT, traits>    ios_type;

  basic_ofstream();
  explicit basic_ofstream(const char *s,
                          ios_base::openmode mode =
                          ios_base::out,
                          long protection = 0666);

  explicit basic_ofstream(int fd);
  basic_ofstream(int fd, char_type* buf, int len);

  virtual ~basic_ofstream();

  basic_filebuf<charT, traits> *rdbuf() const;

  bool is_open();
  void open(const char *s, ios_base::openmode mode =
            ios_type::out, long protection = 0666);

  void close();

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

ofstream

The type `ofstream` is an instantiation of class `basic_ofstream` on type `char`:

```
typedef basic_ofstream<char> ofstream;
```

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wofstream

The type `wofstream` is an instantiation of class `basic_ofstream` on type `wchar_t`:

```
typedef basic_ofstream<wchar_t> wofstream;
```

Constructors

basic_ofstream();

©Copyright 1999 Rogue Wave Software, Inc.

Constructs an object of class ***basic_ofstream<charT,traits>***, initializing the base class ***basic_ostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. After construction a file can be attached to the ***basic_ofstream*** object using the `open` member function.

```
basic_ofstream(const char* s,
               ios_base::openmode mode= ios_base::in,
               long protection= 0666);
```

Constructs an object of class ***basic_ofstream<charT,traits>***, initializing the base class ***basic_ostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the open function `open(s,mode,protection)` in order to attach the file whose name is pointed to by `s`, to the ***basic_ofstream*** object. The third argument, `protection`, is used as the file permissions. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX. It is more limited under DOS since files are always readable and do not have special execute permission.

```
explicit basic_ofstream(int fd);
```

Constructs an object of class ***basic_ofstream<charT,traits>***, initializing the base class ***basic_ostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_ofstream*** object. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. If the function fails, it sets `ios_base::failbit`.

```
basic_ofstream(int fd, char_type* buf,int len);
```

Constructs an object of class ***basic_ofstream<charT,traits>***, initializing the base class ***basic_ostream*** with the associated file buffer, which is initialized by calling the `basic_filebuf` constructor `basic_filebuf<charT,traits>()`. The constructor then calls the `basic_filebuf` open function `open(fd)` in order to attach the file descriptor `fd` to the ***basic_ofstream*** object. The underlying buffer is then replaced by calling the `basic_filebuf` member function `setbuf` with parameters `buf` and `len`. This constructor is not described in the C++ standard, and is included as an extension in order to manipulate pipes, sockets, or other UNIX devices that can be accessed through file descriptors. It also maintains compatibility with the old iostreams library. If the function fails, it sets `ios_base::failbit`.

Destructors

```
virtual ~basic_ofstream();
```

Destroys an object of class `basic_ofstream`.

Member Functions

```
void
close();
```

Calls the associated `basic_filebuf` function `close()` and if this function fails, it calls the `basic_ios` member function `setstate(failbit)`.

```
bool
is_open();
```

Calls the associated `basic_filebuf` function `is_open()` and returns its result.

```
void
open(const char* s,ios_base::openmode =
          ios_base::out, long protection = 0666);
```

Calls the associated `basic_filebuf` function `open(s,mode,protection)` and, if this function fails opening the file, calls the `basic_ios` member function `setstate(failbit)`. The third argument, `protection`, is used as the file permissions. It does not appear in the Standard C++ description and is included as an extension. It determines the file read/write/execute permissions under UNIX, and is more limited under DOS since files are always readable and do not have special execute permission.

```
basic_filebuf<charT,traits>*
rdbuf() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a pointer to the `basic_filebuf` associated with the stream.

Example

See ***basic_fstream***, ***basic_ifstream*** and ***basic_filebuf*** examples.

See Also

char_traits(3C++),***ios_base***(3C++), ***basic_ios***(3C++), ***basic_filebuf***(3C++), ***basic_ifstream***(3C++), ***basic_fstream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.8.1.8

Standards Conformance

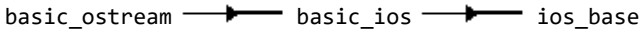
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_ostream



Summary

Assists in formatting and writing output to sequences controlled by a stream buffer.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
ios_type
off_type
ostream
ostream_type
pos_type
traits_type
wostream

Member Functions

bool()
endl()
ends()
flush()
operator<<()
put()
seekp()
sentry()
tellp()
write()
~sentry()

Synopsis

```
#include <ostream>
template<class charT, class traits = char_traits<charT> >
class basic_ostream
: virtual public basic_ios<charT, traits>
```

Description

The class ***basic_ostream*** defines member functions that assist in formatting and writing output to sequences controlled by a stream buffer.

©Copyright 1999 Rogue Wave Software, Inc.

Two groups of member functions share common properties: the formatted output functions (or insertors) and the unformatted output functions. Both groups of functions insert characters by calling ***basic_streambuf*** member functions. They both begin by constructing an object of class ***basic_ostream::sentry*** and, if this object is in good state after construction, the function tries to perform the requested output. The `sentry` object performs exception-safe initialization, such as controlling the status of the stream or locking it in a multithread environment.

Some formatted output functions generate the requested output by converting a value from some scalar to text form and inserting the converted text in the output sequence. The conversion behavior is directly depend on the locale object being imbued in the stream.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_ostream
:virtual public basic_ios<charT, traits> {

 public:

  typedef traits                        traits_type;
  typedef charT                         char_type;
  typedef typename traits::int_type     int_type;
  typedef typename traits::pos_type     pos_type;
  typedef typename traits::off_type     off_type;

  typedef basic_ostream<charT, traits>  ostream_type;
  typedef basic_ios<charT, traits>      ios_type;

  explicit basic_ostream(basic_streambuf<charT, traits>
                       *sb);
  virtual ~basic_ostream();

 class sentry {

  public:

     explicit sentry(basic_ostream<charT,traits>&);
     ~sentry();
     operator bool ();

  };

  ostream_type& operator<<(ostream_type&
                         (*pf)(ostream_type&));
  ostream_type& operator<<(ios_base& (*pf)(ios_base&));
  ostream_type& operator<<(ios_type& (*pf)(ios_type&));

  ostream_type& operator<<(bool n);
  ostream_type& operator<<(short n);
  ostream_type& operator<<(unsigned short n);
  ostream_type& operator<<(int n);
  ostream_type& operator<<(unsigned int n);
  ostream_type& operator<<(long n);
  ostream_type& operator<<(unsigned long n);
  ostream_type& operator<<(float f);
  ostream_type& operator<<(double f);
  ostream_type& operator<<(long double f);

  ostream_type& operator<<(const void *p);

  ostream_type&
          operator<<(basic_streambuf<char_type,traits> &sb);
  ostream_type&
          operator<<(basic_streambuf<char_type,traits> *sb);

  ostream_type& put(char_type c);
  ostream_type& write(const char_type *s, streamsize n);

  ostream_type& flush();

  pos_type tellp();
  ostream_type& seekp(pos_type );
  ostream_type& seekp(off_type , ios_base::seekdir );

 protected:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  basic_ostream();

};


//global character inserter functions

template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>&, charT);

template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>&, char);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&, char);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&, signed char);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&, unsigned char);

template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>&, const charT*);

template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>&, const char*);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&, const char*);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&, const signed char*);

template <class traits>
basic_ostream<char,traits>&
operator<<(basic_ostream<char,traits>&,
           const unsigned char*);

template<class charT, class traits>
basic_ostream<charT, traits>&
endl(basic_ostream<charT, traits>& os);

template<class charT, class traits>
basic_ostream<charT, traits>&
ends(basic_ostream<charT, traits>& os);

template<class charT, class traits>
basic_ostream<charT, traits>&
flush(basic_ostream<charT, traits>& os);
```

Types

`char_type`

The type `char_type` is a synonym for the template parameter `charT`.

`int_type`

The type `int_type` is a synonym of type `traits::in_type`.

`ios_type`

The type `ios_type` is a synonym for `basic_ios<charT, traits>`.

©Copyright 1999 Rogue Wave Software, Inc.

off_type

The type off_type is a synonym of type traits::off_type.

ostream

The type ostream is an instantiation of class basic_ostream on type char:

```
typedef basic_ostream<char> ostream;
```

ostream_type

The type ostream_type is a synonym for basic_ostream<charT, traits>.

pos_type

The type pos_type is a synonym of type traits::pos_type.

traits_type

The type traits_type is a synonym for the template parameter traits.

wostream

The type wostream is an instantiation of class basic_ostream on type wchar_t:

```
typedef basic_ostream<wchar_t> wostream;
```

Constructors

```
explicit basic_ostream(basic_streambuf<charT, traits>* sb);
```

Constructs an object of class basic_ostream, assigning initial values to the base class by calling basic_ios<charT,traits>::init(sb).

Destructors

```
virtual ~basic_ostream();
```

Destroys an object of class basic_ostream.

Sentry Classes

```
explicit sentry(basic_ostream<charT,traits>&);
```

Prepares for formatted or unformatted output. If the basic_ios member function tie() is not a null pointer, the function synchronizes the output sequence with any associated stream. If after any preparation is completed, the basic_ios member function good() is true, the sentry conversion function operator bool() returns true. Otherwise it returns false. In a multithread environment the sentry object constructor is responsible for locking the stream and the stream buffer associated with the stream.

```
~sentry();
```

Destroys an object of class sentry. If the ios_base member function flags() & unitbuf == true, then flushes the buffer. In a multithread environment the sentry object destructor is responsible for unlocking the stream and the stream buffer associated with the stream.

```
operator bool();
```

If after any preparation is completed, the ios_base member function good() is true, the sentry conversion function operator bool() returns true, else it returns false.

Insertors

```
ostream_type&
operator<<(ostream_type& (*pf) (ostream_type&));
```

©Copyright 1999 Rogue Wave Software, Inc.

Calls `pf(*this)`, then returns `*this`. See, for example, the function signature `endl(basic_ostream&)`.

```
ostream_type&
operator<<(ios_type& (*pf) (ios_type&));
```

Calls `pf(*this)`, then returns `*this`.

```
ostream_type&
operator<<(ios_base& (*pf) (ios_base&));
```

Calls `pf(*this)`, then returns `*this`. See, for example, the function signature `dec(ios_base&)`.

```
ostream_type&
operator<<(bool n);
```

Converts the boolean value `n` and outputs it into the `basic_ostream` object's buffer. If the `ios_base` member function `flag() & ios_base::boolalpha` is `false` it tries to write an integer value, which must be `0` or `1`. If the `boolalpha` flag is `true`, it writes characters according to the locale function `numpunct<>::truename()` or `numpunct<>::falsename()`.

```
ostream_type&
operator<<(short n);
```

Converts the `signed short` integer `n`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(unsigned short n);
```

Converts the `unsigned short` integer `n`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(int n);
```

Converts the `signed integer n`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(unsigned int n);
```

Converts the `unsigned integer n`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(long n);
```

Converts the `signed long` integer `n`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(unsigned long n);
```

Converts the unsigned `long` integer n, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(float f);
```

Converts the `float f` and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(double f);
```

Converts the `double f` and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(long double f);
```

Converts the `long double f` and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(void *p);
```

Converts the `pointer p`, and outputs it into the stream buffer, then returns `*this`.

```
ostream_type&
operator<<(basic_streambuf<charT,traits> *sb);
```

©Copyright 1999 Rogue Wave Software, Inc.

If `sb` is null calls the `basic_ios` member function `setstate(badbit)`. Otherwise gets characters from `sb` and inserts them into the stream buffer until any of the following occurs:

- An end-of-file on the input sequence.
- A failure when inserting in the output sequence
- An exception while getting a character from `sb`

If the function inserts no characters or if it stopped because an exception was thrown while extracting a character, it calls the `basic_ios` member function `setstate(failbit)`. If an exception was thrown while extracting a character, it is rethrown.

```
ostream_type&
operator<<(basic_streambuf<charT,traits>& sb);
```

Gets characters from `sb` and inserts them into the stream buffer until any of the following occurs:

- An end-of-file on the input sequence.
- A failure when inserting in the output sequence
- An exception while getting a character from `sb`

If the function inserts no characters or if it stopped because an exception was thrown while extracting a character, it calls the `basic_ios` member function `setstate(failbit)`. If an exception was thrown while extracting a character it is rethrown.

Unformatted Functions

```
ostream_type&
flush();
```

If `rdbuf()` is not a null pointer, calls `rdbuf()->pubsync()` and returns `*this`. If that function returns `-1`, calls `setstate(badbit)`.

```
ostream_type&
put(char_type c);
```

Inserts the character `c`. If the operation fails, calls the `basic_ios` member function `setstate(badbit)`.

```
ostream_type&
seekp(pos_type pos);
```

If the `basic_ios` member function `fail()` returns `false`, executes `rdbuf()->pubseekpos(pos)`, which positions the current pointer of the output sequence at the position designated by `pos`.

```
ostream_type&
seekp(off_type off, ios_base::seekdir dir);
```

If the `basic_ios` member function `fail()` returns `false`, executes `rdbuf()->pubseekpos(off,dir)`, which positions the current pointer of the output sequence at the position designated by `off` and `dir`.

```
pos_type
tellp();
```

If the `basic_ios` member function `fail()` returns `true`, `tellp()` returns `pos_type(off_type(-1))` to indicate failure. Otherwise it returns the current position of the output sequence by calling `rdbuf()-> pubseekoff(0,cur, out)`.

```
ostream_type&
write(const char_type* s, streamsize n);
```

Obtains characters to insert from successive locations of an array whose first element is designated by `s`. Characters are inserted until either of the following occurs:

- `n` characters are inserted
- Inserting in the output sequence fails

©Copyright 1999 Rogue Wave Software, Inc.

In the second case the function calls the `basic_ios` member function `setstate(badbit)`. The function returns `*this`.

Non-member Functions

```
template<class charT, class traits>
basic_ostream<charT, traits>&
endl(basic_ostream<charT, traits>& os);
```

Outputs a `newline` character and flushes the buffer, then returns `os`.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
ends(basic_ostream<charT, traits>& os);
```

Inserts a null character into the output sequence, then returns `os`.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
flush(basic_ostream<charT, traits>& os);
```

Flushes the buffer, then returns `os`.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, charT c);
```

Outputs the character `c` of type `charT` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `charT`. Padding is not ignored.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, char c);
```

Outputs the character `c` of type `char` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `charT`. Conversion from characters of type `char` to characters of type `charT` is performed by the `basic_ios` member function `widen`. Padding is not ignored.

```
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& os, char c);
```

Outputs the character `c` of type `char` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `char`. Padding is not ignored.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const charT* s);
```

Outputs the null-terminated-byte-string `s` of type `charT*` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `charT`.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const char* s);
```

Outputs the null-terminated-byte-string `s` of type `char*` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `charT`. Conversion from characters of type `char` to characters of type `charT` is performed by the `basic_ios` member function `widen`.

```
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& os, const char* s);
```

Outputs the null-terminated-byte-string `s` of type `char*` into the `basic_ostream` object's buffer. Both the stream and the stream buffer are instantiated on type `char`.

```
template<class traits>
basic_ostream<char, traits>&
```

©Copyright 1999 Rogue Wave Software, Inc.

```
operator<<(basic_ostream<char, traits>& os,
           unsigned char c);
```

Returns `os << (char)c`.

```
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& os, signed char c);
```

Returns `os << (char)c`.

```
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& os,
           unsigned char* c);
```

Returns `os << (char*)c`.

```
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& os,
           signed char* c);
```

Returns `os << (char*)c`.

Formatting

The formatting is done through member functions or manipulators.

Manipulators	Member Functions
showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::showpos)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::showbase)
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::uppercase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::showpoint)
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unsetf(ios_base::boolalpha)
unitbuf	setf(ios_base::unitbuf)
nounitbuf	unsetf(ios_base::unitbuf)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)
resetiosflags (ios_base::fmtflags flag)	setf(0,flag)
setiosflags	setf(flag)

©Copyright 1999 Rogue Wave Software, Inc.

`(ios_base::fmtflags flag)`	
`setbase(int base)`	see above
`setfill(char_type c)`	`fill(c)`
`setprecision(int n)`	`precision(n)`
`setw(int n)`	`width(n)`

Description

`showpos`	Generates a + sign in non-negative generated numeric output.
`showbase`	Generates a prefix indicating the numeric base of generated integer output
`uppercase`	Replaces certain lowercase letters with their uppercase equivalents in generated output
`showpoint`	Generates a decimal-point character unconditionally in generated floating-point output
`boolalpha`	Inserts and extracts bool type in alphabetic format
`unitbuf`	Flushes output after each output operation
`internal`	Adds fill characters at a designated internal point in certain generated output. If no such point is designated, it's identical to `right`.
`left`	Adds fill characters on the right (final positions) of certain generated output
`right`	Adds fill characters on the left (initial positions) of certain generated output
`dec`	Converts integer input or generates integer output in decimal base
`hex`	Converts integer input or generates integer output in hexadecimal base
`oct`	Converts integer input or generates integer output in octal base
`fixed`	Generates floating-point output in fixed-point notation
`scientific`	Generates floating-point output in scientific notation
`resetiosflags (ios_base::fmtflags flag)`	Resets the `fmtflags` field `flag`
`setiosflags (ios_base::fmtflags flag)`	Sets up the flag `flag`
`setbase(int base)`	Converts integer input or generates integer output in base `base`. The parameter `base` can be 8, 10 or 16.
`setfill(char_type c)`	Sets the character used to pad (fill) an output conversion to the specified field width
`setprecision(int n)`	Sets the precision (number of digits after the decimal point) to generate on certain output conversions
`setw(int n)`	Sets the field with (number of characters) to generate on certain output conversions

Example

```
//
// stdlib/examples/manual/ostream1.cpp
//
#include<iostream>
#include<ostream>
#include<sstream>
#include<iomanip>

void main ( )
{
   using namespace std;

   float f= 3.14159;
   int   i= 22;
   const char* s= "Randy is the king of stdlib";

   // create a read/write stringbuf object on tiny char
   // and attach it to an istringstream object
   istringstream in( ios_base::in | ios_base::out );

   // tie the ostream object to the istringstream object
   ostream out(in.rdbuf());
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   out << "test beginning !" << endl;

   // output i in hexadecimal
   out << hex << i <<endl;

   // set the field width to 10
   // set the padding character to '@'
   // and output i in octal
   out << setw(10) << oct << setfill('@') << i << endl;

   // set the precision to 2 digits after the separator
   // output f
   out << setprecision(3) << f << endl;

   // output the 17 first characters of s
   out.write(s,17);

   // output a newline character
   out.put('\n');

   // output s
   out << s << endl;

   // output the all buffer to standard output
   cout << in.rdbuf();
}


//
// stdlib/examples/manual/ostream2.cpp
//
#include<iostream>
#include<ostream>
#include<sstream>

void main ( )
{
   using namespace std;

   float f= 3.14159;
   const wchar_t* s= L"Kenavo !";

   // create a read/write stringbuf object on wide char
   // and attach it to an wistringstream object
   wistringstream in( ios_base::in | ios_base::out );

   // tie the wostream object to the wistringstream object
   wostream out(in.rdbuf());

   out << L"test beginning !" << endl;

   // output f in scientific format
   out << scientific << f <<endl;

   // store the current put-pointer position
   wostream::pos_type pos = out.tellp();

   // output s
   out << s << endl;

   // output the all buffer to standard output
   wcout << in.rdbuf() << endl;

   // position the get-pointer
   in.seekg(pos);

   // output s
   wcout << in.rdbuf() << endl;
}
```

See Also

char_traits(3C++), *ios_base*(3C++), *basic_ios*(3C++), *basic_streambuf*(3C++), *basic_iostream*(3C++)

©Copyright 1999 Rogue Wave Software, Inc.

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.6.2.1

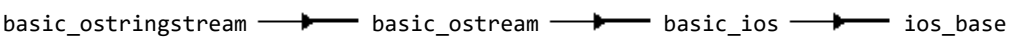
Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_ostringstream



Summary

Supports writing objects of class ***basic_string<charT,traits,Allocator>***

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	ostringstream	
int_type	pos_type	traits_type
ios_type	sb_type	wostringstream
off_type	string_type	

Member Functions

rdbuf()
str()

Synopsis

```
#include <sstream>
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<void> >
class basic_ostringstream
: public basic_ostream<charT, traits>
```

Description

The template class ***basic_ostringstream<charT,traits,Allocator>*** writes to an array in memory. It supports writing objects of class ***basic_string<charT,traits,Allocator>***. It uses a ***basic_stringbuf*** object to control the associated storage. It inherits from ***basic_ostream*** and therefore can use all the formatted and unformatted output functions.

Interface

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_ostringstream
: public basic_ostream<charT, traits> {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 public:

  typedef traits                          traits_type;
  typedef charT                           char_type;
  typedef typename traits::int_type       int_type;
  typedef typename traits::pos_type       pos_type;
  typedef typename traits::off_type       off_type;

  typedef basic_stringbuf<charT, traits, Allocator> sb_type;
  typedef basic_ios<charT, traits> ios_type;
  typedef basic_string<charT, traits, Allocator>
                       string_type;

  explicit basic_ostringstream(ios_base::openmode which =
                               ios_base::out);

  explicit basic_ostringstream(const string_type& str,
                               ios_base::openmode which =
                               ios_base::out);

  virtual ~basic_ostringstream();

  basic_stringbuf<charT,traits,Allocator> *rdbuf() const;

  string_type str() const;

  void str(const string_type& str);

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

ostringstream

The type `ostringstream` is an instantiation of class `basic_ostringstream` on type `char`:

```
typedef basic_ostringstream<char> ostringstream;
```

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

sb_type

The type `sb_type` is an instantiation of class `basic_stringbuf` on type `charT`.

string_type

The type `string_type` is an instantiation of class `basic_string` on type `charT`.

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wostringstream

©Copyright 1999 Rogue Wave Software, Inc.

The type `wostringstream` is an instantiation of class `basic_ostringstream` on type `wchar_t`:

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

Constructors

```
explicit basic_ostringstream(ios_base::openmode which =
                             ios_base::out);
```

Constructs an object of class `basic_ostringstream`, initializing the base class `basic_ostream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(which)`.

```
explicit basic_ostringstream(const string_type& str,
                             ios_base::openmode which =
                             ios_base::out);
```

Constructs an object of class `basic_ostringstream`, initializing the base class `basic_ostream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(str,which)`.

Destructors

```
virtual ~basic_ostringstream();
```

Destroys an object of class `basic_ostringstream`.

Member Functions

```
basic_stringbuf<charT,traits,Allocator>*
rdbuf() const;
```

Returns a pointer to the `basic_stringbuf` associated with the stream.

```
string_type
str() const;
```

Returns a string object of type `string_type` whose contents is a copy of the underlying buffer contents.

```
void
str(const string_type& str);
```

Clears the underlying string buffer and copies the string object `str` into it. If the opening mode is `in`, initializes the input sequence to point to the first character of the buffer. If the opening mode is `out`, initializes the output sequence to point to the first character of the buffer. If the opening mode is `out | app`, initializes the output sequence to point to the last character of the buffer.

Example

See *basic_stringstream*, *basic_istringstream* and *basic_stringbuf* examples.

See Also

char_traits(3C++), *ios_base*(3C++), *basic_ios*(3C++), *basic_stringbuf*(3C++), *basic_string*(3C++), *basic_istringstream*(3C++), *basic_stringstream*(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.7.3

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

basic_streambuf

Summary

Abstract base class for deriving various stream buffers to facilitate control of character sequences.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	streambuf
int_type	traits_type
off_type	wstreambuf
pos_type	

Member Functions

eback()	pbase()	seekpos()	
egptr()	pbump()	setbuf()	sputn()
epptr()	pptr()	setg()	sungetc()
gbump()	pubimbue()	setp()	sync()
getloc()	pubseekoff()	sgetc()	uflow()
gptr()	pubseekpos()	sgetn()	underflow()
imbue()	pubsetbuf()	showmanyc()	which_open_mode()
in_avail()	pubsync()	snextc()	xsgetn()
overflow()	sbumpc()	sputbackc()	xsputn()
pbackfail()	seekoff()	sputc()	

Synopsis

```
#include <streambuf>
template<class charT, class traits = char_traits<charT> >
class basic_streambuf;
```

Description

The class template ***basic_streambuf<charT,traits>*** serves as an abstract base class for deriving various stream buffers to facilitate control of character sequences such as:

- A character input sequence;
- A character output sequence.

©Copyright 1999 Rogue Wave Software, Inc.

Each sequence is associated with three pointers (as described below), which, if non-null, all point into the same `charT` array object. The array object represents, at any moment, a segment of characters from the sequence. Operations performed on a sequence alter the values pointed to by these pointers, perform reads and writes directly to or from associated sequences, and alter "the stream position" and conversion state as needed to maintain this segment to sequence relationship. The three pointers are:

- The beginning pointer, or lowest element address in the array;
- The next pointer, or next element address that is a current candidate for reading or writing;
- The end pointer, or first element address beyond the end of the array.

Stream buffers can impose various constraints on the sequences they control, including:

- The controlled input sequence may be unreadable;
- The controlled output sequence may be unwriteable;
- The controlled sequences can be associated with the contents of other representations for character sequences, such as external files;
- The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

Interface

```
template<class charT, class traits = char_traits<charT> >
class basic_streambuf {

public:

 typedef charT                       char_type;
 typedef traits                      traits_type;

 typedef typename traits::int_type    int_type;
 typedef typename traits::pos_type    pos_type;
 typedef typename traits::off_type    off_type;


 virtual ~basic_streambuf();

 locale pubimbue( const locale& loc);
 locale getloc() const;

 basic_streambuf<char_type, traits> *
   pubsetbuf(char_type *s, streamsize n);

 pos_type pubseekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which =
                     ios_base::in | ios_base::out);

 pos_type pubseekpos(pos_type sp, ios_base::openmode which =
                     ios_base::in | ios_base::out);

 int pubsync();

 ios_base::openmode which_open_mode();

 streamsize   in_avail();
 int_type snextc();
 int_type sbumpc();
 int_type sgetc();
 streamsize sgetn(char_type *s, streamsize n);

 int_type sputbackc(char_type c);
 int sungetc();

 int_type sputc(char_type c);
 streamsize sputn(const char_type *s, streamsize n);

protected:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 basic_streambuf();

 char_type *eback() const;
 char_type *gptr()  const;
 char_type *egptr() const;

 void gbump(int n);
 void setg(char_type *gbeg_arg,char_type *gnext_arg,
           char_type *gend_arg);

 char_type *pbase() const;
 char_type *pptr() const;
 char_type *epptr() const;
 void pbump(int n);
 void setp(char_type *pbeg_arg,char_type *pend_arg);

 virtual void imbue( const locale& loc);

 virtual basic_streambuf<charT, traits>*
   setbuf(char_type *s, streamsize n);

 virtual pos_type seekoff(off_type off,
                          ios_base::seekdir way,
                          ios_base::openmode which =
                          ios_base::in | ios_base::out);

 virtual pos_type seekpos(pos_type sp,
                          ios_base::openmode which =
                          ios_base::in | ios_base::out);

 virtual int showmanyc();
 virtual streamsize xsgetn(char_type *s, streamsize n);

 virtual int_type underflow();
 virtual int_type uflow();

 virtual int_type pbackfail(int_type c = traits::eof());
 virtual streamsize xsputn(const char_type *s,
                           streamsize n);
 virtual int_type overflow(int_type c = traits::eof());

 virtual int sync();

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

int_type

The type `int_type` is a synonym of type `traits::in_type`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

streambuf

The type `streambuf` is an instantiation of class `basic_streambuf` on type `char`:

```
typedef basic_streambuf<char> streambuf;
```

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wstreambuf

©Copyright 1999 Rogue Wave Software, Inc.

The type `wstreambuf` is an instantiation of class `basic_streambuf` on type `wchar_t`:

```
typedef basic_streambuf<wchar_t> wstreambuf;
```

Public Constructors

```
basic_streambuf();
```

Constructs an object of class `basic_streambuf`. Initializes all its pointer member objects to null pointers, and initializes the `getloc()` member function to return the value of `locale::locale()`.

Public Destructors

```
virtual ~basic_streambuf();
```

Destroys an object of class `basic_streambuf`.

Public Member Functions

```
locale
getloc() const;
```

If `pubimbue()` has ever been called, returns the last value of `loc` supplied. Otherwise, it returns the default (global) locale `locale::locale()` in effect at the time of construction.

```
streamsize
in_avail();
```

If a read position is available, returns the number of available characters in the input sequence. Otherwise calls the protected function `showmanyc()`.

```
locale
pubimbue(const locale& loc);
```

Calls the protected function `imbue(loc)`.

```
pos_type
pubseekoff(off_type off, ios_base::seekdir way,
           ios_base::openmode which =
           ios_base::in | ios_base::out );
```

Calls the protected function `seekoff(off,way,which)`.

```
pos_type
pubseekpos(pos_type sp, ios_base::openmode which=
           ios_base::in | ios_base::out );
```

Calls the protected function `seekpos(sp,which)`.

```
basic_streambuf<char_type,traits>*
pubsetbuf(char_type* s,streamsize n);
```

Calls the protected function `setbuf(s,n)` .

```
int
pubsync();
```

Calls the protected function `sync()`.

```
int_type
sbumpc();
```

If the input sequence read position is not available, calls the function `uflow()`. Otherwise it returns `*gptr()` and increments the next pointer for the input sequence.

```
int_type
sgetc();
```

©Copyright 1999 Rogue Wave Software, Inc.

If the input sequence read position is not available, calls the protected function `underflow()`. Otherwise it returns `*gptr()`.

```
streamsize
sgetn(char_type* s, streamsize n);
```

Calls the protected function `xsgetn(s,n)`.

```
int_type
snextc();
```

Calls the function `sbumpc()` and if it returns `traits::eof()`, returns `traits::eof()`. Otherwise it calls the function `sgetc()`.

```
int_type
sputbackc(char_type c);
```

If the input sequence putback position is not available or if `traits::eq(c,gptr() [-1])` returns `false`, calls the protected function `pbackfail(c)`. Otherwise it decrements the next pointer for the input sequence and returns `*gptr()`.

```
int_type
sputc(char_type c);
```

If the output sequence write position is not available, calls the protected function `overflow(traits::to_int_type( c ))`. Otherwise, it stores `c` at the next pointer for the output sequence, increments the pointer, and returns `*pptr()`.

```
streamsize
sputn(const char_type* s, streamsize n);
```

Calls the protected function `xsputn(s,n)`.

```
int_type
sungetc();
```

If the input sequence putback position is not available, calls the protected function `pbackfail()`. Otherwise it decrements the next pointer for the input sequence and returns `*gptr()`.

```
ios_base::openmode
which_open_mode();
```

Returns the mode in which the stream buffer is opened. This function is not described in the C++ standard.

Protected Member Functions

```
char_type*
eback() const;
```

Returns the beginning pointer for the input sequence.

```
char_type*
egptr() const;
```

Returns the end pointer for the input sequence.

```
char_type*
epptr() const;
```

Returns the end pointer for the output sequence.

```
void
gbump(int n);
```

Advances the next pointer for the input sequence by `n`.

```
char_type*
gptr() const;
```

Returns the next pointer for the input sequence.

©Copyright 1999 Rogue Wave Software, Inc.

```
void
imbue(const locale&);
```

Changes any translations based on locale. The default behavior is to do nothing. This function has to be overloaded in the classes derived from `basic_streambuf`. The purpose of this function is to allow the derived class to be informed of changes in locale at the time they occur. The new imbued locale object is only used by the stream buffer; it does not affect the stream itself.

```
int_type
overflow(int_type c = traits::eof() );
```

The member functions `sputc()` and `sputn()` call this function when not enough room can be found in the put buffer to accommodate the argument character sequence. The function returns `traits::eof()` if it fails to make more room available or if it fails to empty the buffer by writing the characters to their output device.

```
int_type
pbackfail(int_type c = traits::eof() );
```

If `c` is equal to `traits::eof()`, `gptr()` is moved back one position. Otherwise `c` is prepended. The function returns `traits::eof()` to indicate failure.

```
char_type*
pbase() const;
```

Returns the beginning pointer for the output sequence.

```
void
pbump(int n);
```

Advances the next pointer for the output sequence by `n`.

```
char_type*
pptr() const;
```

Returns the next pointer for the output sequence.

```
pos_type
seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which =
        ios_base::in | ios_base::out );
```

Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf`. The default behavior is to return an object of type `pos_type` that stores an invalid stream position.

```
pos_type
seekpos(pos_type sp, ios_base::openmode which=
        ios_base::in | ios_base::out );
```

Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf`. The default behavior is to return an object of class `pos_type` that stores an invalid stream position.

```
basic_streambuf*
setbuf(char_type* s, streamsize n);
```

Performs an operation that is defined separately for each class derived from `basic_streambuf`. The purpose of this function is to allow the user to provide his own buffer or to resize the current buffer.

```
void
setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

Sets up a private member for the following to be `true`:

```
   eback() == gbeg, gptr() == gnext and egptr() == gend
```

```
void
setp(char_type* pbeg, char_type* pend);
```

Sets up a private member for the following to be `true`:

©Copyright 1999 Rogue Wave Software, Inc.

pbase() == pbeg, pptr() == pbeg and epptr() == pend

```
int
showmanyc();
```

Returns the number of characters available in the internal buffer, or returns -1.

```
int
sync();
```

Synchronizes the controlled sequences with the internal buffer, in a way that is defined separately for each class derived from basic_streambuf. The default behavior is to do nothing. On failure the return value is -1.

```
int_type
underflow();
```

The public members of basic_streambuf call this function only if gptr() is null or gptr() >= egptr(). This function returns the character pointed to by gptr(), if gptr() is not null and if gptr() < egptr(). Otherwise the function tries to read character into the buffer. If it fails, it returns traits::eof().

```
int_type
uflow();
```

Calls underflow() and if underflow() returns traits::eof(), returns traits::eof(). Otherwise, does gbump(1) and returns the value of *gptr().

```
streamsize
xsgetn(char_type* s, streamsize n);
```

Assigns up to n characters to successive elements of the array whose first element is designated by s. The characters are read from the input sequence. Assigning stops when either n characters have been assigned or a call to sbumpc() would return traits::eof(). The function returns the number of characters read.

```
streamsize
xsputn(const char_type* s, streamsize n);
```

Writes up to n characters to the output sequence. The characters written are obtained from successive elements of the array whose first element is designated by s. Writing stops when either n characters have been written or a call to sputc() would return traits::eof(). The function returns the number of characters written.

See Also

char_traits(3C++), *basic_filebuf*(3C++), *basic_stringbuf*(3C++), *strstreambuf*(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.5.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

basic_string

`Strings Library`

Summary

A templatized class for handling sequences of character-like entities. ***string*** and ***wstring*** are specialized versions of ***basic_string*** for `char`'s and `wchar_t`'s, respectively.

```
typedef basic_string <char> string;
typedef basic_string <wchar_t> wstring;
```

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

append()	erase()	operator!=()	operator[]()
assign()	find()	operator>>()	rbegin()
at()	find_first_not_of()	operator>()	rend()
begin()	find_first_of()	operator>=()	replace()
capacity()	find_last_not_of()	operator<<()	reserve()
compare()	find_last_of()	operator<()	resize()
copy()	getline()	operator<=()	rfind()
c_str()	get_allocator()	operator+()	size()
data()	insert()	operator+=()	substr()
empty()	length()	operator=()	swap()
end()	max_size()	operator==()	

Synopsis

```
#include <string>

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >

class basic_string;
```

Description

©Copyright 1999 Rogue Wave Software, Inc.

basic_string<charT, traits, Allocator> is a homogeneous collection of character-like entities that includes string functions such as compare, append, assign, insert, remove, and replace, along with various searches. ***basic_string*** also functions as an STL sequence container that provides random access iterators. This allows some of the generic algorithms to apply to strings.

Any underlying character-like type may be used as long as an appropriate `char_traits` class is included or the default `traits` class is applicable.

Interface

```
template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
class basic_string {

public:

// Types

typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
typedef typename Allocator::pointer             iterator;
typedef typename Allocator::const_pointer       const_iterator;
typedef std::reverse_iterator<const_iterator>
                                       const_reverse_iterator;
typedef std::reverse_iterator<iterator>         reverse_iterator;

static const size_type npos = -1;

// Constructors/Destructors

explicit basic_string(const Allocator& = Allocator());
basic_string (const basic_string<charT, traits,
              Allocator>&);
basic_string(const basic_string&, size_type,
             size_type = npos,
             const Allocator& a = Allocator());
basic_string(const charT*, size_type,
             const Allocator& = Allocator());
basic_string(const charT*, const Allocator& = Allocator());
basic_string(size_type, charT,
             const Allocator& = Allocator());
template <class InputIterator>
basic_string(InputIterator, InputIterator,
             const Allocator& = Allocator());
~basic_string();

// Assignment operators
 basic_string& operator=(const basic_string&);
 basic_string& operator=(const charT*);
 basic_string& operator=(charT);

// Iterators

 iterator       begin();
 const_iterator begin() const;
 iterator       end();
 const_iterator end() const;

 reverse_iterator       rbegin();
 const_reverse_iterator rbegin() const;
 reverse_iterator       rend();
 const_reverse_iterator rend() const;

// Capacity

   size_type       size() const;
   size_type       length() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   size_type       max_size() const;
   void            resize(size_type, charT);
   void            resize(size_type);
   size_type       capacity() const;
   void            reserve(size_type = 0);
   bool            empty() const;

// Element access

   const_reference operator[](size_type) const;
   reference       operator[](size_type);
   const_reference at(size_type) const;
   reference       at(size_type);

// Modifiers

   basic_string& operator+=(const basic_string&);
   basic_string& operator+=(const charT*);
   basic_string& operator+=(charT);

   basic_string& append(const basic_string&);
   basic_string& append(const basic_string&,
                        size_type, size_type);
   basic_string& append(const charT*, size_type);
   basic_string& append(const charT*);
   basic_string& append(size_type, charT);
   template<class InputIterator>
    basic_string& append(InputIterator, InputIterator);

   basic_string& assign(const basic_string&);
   basic_string& assign(const basic_string&,
                        size_type, size_type);
   basic_string& assign(const charT*, size_type);
   basic_string& assign(const charT*);
   basic_string& assign(size_type, charT);
   template<class InputIterator>
    basic_string& assign(InputIterator, InputIterator);

   basic_string& insert(size_type, const basic_string&);
   basic_string& insert(size_type, const basic_string&,
                        size_type, size_type);
   basic_string& insert(size_type, const charT*, size_type);
   basic_string& insert(size_type, const charT*);
   basic_string& insert(size_type, size_type, charT);
   iterator insert(iterator, charT = charT());
   void insert(iterator, size_type, charT);
   template<class InputIterator>
    void insert(iterator, InputIterator, InputIterator);

   basic_string& erase(size_type = 0, size_type= npos);
   iterator erase(iterator);
   iterator erase(iterator, iterator);

   basic_string& replace(size_type, size_type,
                         const basic_string&);
   basic_string& replace(size_type, size_type,
                         const basic_string&,
                         size_type, size_type);
   basic_string& replace(size_type, size_type,
                         const charT*, size_type);
   basic_string& replace(size_type, size_type,
                         const charT*);
   basic_string& replace(size_type, size_type,
                         size_type, charT);
   basic_string& replace(iterator, iterator,
                         const basic_string&);
   basic_string& replace(iterator, iterator,
                         const charT*, size_type);
   basic_string& replace(iterator, iterator,
                         const charT*);
   basic_string& replace(iterator, iterator,
                         size_type, charT);
   template<class InputIterator>
    basic_string& replace(iterator, iterator,
                         InputIterator, InputIterator);

   size_type copy(charT*, size_type, size_type = 0) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   void swap(basic_string<charT, traits, Allocator>&);

// String operations

   const charT* c_str() const;
   const charT* data() const;
   const allocator_type& get_allocator() const;

   size_type find(const basic_string&,
                  size_type = 0) const;
   size_type find(const charT*,
                  size_type, size_type) const;
   size_type find(const charT*, size_type = 0) const;
   size_type find(charT, size_type = 0) const;
   size_type rfind(const basic_string&,
                   size_type = npos) const;
   size_type rfind(const charT*,
                   size_type, size_type) const;
   size_type rfind(const charT*,
                   size_type = npos) const;
   size_type rfind(charT, size_type = npos) const;

   size_type find_first_of(const basic_string&,
                           size_type = 0) const;
   size_type find_first_of(const charT*,
                           size_type, size_type) const;
   size_type find_first_of(const charT*,
                           size_type = 0) const;
   size_type find_first_of(charT, size_type = 0) const;

   size_type find_last_of(const basic_string&,
                          size_type = npos) const;
   size_type find_last_of(const charT*,
                          size_type, size_type) const;
   size_type find_last_of(const charT*, size_type = npos)
                         const;
   size_type find_last_of(charT, size_type = npos) const;

   size_type find_first_not_of(const basic_string&,
                               size_type = 0) const;
   size_type find_first_not_of(const charT*,
                               size_type, size_type) const;
   size_type find_first_not_of(const charT*, size_type = 0)
                           const;
   size_type find_first_not_of(charT, size_type = 0) const;

   size_type find_last_not_of(const basic_string&,
                              size_type = npos) const;
   size_type find_last_not_of(const charT*,
                              size_type, size_type) const;
   size_type find_last_not_of(const charT*,
                              size_type = npos) const;
   size_type find_last_not_of(charT, size_type = npos)
                             const;

   basic_string substr(size_type = 0, size_type = npos)
                      const;
   int compare(const basic_string&) const;
   int compare(size_type, size_type, const basic_string&)
              const;
   int compare(size_type, size_type, const basic_string&,
               size_type, size_type) const;
   int compare(size_type, size_type, charT*) const;
   int compare(charT*) const;
   int compare(size_type, size_type, const charT*,
               size_type) const;
};

// Non-member Operators

template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&,
                         const basic_string&);
template <class charT, class traits, class Allocator>
 basic_string operator+ (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 basic_string operator+ (charT, const basic_string&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&, const charT*);
template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&, charT);

template <class charT, class traits, class Allocator>
 bool operator== (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator== (const charT*, const basic_string&);
template <class charT, class traits , class Allocator>
 bool operator== (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator< (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator< (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator< (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator!= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator!= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator!= (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator> (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator> (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator> (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator<= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator<= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator<= (const basic_string&, const charT*);


template <class charT, class traits, class Allocator>
 bool operator>= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator>= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator>= (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& a,
          basic_string<charT,traits,Allocator>& b);

template<class charT, class traits, class Allocator>
 basic_istream<charT, traits>& operator>>
                           (istream&, basic_string&);
template <class charT, class traits, class Allocator>
 basic_ostream<charT, traits>& operator<<
                           (ostream&, const basic_string&);
template <class Stream, class charT,
          class traits, class Allocator>
 basic_istream<charT, traits>& getline
                           (Stream&, basic_string&, charT);
```

Constructors

In all cases, the `Allocator` parameter is used for storage management.

```
explicit
basic_string (const Allocator& a = Allocator());
```

The default constructor. Creates a ***basic_string*** with the following effects:

`data()`	a non-null pointer that is copyable and can have `0` added to it
`size()`	0

©Copyright 1999 Rogue Wave Software, Inc.

`capacity()` an unspecified value

```
basic_string (const basic_string<T, traits,
              Allocator>& str);
```

Creates a string that is a copy of `str`.

```
basic_string (const basic_string& str, size_type pos,
              size_type n= npos, const allocator&
              a=allocator());
```

Creates a string of `pos<=size()` and determines length `rlen` of the initial string value as the smaller of `n` and `str.size() - pos`. This has the following effects:

`data()`	points to the first element of an allocated copy of `rlen` elements of the string controlled by `str` beginning at position `pos`
`size()`	rlen
`capacity()`	a value at least as large as `size()`
`get_allocator()`	str.get_allocator()

An `out_of_range` exception is thrown if `pos>str.size()`.

```
basic_string (const charT* s, size_type n,
              const Allocator& a = Allocator());
```

Creates a string that contains the first `n` characters of `s`. `s` must not be a `NULL` pointer. The effects of this constructor are:

`data()`	points to the first element of an allocated copy of the array whose first element is pointed to by `s`
`size()`	n
`capacity()`	a value at least as large as `size()`

A `length_error` exception is thrown if `n == npos.`

```
basic_string (const charT * s,
              const Allocator& a = Allocator());
```

Constructs a string containing all characters in `s` up to, but not including, a `traits::eos()` character. `s` must not be a null pointer. The effects of this constructor are:

`data()`	points to the first element of an allocated copy of the array whose first element is pointed to by `s`
`size()`	traits::length(s)
`capacity()`	a value at least as large as `size()`

```
basic_string (size_type n, charT c,
              const Allocator& a = Allocator());
```

Constructs a string containing `n` repetitions of `c`. A `length_error` exception is thrown if `n == npos`. The effects of this constructor are:

`data()`	points to the first element of an allocated array of `n` elements, each storing the initial value `c`
`size()`	n
`capacity()`	a value at least as large as `size()`

```
template <class InputIterator>
basic_string (InputIterator first, InputIterator last,
              const Allocator& a = Allocator());
```

Creates a ***basic_string*** of length `last - first` filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The effects of this constructor are:

`data()`	points to the first element of an allocated copy of the elements in the range `[first,last)`
`size()`	distance between `first` and `last`
`capacity()`	a value at least as large as `size()`

©Copyright 1999 Rogue Wave Software, Inc.

Destructors

```
~basic_string ();
```

Releases any allocated memory for this ***basic_string***.

Operators

```
basic_string&
operator= (const basic_string& str);
```

Sets the contents of this string to be the same as `str`. The effects of `operator=` are:

`data()`	points to the first element of an allocated copy of the array whose first element is pointed to by `str.size()`
`size()`	str.size()
`capacity()`	a value at least as large as `size()`

```
basic_string&
operator= (const charT * s);
```

Sets the contents of this string to be the same as `s` up to, but not including, the `traits::eos()` character.

```
basic_string&
operator= (charT c);
```

Sets the contents of this string to be equal to the single `charT c`.

```
const_reference
operator[] (size_type pos) const;
reference
operator[] (size_type pos);
```

If `pos < size()`, returns the element at position `pos` in this string. If `pos == size()`, the `const` version returns `charT()`, the behavior of the non-`const` version is undefined. The reference returned by either version is invalidated by any call to `c_str()`, `data()`, or any non-`const` member function for the object.

```
basic_string&
operator+= (const basic_string& s);
basic_string&
operator+= (const charT* s);
basic_string&
operator+= (charT c);
```

Concatenates a string onto the current contents of this string. The second member operator uses `traits::length()` to determine the number of elements from `s` to add. The third member operator adds the single character `c`. All return a reference to this string after completion.

Iterators

```
iterator begin ();
const_iterator begin () const;
```

Returns an iterator initialized to the first element of the string.

```
iterator end ();
const_iterator end () const;
```

Returns an iterator initialized to the position after the last element of the string.

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

Returns an iterator equivalent to `reverse_iterator(end())`.

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Returns an iterator equivalent to `reverse_iterator(begin())`.

©Copyright 1999 Rogue Wave Software, Inc.

Allocator

```
const allocator_type get_allocator () const;
```

Returns a copy of the allocator used by self for storage management.

Member Functions

```
basic_string&
append (const basic_string& s, size_type pos,
        size_type npos);
basic_string&
append (const basic_string& s);
basic_string&
append (const charT* s, size_type n);
basic_string&
append (const charT* s);
basic_string&
append (size_type n, charT c );
template<class InputIterator>
basic_string&
append (InputIterator first, InputIterator last);
```

Append another string to the end of this string. The first two functions append the lesser of `n` and `s.size() - pos` characters of `s`, beginning at position `pos` to this string. The second member throws an `out_of_range` exception if `pos > str.size()`. The third member appends `n` characters of the array pointed to by `s`. The fourth variation appends elements from the array pointed to by `s` up to, but not including, a `charT()` character. The fifth variation appends `n` repetitions of `c`. The final `append` function appends the elements specified in the range `[first, last)`.

All functions throw a `length_error` exception if the resulting lengths exceed `max_size()`. All return a reference to this string after completion.

```
basic_string&
assign (const basic_string& s);
basic_string&
assign (const basic_string& s,
        size_type pos, size_type n);
basic_string&
assign (const charT* s, size_type n);
basic_string&
assign (const charT* s);
basic_string&
assign (size_type n, charT c );
template<class InputIterator>
basic_string&
assign (InputIterator first, InputIterator last);
```

Replace the value of this string with the value of another.

All versions of the function assign values to this string. The first two variations assign the lesser of `n` and `s.size() - pos` characters of `s`, beginning at position `pos`. The second variation throws an `out_of_range` exception if `pos > str.size()`. The third version of the function assigns `n` characters of the array pointed to by `s`. The fourth version assigns elements from the array pointed to by `s` up to, but not including, a `charT()` character. The fifth assigns one or `n` repetitions of `c`. The last variation assigns the members specified by the range `[first, last)`.

All functions throw a `length_error` exception if the resulting lengths exceed `max_size()`. All return a reference to this string after completion.

```
const_reference
at (size_type pos) const;
reference
at (size_type pos);
```

If `pos < size()`, returns the element at position `pos` in this string. Otherwise, an `out_of_range` exception is thrown.

```
size_type
capacity () const;
```

Returns the current storage capacity of the string. This is guaranteed to be at least as large as `size()`.

©Copyright 1999 Rogue Wave Software, Inc.

```
int
compare (const basic_string& str);
```

Returns the result of a lexicographical comparison between elements of this string and elements of `str`. The return value is:

`<0` if size() < str.size()
`0` if size() == str.size()
`>0` if size() > str.size()

```
int
compare (size_type pos1, size_type n1,
         const basic_string& str) const;
int
compare (size_type pos1, size_type n1,
         const basic_string& str,
         size_type pos2, size_type n2) const;
int
compare (charT* s) const;
int
compare (size_type pos, size_type n1, charT* s) const;
int
compare (size_type pos, size_type n1, charT* s,
         size_type n2) const;
```

Returns the result of a lexicographical comparison between elements of this string and a given comparison string. The members return, respectively:

```
basic_string(*this,pos1,n1).compare (str)
basic_string(*this,pos1,n1).compare (basic_string
                                     (str, pos2, n2))
*this.compare (basic_string(s))
basic_string(*this,pos,n1).compare (basic_string
                                    (s, npos))
basic_string(*this,pos,n1).compare (basic_string (s,n2))
```

```
size_type
copy (charT* s, size_type n, size_type pos = 0) const;
```

Replaces elements in memory with copies of elements from this string. An `out_of_range` exception is thrown if `pos > size()`. The lesser of `n` and `size() - pos` elements of this string, starting at position `pos`, are copied into the array pointed to by `s`. No terminating null is appended to `s`.

```
const charT*
c_str () const;
const charT*
data () const;
```

Returns a pointer to the initial element of an array whose first `size()` elements are copies of the elements in this string. A `charT()` element is appended to the end. The elements of the array may not be altered, and the returned pointer is only valid until a non-`const` member function of this string is called. If `size()` is zero, the `data()` function returns a non-`NULL` pointer.

```
bool empty () const;
```

Returns `size() == 0`.

```
basic_string&
erase (size_type pos = 0, size_type n = npos);
iterator
erase (iterator p);
iterator
erase (iterator first, iterator last);
```

This function removes elements from the string, collapsing the remaining elements, as necessary, to remove any space left empty.

The first version of the function removes the smaller of `n` and `size() - pos` starting at position `pos`. An `out_of_range` exception is thrown if `pos > size()`.

©Copyright 1999 Rogue Wave Software, Inc.

For the second version, `p` must be a valid iterator on the string, and the function removes the character referred to by `p`.

For the last version of `erase`, both `first` and `last` must be valid iterators on the string, and the function removes the characters defined by the range `[first, last)`. The destructors for all removed characters are called.

All versions of `erase` return a reference to the string after completion.

```
size_type
find (const basic_string& str, size_type pos = 0) const;
```

Searches for the first occurrence of the substring specified by `str` in this string, starting at position `pos`. If found, it returns the index of the first character of the matching substring. If not found, returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find (const charT* s, size_type pos, size_type n) const;
size_type
find (const charT* s, size_type pos = 0) const;
size_type
find (charT c, size_type pos = 0) const;
```

Searches for the first sequence of characters in this string that match a specified string. The variations of this function return, respectively:

```
find(basic_string(s,n), pos)
find(basic_string(s), pos)
find(basic_string(1, c), pos)
```

```
size_type
find_first_not_of (const basic_string& str,
                   size_type pos = 0) const;
```

Searches for the first element of this string at or after position `pos` that is not equal to any element of `str`. If found, `find_first_not_of` returns the index of the non-matching character. If all of the characters match, the function returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_first_not_of (const charT* s,
                   size_type pos, size_type n) const;
size_type
find_first_not_of (const charT* s,
                   size_type pos = 0) const;
size_type
find_first_not_of (charT c, size_type pos = 0) const;
```

Searches for the first element in this string at or after position `pos` that is not equal to any element of a given set of characters. The members return, respectively:

```
find_first_not_of(basic_string(s,n), pos)
find_first_not_of(basic_string(s), pos)
find_first_not_of(basic_string(1, c), pos)
```

```
size_type
find_first_of(const basic_string& str,
              size_type pos = 0) const;
```

Searches for the first occurrence at or after position `pos` of any element of `str` in this string. If found, the index of this matching character is returned. If not found, `npos` is returned. Equality is defined by `traits::eq()`.

```
size_type
find_first_of(const charT* s, size_type pos,
              size_type n) const;
size_type
find_first_of(const charT* s, size_type pos = 0) const;
size_type
find_first_of (charT c, size_type pos = 0) const;
```

Searches for the first occurrence in this string of any element in a specified string. The `find_first_of` variations return, respectively:

©Copyright 1999 Rogue Wave Software, Inc.

```
find_first_of(basic_string(s,n), pos)
find_first_of(basic_string(s), pos)
find_first_of(basic_string(1, c), pos)

size_type
find_last_not_of(const basic_string& str,
                 size_type pos = npos) const;
```

Searches for the last element of this string at or before position `pos` that is not equal to any element of `str`. If `find_last_not_of` finds a non-matching element, it returns the index of the character. If all the elements match, the function returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_last_not_of(const charT* s,
                 size_type pos, size_type n) const;
size_type
find_last_not_of(const charT* s, size_type pos = npos) const;
size_type
find_last_not_of(charT c, size_type pos = npos) const;
```

Searches for the last element in this string at or before position `pos` that is not equal to any element of a given set of characters. The members return, respectively:

```
find_last_not_of(basic_string(s,n), pos)
find_last_not_of(basic_string(s), pos)
find_last_not_of(basic_string(1, c), pos)

size_type
find_last_of(const basic_string& str,
             size_type pos = npos) const;
```

Searches for the last occurrence of any element of `str` at or before position `pos` in this string. If found, `find_last_of` returns the index of the matching character. If not found, `find_last_of` returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_last_of(const charT* s, size_type pos,
             size_type n) const;
size_type
find_last_of(const charT* s, size_type pos = npos) const;
size_type
find_last_of(charT c, size_type pos = npos) const;
```

Searches for the last occurrence in this string of any element in a specified string. The members return, respectively:

```
find_last_of(basic_string(s,n), pos)
find_last_of(basic_string(s), pos)
find_last_of(basic_string(1, c), pos)

basic_string&
insert(size_type pos1, const basic_string& s);
basic_string&
insert(size_type pos, const basic_string& s,
       size_type pos2 = 0, size_type n = npos);
basic_string&
insert(size_type pos, const charT* s, size_type n);
basic_string&
insert(size_type pos, const charT* s);
basic_string&
insert(size_type pos, size_type n, charT c);
```

Inserts additional elements at position `pos` in this string. All of the variants of this function throw an `out_of_range` exception if `pos > size()`. All variants also throw a `length_error` if the resulting strings exceed `max_size()`. Elements of this string are moved apart as necessary to accommodate the inserted elements. All return a reference to this string after completion.

The second variation of this function inserts the lesser of `n` and `s.size() - pos2` characters of `s`, beginning at position `pos2` in this string. This version throws an `out_of_range` exception if `pos2 > s.size()`.

The third version inserts `n` characters of the array pointed to by `s`.

The fourth inserts elements from the array pointed to by `s` up to, but not including, a `charT()` character.

©Copyright 1999 Rogue Wave Software, Inc.

Finally, the fifth variation inserts n repetitions of c.

```
iterator
insert(iterator p, charT);
void
insert(iterator p, size_type n, charT c);
template<class InputIterator>
void
insert(iterator p, InputIterator first, InputIterator last);
```

Inserts additional elements in this string immediately before the character referred to by p. All of these versions of insert require that p is a valid iterator on this string. The first version inserts a copy of c. The second version inserts n repetitions of c. The third version inserts characters in the range [first, last). The first version returns p.

```
size_type
length() const;
```

Returns the number of elements contained in this string.

```
size_type
max_size() const;
```

Returns the maximum possible size of the string.

```
size_type
rfind (const basic_string& str, size_type pos = npos) const;
```

Searches for the last occurrence of the substring specified by str in the string, where the index of the first character of the substring is less than pos. If found, the index of the first character that matches substring is returned. If not found, npos is returned. Equality is defined by traits::eq().

```
size_type
rfind(const charT* s, size_type pos, size_type n) const;
size_type
rfind(const charT* s, size_type pos = npos) const;
size_type
rfind(charT c, size_type pos = npos) const;
```

Searches for the last sequence of characters in this string matching a specified string. The rfind variations return, respectively:

```
rfind(basic_string(s,n), pos)
rfind(basic_string(s), pos)
rfind(basic_string(1, c), pos)
```

```
basic_string&
replace(size_type pos, size_type n1, const basic_string& s);
basic_string&
replace(size_type pos1, size_type n1,
        const basic_string& str,
        size_type pos2, size_type n2);
basic_string&
replace(size_type pos, size_type n1, const charT* s,
        size_type n2);
basic_string&
replace(size_type pos, size_type n1, const charT* s);
basic_string&
replace(size_type pos, size_type n1, size_type n2, charT c);
```

The replace function replaces selected elements of this string with an alternate set of elements. All of these versions insert the new elements in place of n1 elements in this string, starting at position pos. They each throw an out_of_range exception if pos1 > size() and a length_error exception if the resulting string size exceeds max_size().

The second version replaces elements of the original string with n2 characters from string s starting at position pos2. It throws the out_of_range exception if pos2 > s.size(). The third variation of the function replaces elements in the original string with n2 elements from the array pointed to by s. The fourth version replaces elements in the string with elements from the array pointed to by s, up to, but not including, a charT() character. The fifth replaces n elements with n2 repetitions of character c.

©Copyright 1999 Rogue Wave Software, Inc.

```
basic_string&
replace(iterator i1, iterator i2,
        const basic_string& str);
basic_string&
replace(iterator i1, iterator i2, const charT* s,
        size_type n);
basic_string&
replace(iterator i1, iterator i2, const charT* s);
basic_string&
replace(iterator i1, iterator i2, size_type n,
        charT c);
template<class InputIterator>
basic_string&
replace(iterator i1, iterator i2,
        InputIterator j1, InputIterator j2);
```

Replaces selected elements of this string with an alternative set of elements. All of these versions of `replace` require iterators `i1` and `i2` to be valid iterators on this string. The elements specified by the range `[i1, i2)` are replaced by the new elements.

The first version shown here replaces all members in `str`.

The second version starts at position `i1`, and replaces the next `n` characters with `n` characters of the array pointed to by `s`.

The third variation replaces string elements with elements from the array pointed to by `s` up to, but not including, a `charT() character.`

The fourth version replaces string elements with `n` repetitions of `c`.

The last variation shown here replaces string elements with the members specified in the range `[j1, j2)`.

```
void
reserve(size_type res_arg=0);
```

Assures that the storage capacity is at least `res_arg`. Throws a `length_error` exception if res_arg `> max_size()`.

```
void
resize(size_type n, charT c);
void
resize(size_type n);
```

Changes the capacity of this string to `n`. If the new capacity is smaller than the current size of the string, then the string is truncated. If the capacity is larger, then the string is padded with `c` characters. The latter `resize` member pads the string with default characters specified by charT`()`. Throws a `length_error` exception if `n > max_size()`.

```
size type
size() const;
```

Return the number of elements contained in this string.

```
basic_string
substr(size_type pos = 0, size_type n = npos) const;
```

Returns a string composed of copies of the lesser of `n` and `size()` characters in this string starting at index `pos`. Throws an `out_of_range` exception if `pos > size().`

```
void
swap(basic_string& s);
```

Swaps the contents of this string with the contents of `s`.

Non-member Operators

```
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string& lhs, const basic_string& rhs);
```

Returns a string of length `lhs.size() + rhs.size()`, where the first `lhs.size()` elements are copies of the elements of `lhs`, and the next `rhs.size()` elements are copies of the elements of `rhs`.

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class charT, class traits, class Allocator>
basic_string
operator+(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(charT lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string& lhs, const charT* rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string& lhs, charT rhs);
```

Returns a string that represents the concatenation of two string-like entities. These functions return, respectively:

```
basic_string(lhs) + rhs
basic_string(1, lhs) + rhs
lhs + basic_string(rhs)
lhs + basic_string(1, rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator==(const basic_string& lhs,
           const basic_string& rhs);
```

Returns a boolean value of `true` if `lhs` and `rhs` are equal, and `false` if they are not. Equality is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator==(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator==(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value indicating whether `lhs` and `rhs` are equal. Equality is defined by the `compare()` member function. These functions return, respectively:

```
basic_string(lhs) == rhs
lhs == basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator!=(const basic_string& lhs,
           const basic_string& rhs);
```

Returns a boolean value representing the inequality of `lhs` and `rhs`. Inequality is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator!=(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator!=(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the inequality of `lhs` and `rhs`. Inequality is defined by the `compare()` member function. The functions return, respectively:

```
basic_string(lhs) != rhs
lhs != basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator<(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value representing the lexicographical less-than relationship of `lhs` and `rhs`. Less-than is defined by the `compare()` member.

```
template<class charT, class traits, class Allocator>
bool
operator<(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
bool
operator<(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical less-than relationship of `lhs` and `rhs`. Less-than is defined by the `compare()` member function. These functions return, respectively:

```
basic_string(lhs) < rhs
lhs < basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator>(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value representing the lexicographical greater-than relationship of `lhs` and `rhs`. Greater-than is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator>(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator>(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical greater-than relationship of `lhs` and `rhs`. Greater-than is defined by the `compare()` member. The functions return, respectively:

```
basic_string(lhs) > rhs
lhs > basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator<=(const basic_string& lhs,
             const basic_string& rhs);
```

Returns a boolean value representing the lexicographical less-than-or-equal relationship of `lhs` and `rhs`. Less-than-or-equal is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator<=(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator<=(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical less-than-or-equal relationship of `lhs` and `rhs`. Less-than-or-equal is defined by the `compare()` member function. These functions return, respectively:

```
basic_string(lhs) <= rhs
lhs <= basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator>=(const basic_string& lhs,
           const basic_string& rhs);
```

Returns a boolean value representing the lexicographical greater-than-or-equal relationship of `lhs` and `rhs`. Greater-than-or-equal is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator>=(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator>=(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical greater-than-or-equal relationship of `lhs` and `rhs`. Greater-than-or-equal is defined by the `compare()` member. The functions return, respectively:

```
basic_string(lhs) >= rhs
lhs >= basic_string(rhs)
```

```
template <class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& a,
          basic_string<charT,traits,Allocator>& b);
```

©Copyright 1999 Rogue Wave Software, Inc.

Swaps the contents of `a` and `b` by calling `a`'s swap function on `b`.

```
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is,
           basic_string& str);
```

Reads `str` from `is` using `traits::char_in` until a `traits::is_del()` element is read. All elements read, except the delimiter, are placed in `str`. After the read, the function returns `is`.

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream <charT, traits>& os,
           const<charT, traits allocator> str);
```

Writes all elements of `str` to `os` in order from first to last, using `traits::char_out()`. After the write, the function returns `os`.

Non-member Functions

```
template <class Stream, class charT, class traits,
          class Allocator>
basic_istream<charT, traits>
getline(basic_istream<charT, traits> is,
        <charT, traits allocator> str, charT delim);
```

An unformatted input function that extracts characters from `is` into `str` until `npos - 1` characters are read, the end of the input sequence is reached, or the character read is `delim`. The characters are read using `traits::char_in()`.

Example

```
//
// string.cpp
//
 #include<string>
 #include <iostream>
 using namespace std;

 int main()
 {
   string test;

   //Type in a string over five characters long
   while(test.empty() || test.size() <= 5)
   {
     cout << "Type a string between 5 and 100
              characters long. " << endl;
     cin >> test;
   }

   //Test operator[] access
   cout << "Changing the third character from "
        << test[2] << " to * " << endl;
   test[2] = '*';
   cout << "now its: " << test << endl << endl;

   //Try the insertion member function
   cout << "Identifying the middle: ";
   test.insert(test.size() / 2, "(the middle is here!)");
   cout << test << endl << endl;


   //Try replacement
   cout << "I didn't like the word 'middle',so "
           "instead, I'll say:" << endl;

   test.replace(test.find("middle",0), 6, "center");
   cout << test << endl;

   return 0;
 }
```

Program Output

©Copyright 1999 Rogue Wave Software, Inc.

```
Type a string between 5 and 100 characters long.
roguewave
Changing the third character from g to *
now its: ro*uewave
Identifying the middle: ro*u(the middle is here!)ewave
I didn't like the word 'middle', so instead, I'll say:
ro*u(the center is here!)ewave
```

See Also

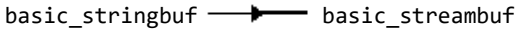
allocator, ***string***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_stringbuf



Summary

Associates the input or output sequence with a sequence of arbitrary characters.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	stringbuf
ios_type	string_type
off_type	traits_type
pos_type	wstringbuf

Member Functions

overflow()	
pbackfail()	
seekoff()	underflow()
seekpos()	xsputn()
setbuf()	
str()	

Synopsis

```
#include <sstream>
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_stringbuf
: public basic_streambuf<charT, traits>
```

Description

The class ***basic_stringbuf*** is derived from ***basic_streambuf***. Its purpose is to associate the input or output sequence with a sequence of arbitrary characters. The sequence can be initialized from, or made available as, an object of class ***basic_string***. Each object of type ***basic_stringbuf<charT, traits,Allocator>*** controls two character sequences:

- A character input sequence;
- A character output sequence.

©Copyright 1999 Rogue Wave Software, Inc.

Note: see ***basic_streambuf.***

The two sequences are related to each other, but are manipulated separately. This allows you to read and write characters at different positions in objects of type ***basic_stringbuf*** without any conflict (as opposed to the ***basic_filebuf*** objects, in which a joint file position is maintained).

Interface

```
template<class charT, class traits = char_traits<charT>,
         class allocator<charT> >
class basic_stringbuf
: public basic_streambuf<charT, traits> {

  public:

   typedef charT                        char_type;
   typedef typename traits::int_type    int_type;
   typedef typename traits::pos_type    pos_type;
   typedef typename traits::off_type    off_type;

   typedef basic_ios<charT, traits>     ios_type;
   typedef basic_string<charT, traits,
                        Allocator>      string_type;


   explicit basic_stringbuf(ios_base::openmode which =
                            (ios_base::in | ios_base::out));

   explicit basic_stringbuf(const string_type& str,
                            ios_base::openmode which =
                            (ios_base::in | ios_base::out));

   virtual ~basic_stringbuf();

   string_type str() const;
   void str(const string_type& str_arg);

 protected:

   virtual int_type underflow();
   virtual int_type pbackfail(int_type c = traits::eof());
   virtual int_type overflow(int_type c = traits::eof());
   virtual basic_streambuf<charT,traits>*
    setbuf(char_type *s,streamsize n);

   virtual pos_type seekoff(off_type off,
                            ios_base::seekdir way,
                            ios_base::openmode which =
                            ios_base::in | ios_base::out);

   virtual pos_type seekpos(pos_type sp,
                            ios_base::openmode which =
                            ios_base::in | ios_base::out);

   virtual streamsize xsputn(const char_type* s,
                             streamsize n);

};
```

Types

`char_type`

The type `char_type` is a synonym for the template parameter `charT`.

`ios_type`

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

`off_type`

The type `off_type` is a synonym of type `traits::off_type`.

©Copyright 1999 Rogue Wave Software, Inc.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

string_type

The type `string_type` is an instantiation of class `basic_string` on type `charT`.

stringbuf

The type `stringbuf` is an instantiation of class `basic_stringbuf` on type `char`:

```
typedef basic_stringbuf<char> stringbuf;
```

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wstringbuf

The type `wstringbuf` is an instantiation of class `basic_stringbuf` on type `wchar_t`:

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

Constructors

```
explicit basic_stringbuf(ios_base::openmode which =
                ios_base::in | ios_base::out);
```

Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing the open mode with `which`.

```
explicit basic_stringbuf(const string_type& str,
                ios_base::openmode which =
                ios_base::in | ios_base::out);
```

Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing the open mode with `which`. The string object `str` is copied to the underlying buffer. If the opening mode is `in`, initializes the input sequence to point to the first character of the buffer. If the opening mode is `out`, it initializes the output sequence to point to the first character of the buffer. If the opening mode is `out | app`, it initializes the output sequence to point to the last character of the buffer.

Destructors

```
virtual ~basic_stringbuf();
```

Destroys an object of class `basic_stringbuf`.

Member Functions

```
int_type
overflow(int_type c = traits::eof() );
```

If the output sequence has a put position available, and `c` is not `traits::eof()`, then this function writes `c` into it. If there is no position available, the function increases the size of the buffer by allocating more memory and then writes `c` at the new current put position. If the operation fails, the function returns `traits::eof()`. Otherwise it returns `traits::not_eof(c)`.

```
int_type
pbackfail( int_type c = traits::eof() );
```

Puts back the character designated by `c` into the input sequence. If `traits::eq_int_type(c,traits::eof())` returns `true`, the function moves the input sequence one position backward. If the operation fails, the function returns `traits::eof()`. Otherwise it returns `traits::not_eof(c)`.

```
pos_type
seekoff(off_type off, ios_base::seekdir way,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        ios_base::openmode which =
        ios_base::in | ios_base::out);
```

If the open mode is `in | out`, this function alters the stream position of both the input and the output sequences. If the open mode is `in`, it alters the stream position of only the input sequence, and if it is `out`, it alters the stream position of the output sequence. The new position is calculated by combining the two parameters `off` (displacement) and `way` (reference point). If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position.

```
pos_type
seekpos(pos_type sp,ios_base::openmode which =
        ios_base::in | ios_base::out);
```

If the open mode is `in | out`, `seekpos()` alters the stream position of both the input and the output sequences. If the open mode is `in`, it alters the stream position of the input sequence only, and if the open mode is `out`, it alters the stream position of the output sequence only. If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position.

```
basic_streambuf<charT,traits>*
setbuf(char_type*s, streamsize n);
```

A stringbuf maintains an underlying character array for storing buffered characters. This function gives you a way to resize or replace that buffer, with certain restrictions.

First of all, `n` must be greater than the number of characters currently in the buffer. If `n` is too small, then `setbuf` has no effect and returns a null pointer to indicate failure.

If `n` is large enough, then this function has one of two effects:

- If `s` is not a null pointer, then `setbuf` copies the buffered contents of the stringbuf into the `n` character array starting at `s` and installs `s` as the underlying character array used by the stringbuf. `s` replaces the old underlying array. In this case, the function returns `s` on success or a null pointer to indicate failure.

  Note that the stringbuf now owns `s` and deletes it if a subsequent call to `setbuf` replaces it or, failing that, when the stringbuf is destroyed. A program should not itself delete a pointer passed to `setbuf` if the call succeeds. And `s` must have been allocated from the heap using new.

- If `s` is a null pointer, then `setbuf` resizes the underlying character array to `n` characters. The function returns a pointer to the beginning of the resized array if the operation is successful, or a null pointer if not.

```
string_type
str() const;
```

Returns a string object of type `string_type` whose content is a copy of the underlying buffer contents.

```
void
str(const string_type& str_arg);
```

Clears the underlying buffer and copies the string object `str_arg` into it. If the opening mode is `in`, initializes the input sequence to point to the first character of the buffer. If the opening mode is `out`, the function initializes the output sequence to point to the first character of the buffer. If the opening mode is `out | app`, it initializes the output sequence to point to the last character of the buffer.

```
int_type
underflow();
```

If the input sequence has a read position available, the function returns the contents of this position. Otherwise it tries to expand the input sequence to match the output sequence, and if possible returns the content of the new current position. The function returns `traits::eof()` to indicate failure.

```
streamsize
xsputn(const char_type* s, streamsize n);
```

Writes up to `n` characters to the output sequence. The characters written are obtained from successive elements of the array whose first element is designated by `s`. The function returns the number of characters written.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
// stdlib/examples/manual/stringbuf.cpp
//
#include<iostream>
#include<sstream>
#include<string>

void main ( )
{
  using namespace std;

  // create a read/write string-stream object on tiny char
  // and attach it to an ostringstream object
  ostringstream out_1(ios_base::in | ios_base::out);

  // tie the istream object to the ostringstream object
  istream in_1(out_1.rdbuf());

  // output to out_1
  out_1 << "Here is the first output";

  // create a string object on tiny char
  string string_ex("l'heure est grave !");

  // open a read only string-stream object on tiny char
  // and initialize it
  istringstream in_2(string_ex);

  // output in_1 to the standard output
  cout << in_1.str() << endl;

  // output in_2 to the standard output
  cout << in_2.rdbuf() << endl;

  // reposition in_2 at the beginning
  in_2.seekg(0);

  stringbuf::pos_type pos;

  // get the current put position
  // equivalent to
  // out_1.tellp();
  pos = out_1.rdbuf()->pubseekoff(0,ios_base::cur,
                                  ios_base::out);

  // append the content of stringbuffer
  // pointed to by in_2 to the one
  // pointed to by out_1
  out_1 << ' ' << in_2.rdbuf();

  // output in_1 to the standard output
  cout << in_1.str() << endl;

  // position the get sequence
  // equivalent to
  // in_1.seekg(pos);
  in_1.rdbuf()->pubseekpos(pos, ios_base::in);

  // output "l'heure est grave !"
  cout << in_1.rdbuf() << endl << endl;
}
```

See Also

char_traits(3C++),*ios_base*(3C++), *basic_ios*(3C++), *basic_streambuf*(3C++), *basic_string*(3C++), *basic_istringstream*(3C++), *basic_ostringstream*(3C++), *basic_stringstream*(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.7.1

Standards Conformance

©Copyright 1999 Rogue Wave Software, Inc.

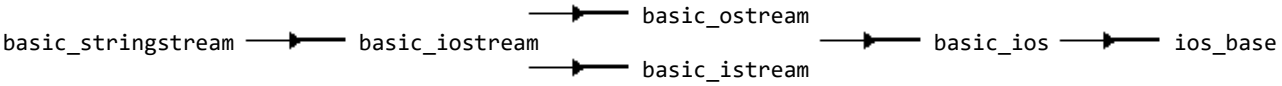
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

basic_stringstream



Summary

Supports writing and reading objects of class ***basic_string<charT,traits,Alocator>*** to/from an array in memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	pos_type	
int_type	sb_type	traits_type
ios_type	stringstream	wstringstream
off_type	string_type	

Member Functions

rdbuf()
str()

Synopsis

```
#include <sstream>
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_stringstream
: public basic_iostream<charT, traits>
```

Description

The template class ***basic_stringstream<charT,traits,Allocator>*** reads and writes to an array in memory. It supports writing and reading objects of class ***basic_string<charT,traits,Alocator>***. It uses a `basic_stringbuf` object to control the associated storage. It inherits from ***basic_iostream*** and therefore can use all the formatted and unformatted output and input functions.

Interface

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
```

©Copyright 1999 Rogue Wave Software, Inc.

```
class basic_stringstream
: public basic_iostream<charT, traits> {
 public:
  typedef traits                          traits_type;
  typedef charT                           char_type;
  typedef typename traits::int_type       int_type;
  typedef typename traits::pos_type       pos_type;
  typedef typename traits::off_type       off_type;

  typedef basic_stringbuf<charT, traits, Allocator> sb_type;
  typedef basic_ios<charT, traits> ios_type;
  typedef basic_string<charT, traits, Allocator>
          string_type;

  explicit basic_stringstream(ios_base::openmode which =
                              ios_base::out | ios_base::in);

  explicit basic_stringstream(const string_type& str,
                              ios_base::openmode which =
                              ios_base::out | ios_base::in);

  virtual ~basic_stringstream();

  basic_stringbuf<charT,traits,Allocator> *rdbuf() const;
  string_type str() const;
  void str(const string_type& str);

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

int_type

The type `int_type` is a synonym of type `traits::in_type`.

ios_type

The type `ios_type` is an instantiation of class `basic_ios` on type `charT`.

off_type

The type `off_type` is a synonym of type `traits::off_type`.

pos_type

The type `pos_type` is a synonym of type `traits::pos_type`.

sb_type

The type `sb_type` is an instantiation of class `basic_stringbuf` on type `charT`.

string_type

The type `string_type` is an instantiation of class `basic_string` on type `charT`.

stringstream

The type `stringstream` is an instantiation of class `basic_stringstream` on type `char`:

```
typedef basic_stringstream<char> stringstream;
```

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

wstringstream

The type `wstringstream` is an instantiation of class `basic_stringstream` on type `wchar_t`:

©Copyright 1999 Rogue Wave Software, Inc.

```
typedef basic_stringstream<wchar_t> wstringstream;
```

Constructors

```
explicit basic_stringstream(ios_base::openmode which =
                  ios_base::in | ios_base::out);
```

Constructs an object of class `basic_stringstream`, initializing the base class `basic_iostream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(which)`.

```
explicit basic_stringstream(const string_type& str,
                  ios_base::openmode which =
                  ios_base::in | ios_base::out);
```

Constructs an object of class `basic_stringstream`, initializing the base class `basic_iostream` with the associated string buffer. The string buffer is initialized by calling the `basic_stringbuf` constructor `basic_stringbuf<charT,traits,Allocator>(str,which)`.

Destructors

```
virtual ~basic_stringstream();
```

Destroys an object of class `basic_stringstream`.

Member Functions

```
basic_stringbuf<charT,traits,Allocator>*
rdbuf() const;
```

Returns a pointer to the `basic_stringbuf` associated with the stream.

```
string_type
str() const;
```

Returns a string object of type `string_type` whose contents is a copy of the underlying buffer contents.

```
void
str(const string_type& str);
```

Clears the string buffer and copies the string object `str` into it. If the opening mode is `in`, initializes the input sequence to point to the first character of the buffer. If the opening mode is `out`, initializes the output sequence to point to the first character of the buffer. If the opening mode is `out | app`, initializes the output sequence to point to the last character of the buffer.

Example

```
//
// stdlib/examples/manual/stringstream.cpp
//
#include<iostream>
#include<sstream>

void main ( )
{
  using namespace std;

  // create a bi-directional wstringstream object
  wstringstream inout;

  // output characters
  inout << L"Das ist die rede von einem man" << endl;
  inout << L"C'est l'histoire d'un home" << endl;
  inout << L"This is the story of a man" << endl;

  wchar_t p[100];

  // extract the first line
  inout.getline(p,100);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // output the first line to stdout
  wcout << endl << L"Deutch :" << endl;
  wcout << p;

  // extract the second line
  inout.getline(p,100);

  // output the second line to stdout
  wcout << endl << L"Francais :" << endl;
  wcout << p;

  // extract the third line
  inout.getline(p,100);

  // output the third line to stdout
  wcout << endl << L"English :" << endl;
  wcout << p;

  // output the all content of the
  //wstringstream object to stdout
  wcout << endl << endl << inout.str();
}
```

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***basic_stringbuf***(3C++), ***basic_string***(3C++), ***basic_istringstream***(3C++), ***basic_ostringstream***(3c++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.7.3

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Bidirectional Iterators

`Iterator`

Summary

An iterator that can both read and write and can traverse a container in both directions

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

NOTE:For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Bidirectional iterators can move both forwards and backwards through a container, and have the ability to both read and write data. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator descriptions listed below:

`a and b`	values of type `X`
`n`	value representing a `distance` between two iterators
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Bidirectional Iterators

A bidirectional iterator must meet all the requirements listed below. Note that most of these requirements are also the requirements for forward iterators.

`X u`	u might have a singular value
`X()`	X() might be singular
`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`r = a`	assignment, `r == a`
`a == b, a != b`	return value convertible to `bool`
`a->m`	equivalent to `(*a).m`
`*a`	return value convertible to `T&`
`++r`	returns `X&`

©Copyright 1999 Rogue Wave Software, Inc.

`r++`	return value convertible to const `X&`
`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return value convertible to const `X&`
`*r--`	returns `T&`

Like forward iterators, bidirectional iterators have the condition that `a == b` implies `*a== *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

See Also

Containers,*Iterators*, *Forward Iterators*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

binary_function

`Function Object`

Summary

Base class for creating binary function objects.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Arg1, class Arg2, class Result>
    struct binary_function{
         typedef Arg1 first_argument_type;
         typedef Arg2 second_argument_type;
         typedef Result result_type;
    };
```

Description

Function objects are objects with an `operator()` defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an `operator()` defined or a pointer to a function. The Standard C++ Library includes both a standard set of function objects, and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take two arguments are called *binary function objects.* Binary function objects are required to include the typedefs `first_argument_type`, `second_argument_type`, and `result_type`. The ***binary_function*** class makes the task of creating templatized binary function objects easier by including the necessary typedefs for a binary function object. You can create your own binary function objects by inheriting from ***binary_function***.

See Also

Function Objects, ***unary_function***, the Function Objects section of the User's Guide.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

binary_negate

`Function Object`

Summary

A function object that returns the complement of the result of its binary predicate.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator()()

Synopsis

```
#include <functional>
template<class Predicate>
class binary_negate ;
```

Description

binary_negate is a function object class with a return type for the function adaptor ***not2***. ***not2*** is a function adaptor, known as a negator, that takes a binary predicate function object as its argument and returns a binary predicate function object that is the complement of the original.

Note that ***not2*** works only with function objects that are defined as subclasses of the class ***binary_function***.

Interface

```
template<class Predicate>
class binary_negate
  : public binary_function<typename
                           Predicate::first_argument_type,
                           typename
                           Predicate::second_argument_type,
                           bool>
{
public:

  typedef typename binary_function<typename
   Predicate::first_argument_type, typename
   Predicate::second_argument_type,
       bool>::second_argument_type second_argument_type;

  explicit binary_negate (const Predicate&);
  bool operator()
       (const typename Predicate::first_argument_type&,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        const typename Predicate::second_argument_type&)
        const;
};


// Non-member Functions

template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Constructors

```
explicit binary_negate(const Predicate& pred);
```

Constructs a binary_negate object from predicate `pred`.

Operators

```
bool
operator()(const first_argument_type& x,
           const second_argument_type& y) const;
```

Returns the result of `pred(x,y)`.

See Also

binary_function, ***not2***, ***unary_negate***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

binary_search

`Algorithm`

Summary

Performs a binary search for a value on a container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value);
template <class ForwardIterator, class T, class Compare>
bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value, Compare comp);
```

Description

The ***binary_search*** algorithm, like other related algorithms (***equal_range***, ***lower_bound*** and ***upper_bound***) performs a binary search on ordered containers. All binary search algorithms have two versions. The first version uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version allows you to include a function object of type `Compare`, which it assumes was the function used to sort the sequence. The function object must be a binary predicate.

The ***binary_search*** algorithm returns `true` if a sequence contains an element equivalent to the argument `value`. The first version of ***binary_search*** returns `true` if the sequence contains at least one element that is equal to the search value. The second version of the ***binary_search*** algorithm returns `true` if the sequence contains at least one element that satisfies the conditions of the comparison function. Formally, ***binary_search*** returns `true` if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions:

```
!(*i < value) && !(value < *i)
```

or

```
comp(*i, value) == false && comp(value, *i) == false
```

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

binary_search performs at most `log(last - first) + 2` comparisons.

Example

```
  //
  // b_search.cpp
  //
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
  typedef vector<int>::iterator iterator;
  int d1[10] = {0,1,2,2,3,4,2,2,6,7};
  //
  // Set up a vector
  //
  vector<int> v1(d1,d1 + 10);
  //
  // Try binary_search variants
  //
  sort(v1.begin(),v1.end());
  bool b1 = binary_search(v1.begin(),v1.end(),3);
  bool b2 =
    binary_search(v1.begin(),v1.end(),11,less<int>());
  //
  // Output results
  //
  cout << "In the vector: ";
  copy(v1.begin(),v1.end(),
          ostream_iterator<int,char>(cout," "));

  cout << endl << "The number 3 was "
       << (b1 ? "FOUND" : "NOT FOUND");
  cout << endl << "The number 11 was "
       << (b2 ? "FOUND" : "NOT FOUND") << endl;
  return 0;
}
```

Program Output

```
In the vector: 0 1 2 2 2 2 3 4 6 7
The number 3 was FOUND
The number 11 was NOT FOUND
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

equal_range, ***lower_bound***, ***upper_bound***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

bind1st, bind2nd, binder1st, binder2nd

`Function Object`

Summary

Templatized utilities to bind values to function objects.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Operation>
class binder1st : public unary_function<typename
                  Operation::second_argument_type,
                  typename Operation::result_type> ;

template <class Operation, class T>
binder1st<Operation> bind1st (const Operation&, const T&);
template <class Operation>
class binder2nd : public unary_function<typename
                  Operation::first_argument_type,
                  typename Operation::result_type> ;

template <class Operation, class T>
binder2nd<Operation> bind2nd (const Operation&, const T&);
```

Description

Because so many functions included in the standard library take other functions as arguments, the library includes classes that let you build new function objects out of old ones. Both `bind1st()` and `bind2nd()` are functions that take as arguments a binary function object `f` and a value `x`, and return, respectively, classes ***binder1st*** and ***binder2nd***. The underlying function object must be a subclass of ***binary_function***.

Class ***binder1st*** binds the value to the first argument of the binary function, and ***binder2nd*** does the same thing for the second argument of the function. The resulting classes can be used in place of a unary predicate in other function calls.

For example, you could use the ***count_if*** algorithm to count all elements in a vector that are less than or equal to 7, using the following:

```
count_if (v.begin, v.end, bind1st(greater<int> (),7),
          littleNums)
```

©Copyright 1999 Rogue Wave Software, Inc.

This function adds one to `littleNums` each time the predicate is `true`, in other words, each time 7 is greater than the element.

Interface

```
// Class binder1st
 template <class Operation>
 class binder1st
   : public unary_function<typename
                           Operation::second_argument_type,
                           typename Operation::result_type>
{
public:

   binder1st(const Operation&,
             const typename
             Operation::first_argument_type&);
   typename Operation::result_type operator()
           (const typename Operation::second_argument_type&)
            const;
};

// Class binder2nd
 template <class Operation>
 class binder2nd
   : public unary_function<typename
                           Operation::first_argument_type,
                           typename Operation::result_type>
{
public:

   binder2nd(const Operation&,
             const typename
             Operation::second_argument_type&);
   typename Operation::result_type operator()
           (const typename Operation::first_argument_type&)
            const;
};

// Creator bind1st

   template <class Operation, class T>
   binder1st<Operation> bind1st (const Operation&,
                                 const T&);

// Creator bind2nd

   template<class Operation, class T>
   binder2nd <Operation> bind2nd(const Operation&,
                                 const T&);
```

Example

```
//
// binders.cpp
//
 #include <functional>
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;
 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[4] = {1,2,3,4};
   //
   // Set up a vector
   //
   vector<int> v1(d1,d1 + 4);
   //
   // Create an 'equal to 3' unary predicate by binding 3 to
   // the equal_to binary predicate.
   //
   binder1st<equal_to<int> > equal_to_3 =
      bind1st(equal_to<int>(),3);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   //
   // Now use this new predicate in a call to find_if
   //
   iterator it1 = find_if(v1.begin(),v1.end(),equal_to_3);
   //
   // Even better, construct the new predicate on the fly
   //
   iterator it2 =
      find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
   //
   // And now the same thing using bind2nd
   // Same result since == is commutative
   //
   iterator it3 =
      find_if(v1.begin(),v1.end(),bind2nd(equal_to<int>(),3));
   //
   // it3 = v1.begin() + 2
   //
   // Output results
   //
   cout << *it1 << " " << *it2 << " " << *it3 << endl;
   return 0;
 }
```

Program Output

```
3 3 3
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

bitset

`Container`

Summary

A template class and related functions for storing and manipulating fixed-size sequences of bits.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

any()	operator&=()	operator==()	
count()	operator>>()	operator^()	size()
flip()	operator>>=()	operator^=()	test()
none()	operator<<()	operator~()	to_ulong()
operator!=()	operator<<=()	reset()	
operator&()	operator=()	set()	

Synopsis

```
#include <bitset>
template <size_t N>
class bitset ;
```

Description

bitset<size_t N> is a class that describes objects that can store a sequence consisting of a fixed number of bits, `N`. Each bit represents either the value zero (`reset`) or one (`set`) and has a non-negative position `pos`.

Errors and Exceptions

Bitset constructors and member functions may report the following three types of errors - each associated with a distinct exception:

- Invalid-argument error or `invalid_argument()` exception;
- Out-of-range error or `out_of_range()` exception;
- Overflow error or `over-flow_error()` exception;

If exceptions are not supported on your compiler, you get an assertion failure instead of an exception.

©Copyright 1999 Rogue Wave Software, Inc.

Interface

```
template <size_t N>
class bitset {

public:

// bit reference:

  class reference {
   friend class bitset;
  public:

   ~reference();
   reference& operator= (bool);
   reference& operator= (const reference&);
   bool operator~() const;
   operator bool() const;
   reference& flip();
  };


// Constructors
  bitset ();
  bitset (unsigned long);
  template<class charT, class traits, class Allocator>
  explicit bitset
           (const basic_string<charT, traits, Allocator>,
            typename basic_string<charT, traits,
                                  Allocator>::size_type=0,
            typename basic_string<charT, traits,
                                  Allocator>::size_type=
            basic_string<charT, traits, Allocator>::npos);
  bitset (const bitset<N>&);
  bitset<N>& operator= (const bitset<N>&);

// Bitwise Operators and Bitwise Operator Assignment
   bitset<N>& operator&= (const bitset<N>&);
   bitset<N>& operator|= (const bitset<N>&);
   bitset<N>& operator^= (const bitset<N>&);
   bitset<N>& operator<<= (size_t);
   bitset<N>& operator>>= (size_t);

// Set, Reset, Flip
   bitset<N>& set ();
   bitset<N>& set (size_t, int = 1);
   bitset<N>& reset ();
   bitset<N>& reset (size_t);
   bitset<N> operator~() const;
   bitset<N>& flip ();
   bitset<N>& flip (size_t);

// element access
   reference operator[] (size_t);
   unsigned long to_ulong() const;
   template<class charT, class traits, class Allocator>
   basic_string<charT, traits, Allocator> to_string();
   size_t count() const;
   size_t size() const;
   bool operator== (const bitset<N>&) const;
   bool operator!= (const bitset<N>&) const;
   bool test (size_t) const;
   bool any() const;
   bool none() const;
   bitset<N> operator<< (size_t) const;
   bitset<N> operator>> (size_t) const;

};

// Non-member operators
template <size_t N>
bitset<N> operator& (const bitset<N>&, const bitset<N>&);

template <size_t N>
bitset<N> operator| (const bitset<N>&, const bitset<N>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <size_t N>
bitset<N> operator^ (const bitset<N>&, const bitset<N>&);

template <size_t N>
istream& operator>> (istream&, bitset<N>&);

template <size_t N>
ostream& operator<< (ostream&, const bitset<N>&)
```

Constructors

bitset();

Constructs an object of class `bitset<N>`, initializing all bit values to zero.

bitset(unsigned long val);

Constructs an object of class `bitset<N>`, initializing the first `M` bit values to the corresponding bits in `val`. `M` is the smaller of `N` and the value `CHAR_BIT * sizeof(unsigned long)`. If `M < N`, remaining bit positions are initialized to zero. Note: `CHAR_BIT` is defined in `<climits>`.

```
template<class charT, class traits, class Allocator>

explicit

bitset (const basic_string<charT, traits, Allocator>,
        typename basic_string<charT, traits,
                              Allocator>::size_type=0,
        typename basic_string<charT, traits,
                              Allocator>::size_type=
        basic_string<charT, traits, Allocator>::npos);
```

Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`. The function throws an `invalid_argument` exception if any of the `rlen` characters in `str`, beginning at position `pos`, is other than `0` or `1`. Otherwise, the function constructs an object of class ***bitset<N>***, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`. This constructor expects `pos <= str.size()`. If that is not `true`, the constructor throws an `out_of_range` exception.

bitset(const bitset<N>& rhs);

Creates a copy of `rhs`.

Assignment Operators

```
bitset<N>&
operator=(const bitset<N>& rhs);
```

Erases all bits in self, then inserts into self a copy of each bit in `rhs`. Returns a reference to `*this`.

Operators

```
bool
operator==(const bitset<N>& rhs) const;
```

Returns `true` if the value of each bit in `*this` equals the value of each corresponding bit in `rhs`. Otherwise returns `false`.

```
bool
operator!=(const bitset<N>& rhs) const;
```

Returns `true` if the value of any bit in `*this` is not equal to the value of the corresponding bit in `rhs`. Otherwise returns `false`.

```
bitset<N>&
operator&=(const bitset<N>& rhs);
```

Clears each bit in `*this` for which the corresponding bit in `rhs` is clear and leaves all other bits unchanged. Returns `*this`.

©Copyright 1999 Rogue Wave Software, Inc.

```
bitset<N>&
operator|=(const bitset<N>& rhs);
```

Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged. Returns `*this`.

```
bitset<N>&
operator^=(const bitset<N>& rhs);
```

Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged. Returns `*this`.

```
bitset<N>&
operator<<=(size_t pos);
```

Replaces each bit at position `I` with `0` if `I < pos` or with the value of the bit at `I - pos` if `I >= pos`. Returns `*this`.

```
bitset<N>&
operator>>=(size_t pos);
```

Replaces each bit at position `I` with `0` if `pos >= N-I` or with the value of the bit at position `I + pos` if `pos < N-I`. Returns `*this`.

```
bitset<N>&
operator>>(size_t pos) const;
```

Returns `bitset<N>(*this) >>= pos`.

```
bitset<N>&
operator<<(size_t pos) const;
```

Returns `bitset<N>(*this) <<= pos`.

```
bitset<N>
operator~() const;
```

Returns the bitset that is the logical complement of each bit in `*this`.

```
bitset<N>
operator&(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `AND` of `lhs` with `rhs`.

```
bitset<N>
operator|(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `OR` of `lhs` with `rhs`.

```
bitset<N>
operator^(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `XOR` of `lhs` with `rhs`.

```
template <size_t N>
istream&
operator>>(istream& is, bitset<N>& x);
```

Extracts up to `N` characters (single-byte) from `is`. Stores these characters in a temporary object `str` of type `string`, then evaluates the expression `x = bitset<N>(str)`. Characters are extracted and stored until any of the following occurs:

- `N` characters have been extracted and stored
- An end-of-file is reached on the input sequence
- The next character is neither '0' nor '1'. In this case, the character is not extracted

Returns `is`.

©Copyright 1999 Rogue Wave Software, Inc.

```
template <size_t N>
ostream&
operator<<(ostream& os, const bitset<N>& x);
```

Returns `os << x.to_string()`

Member Functions

```
bool
any() const;
```

Returns `true` if any bit in `*this` is set. Otherwise returns `false`.

```
size_t
count() const;
```

Returns a count of the number of bits set in `*this`.

```
bitset<N>&
flip();
```

Flips all bits in `*this`, and returns `*this`.

```
bitset<N>&
flip(size_t pos);
```

Flips the bit at position `pos` in `*this` and returns `*this`. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
bool
none() const;
```

Returns `true` if no bit in `*this` is set. Otherwise returns `false`.

```
bitset<N>&
reset();
```

Resets all bits in `*this`, and returns `*this`.

```
bitset<N>&
reset(size_t pos);
```

Resets the bit at position `pos` in `*this`. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
bitset<N>&
set();
```

Sets all bits in `*this`, and returns `*this`.

```
bitset<N>&
set(size_t pos, int val = 1);
```

Stores a new value in the bits at position `pos` in `*this`. If `val` is nonzero, the stored value is one, otherwise it is zero. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
size_t
size() const;
```

Returns the template parameter `N`.

```
bool
test(size_t pos) const;
```

Returns `true` if the bit at position `pos` is set. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>
to_string();
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns an object of type `string`, `N` characters long.

Each position in the new string is initialized with a character ('0' for zero and '1' for one) representing the value stored in the corresponding bit position of `*this`. Character position `N - 1` corresponds to bit position 0. Subsequent decreasing character positions correspond to increasing bit positions.

```
unsigned long
to_ulong() const;
```

Returns the integral value corresponding to the bits in `*this`. Throws an `overflow_error` if these bits cannot be represented as type `unsigned long`.

See Also

Containers

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

cerr

`Pre-defined stream`

Summary

Controls output to an unbuffered stream buffer associated with the object `stderr` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern ostream cerr;
ostream cerr;
```

Description

The object `cerr` controls output to an unbuffered stream buffer associated with the object `stderr` declared in `<cstdio>`. By default, the standard C and C++ streams are synchronized, but you can improve performance by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

Formatting

The formatting is done through member functions or manipulators. See `cout` or `basic_ostream` for details.

Example

```
//
// cerr example
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  ifstream in("file_name.txt");

  // output the all file to stdout
  if ( in )
    cout << in.rdbuf();
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  else
    // if the ifstream object is in a bad state
    // output an error message to stderr
    cerr << "Error while opening the file" << endl;
}
```

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cout***(3C++), ***cin***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***iomanip***(3C++), ***wclog***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.1

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

char_traits

Summary

A traits class with types and operations for the ***basic_string*** container and ***iostream*** classes.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
off_type
pos_type
state_type

Member Functions

assign()
compare()
copy()
eof()
eq()
eq_int_type()
find()
length()
lt()
move()
not_eof()
to_char_type()
to_int_type()

Synopsis

```
#include <string>
template<class charT>
struct char_traits
```

Description

The template structure ***char_traits<charT>*** defines the types and functions necessary to implement the ***iostreams*** and ***string*** template classes. It is templatized on `charT`, which represents the character container type. Each specialized version of ***char_traits<charT>*** includes the default definitions corresponding to the specialized character container type.

Users have to provide specialization for ***char_traits*** if they use character types other than `char` and `wchar_t`.

Interface

```
template<class charT>
struct char_traits {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef charT                    char_type;
  typedef INT_T                    int_type;
  typedef POS_T                    pos_type;
  typedef OFF_T                    off_type;
  typedef STATE_T                  state_type;

  static char_type    to_char_type(const int_type&);
  static int_type     to_int_type(const char_type&);
  static bool         eq(const char_type&,const char_type&);
  static bool         eq_int_type(const int_type&,
                                  const int_type&);

  static int_type         eof();
  static int_type         not_eof(const int_type&);

  static void             assign(char_type&,
                                 const char_type&);
  static bool             lt(const char_type&,
                             const char_type&);
  static int              compare(const char_type*,
                                  const char_type*,size_t);
  static size_t           length(const char_type*);
  static const char_type* find(const char_type*,int n,
                               const char_type&);

  static char_type*       move(char_type*,
                               const char_type*,size_t);
  static char_type*       copy(char_type*,const char_type*,
                               size_t);
  static char_type*       assign(char_type*,size_t,
                                 const char_type&);

};
```

Types

`char_type`

The type `char_type` represents the character container type. It must be convertible to `int_type`.

`int_type`

The type `int_type` is another character container type that can also hold an end-of-file value. It is used as the return type of some of the iostream class member functions. If `char_type` is either `char` or `wchar_t`, `int_type` is `int` or `wint_t`, respectively.

`off_type`

The type `off_type` represents offsets to positional information. It is used to represent:

- A signed displacement, measured in characters, from a specified position within a sequence.
- An absolute position within a sequence.

The value `off_type(-1)` can be used as an error indicator. Value of type `off_type` can be converted to type `pos_type`, but no validity of the resulting `pos_type` value is ensured.

If `char_type` is either `char` or `wchar_t`, `off_type` is `streamoff` or `wstreamoff`, respectively.

`pos_type`

The type `pos_type` describes an object that can store all the information necessary to restore an arbitrary sequence to a previous stream position and conversion state. The conversion `pos_type(off_type(-1))` constructs the invalid `pos_type` value to signal error.

If `char_type` is either `char` or `wchar_t`, `pos_type` is `streampos` or `wstreampos`, respectively.

`state_type`

The type `state_type` holds the conversion state, and is compatible with the function `locale::codecvt()`.

©Copyright 1999 Rogue Wave Software, Inc.

If `char_type` is either `char` or `wchar_t`, `state_type` is `mbstate_t`.

Types Default-Values

specialization type	on char	on wchar_t
`char_type`	`char`	`wchar_t`
`int_type`	`int`	`wint_t`
`off_type`	`streamoff`	`wstreamoff`
`pos_type`	`streampos`	`wstreampos`
`state_type`	`mbstate_t`	`mbstate_t`

Value Functions

```
void
assign(char_type& c1, const char_type& c2);
```

Assigns one character value to another. The value of `c2` is assigned to `c1`.

```
char_type*
assign(char_type* s, size_t n, const char_type& a);
```

Assigns one character value to `n` elements of a character array. The value of `a` is assigned to `n` elements of `s`.

```
char_type*
copy(char_type* s1, const char_type* s2, size_t n);
```

Copies `n` characters from the object pointed to by `s1` into the object pointed to by `s2`. The ranges of `(s1,s1+n)` and `(s2,s2+n)` may not overlap.

```
int_type
eof();
```

Returns an `int_type` value that represents the end-of-file. It is returned by several functions to indicate end-of-file state, or to indicate an invalid return value.

```
const char_type*
find(const char_type* s, int n, const char_type& a);
```

Looks for the value of `a` in `s`. Only `n` elements of `s` are examined. Returns a pointer to the matched element if one is found. Otherwise returns `0`.

```
size_t
length(const char_type* s);
```

Returns the length of a null terminated character string pointed to by `s`.

```
char_type*
move(char_type* s1, const char_type* s2, size_t n);
```

Moves `n` characters from the object pointed to by `s1` into the object pointed to by `s2`. The ranges of `(s1,s1+n)` and `(s2,s2+n)` may overlap.

```
int_type
not_eof(const int_type& c);
```

Returns `c` if `c` is not equal to the end-of-file value. Otherwise returns `0`.

Test Functions

```
int
compare(const char_type* s1,const char_type* s2,size_t n);
```

Compares `n` values from `s1` with `n` values from `s2`. Returns `1` if `s1` is greater than `s2`, `-1` if `s1` is less than `s2`, or `0` if they are equal.

```
bool
eq(const char_type& c1, const char_type& c2);
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns `true` if `c1` and `c2` represent the same character.

```
bool
eq_int_type(const int_type& c1, const int_type& c2);
```

Returns `true` if `c1` and `c2` are equal.

```
bool
lt(const char_type& c1,const char_type& c2);
```

Returns `true` if `c1` is less than `c2`.

Conversion Functions

```
char_type
to_char_type(const int_type& c);
```

Converts a valid character represented by a value of type `int_type` to the corresponding `char_type` value.

```
int_type
to_int_type(const char_type& c);
```

Converts a valid character represented by a value of type `char_type` to the corresponding `int_type` value.

See Also

iosfwd(3C++), *fpos*(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 21.1.4, 21.1.5, 27.1.2.

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

cin

`Pre-defined stream`

Summary

Controls input from a stream buffer associated with the object `stdin` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern istream cin;
istream cin;
```

Description

The object `cin` controls input from a stream buffer associated with the object `stdin` declared in `<cstdio>`. By default, the standard C and C++ streams are synchronized, but you can improve performance by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

After the object `cin` is initialized, `cin.tie()` returns `&cout`, which implies that `cin` and `cout` are synchronized.

Example

```
//
// cin example #1
//
#include <iostream>

void main ( )
{
  using namespace std;

  int i;
  float f;
  char c;

  //read an integer, a float and a character from stdin
  cin >> i >> f >> c;

  // output i, f and c to stdout
  cout << i << endl << f << endl << c << endl;
}

//
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// cin example #2
//
#include <iostream>

void main ( )
{
  using namespace std;

  char p[50];

  // remove all the white spaces
  cin >> ws;

  // read characters from stdin until a newline
  // or 49 characters have been read
  cin.getline(p,50);

  // output the result to stdout
  cout << p;
}
```

When inputting `" Grendel the monster" (newline)` in the previous test, the output is `"Grendel the monster"`. The manipulator `ws` removes spaces.

See Also

basic_istream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cout***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***wclog***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.1

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

clog

`Pre-defined stream`

Summary

Controls output to a stream buffer associated with the object `stderr` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern ostream clog;
ostream clog;
```

Description

The object `clog` controls output to a stream buffer associated with the object `stderr` declared in `<cstdio>`. The difference between `clog` and `cerr` is that `clog` is buffered but `cerr` is not. Therefore, commands like `clog << "ERROR !!";` and `fprintf(stderr,"ERROR !!");` are not synchronized.

Formatting

The formatting is done through member functions or manipulators. See ***cout*** or ***basic_ostream*** for details.

Example

```
//
// clog example
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  ifstream in("file_name.txt");

  // output the all file to stdout
  if ( in )
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    cout << in.rdbuf();
  else
    // if the ifstream object is in a bad state
    // output an error message to stderr
    clog << "Error while opening the file" << endl;
}
```

Warnings

clog can be used to redirect some of the errors to another recipient. For example, you might want to redirect them to a file named `my_err`:

```
ofstream out("my_err");
if ( out )
  clog.rdbuf(out.rdbuf());

else

  cerr << "Error while opening the file" << endl;
```

Then when you are doing something like `clog << "error number x";` the error message is output to the file `my_err`. Obviously, you can use the same scheme to redirect ***clog*** to other devices.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cout***(3C++), ***cin***(3C++), ***cerr***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***wclog***(3C++), ***iomanip***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++),

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.1

Standards Conformance

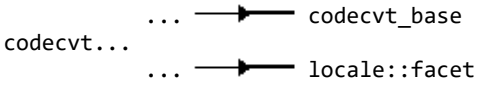
ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

codecvt



Summary

A code conversion facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

extern_type
id
intern_type
state_type

Member Functions

always_noconv()
do_always_noconv()
do_encoding()
do_in()
do_length()
do_max_length()
do_out()
encoding()
in()
length()
max_length()
out()
unshift()

Synopsis

```
#include <locale>
class codecvt_base;
template <class internT, class externT, class stateT>
class codecvt;
```

Description

The ***codecvt<internT,externT,stateT>*** template has code conversion facilities. Default implementations of ***codecvt<char,wchar_t,mbstate_t>*** and ***codecvt<wchar_t,char,mbstate_t>*** use `ctype<wchar_t>::widen` and `ctype<wchar_t>::narrow` respectively. The default implementation of ***codecvt<wchar_t,wchar_t,mbstate_t>*** simply uses `memcpy` (no particular conversion applied).

©Copyright 1999 Rogue Wave Software, Inc.

Interface

```
class codecvt_base {
public:
  enum result { ok, partial, error, noconv };
};

template <class internT, class externT, class stateT>
class codecvt : public locale::facet, public codecvt_base {
public:
  typedef internT  intern_type;
  typedef externT  extern_type;
  typedef stateT   state_type;

  explicit codecvt(size_t = 0)
  result out(stateT&, const internT*,
             const internT*, const internT*&,
             externT*, externT*, externT*&) const;
  result unshift(stateT&, externT*, externT*, extern*&)
                 const;
  result in(stateT&, const externT*,
            const externT*, const externT*&,
            internT*, internT*, internT*&) const;

  int encoding() const throw();
  bool always_noconv() const throw();

  int length(const stateT&, const externT*, const externT*,
             size_t) const;

  int max_length() const throw();
  static locale::id id;

protected:
  ~codecvt();  // virtual
  virtual result do_out(stateT&,
                        const internT*,
                        const internT*,
                        const internT*&,
                        externT*, externT*,
                        externT*&) const;
  virtual result do_in(stateT&,
                       const externT*,
                       const externT*,
                       const externT*&,
                       internT*, internT*,
                       internT*&) const;
  virtual result do_unshift(stateT&,
                            externT*, externT*,
                            externT*&) const;

  virtual int do_encoding() const throw();
  virtual bool do_always_noconv() const throw();
  virtual int do_length(const stateT&, const internT*,
                        const internT*,
                        size_t) const;

  virtual int do_max_length() const throw();
};
```

Types

intern_type

Type of character to convert from.

extern_type

Type of character to convert to.

state_type

Type to keep track of state and determine the codeset(s) to be converted.

©Copyright 1999 Rogue Wave Software, Inc.

Constructors

```
explicit codecvt(size_t refs = 0)
```

Construct a ***codecvt*** facet. If the `refs` argument is `0` (default), then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

Destructors

```
~codecvt();  // virtual and protected
```

Destroy the facet.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***codecvt*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the public `length` function simply calls its protected cousin `do_length`.

```
bool
always_noconv() const
  throw();
int
encoding() const
  throw();
result
in(stateT& state, const externT* from,
   const externT* from_end, const externT*& from_next,
   internT* to, internT* to_limit, internT*& to_next) const;
int
length(const stateT& state, const internT* from,
       const internT* end,
       size_t max) const;
int
max_length() const
  throw();
result
out(stateT& state, const internT* from,
    const internT* from_end, const internT*& from_next,
    externT* to, externT* to_limit, externT*& to_next)
    const;
result
unshift(stateT& state, externT* to, externT* to_limit,
        externT*& to) const;
```

Each of these public member functions `xxx` simply calls the corresponding protected `do_xxx` function.

Protected Member Functions

```
virtual bool
do_always_noconv() const
  throw();
```

Returns `true` if no conversion is required. This is the case if `do_in` and `do_out` return `noconv` for all valid arguments. The instantiation `codecvt<char,char,mbstate_t>` returns `true`, while all other default instantiations return `false`.

```
virtual int
do_encoding() const
  throw();
```

Returns one of the following

©Copyright 1999 Rogue Wave Software, Inc.

- `-1` if the encoding on the external character sequence is dependent on state.
- A constant number representing the number of external characters per internal character in a fixed width encoding.
- `0` if the encoding is uses a variable width.

```
virtual result
do_in(stateT& state,
      const externT* from,
      const externT* from_end,
      const externT*& from_next,
      internT* to, internT* to_limit,
      internT*& to_next) const;
virtual result
do_out(stateT& state,
       const internT* from,
       const internT* from_end,
       const internT*& from_next,
       externT* to, externT* to_limit,
       externT*& to_next) const;
```

Both functions take characters in the range of `[from,from_end)`, apply an appropriate conversion, and place the resulting characters in the buffer starting at `to`. Each function converts at most `from_end-from internT` characters, and stores no more than `to_limit-to externT` characters. Both `do_out` and `do_in` stop if they find a character they cannot convert. In any case, `from_next` and `to_next` are always left pointing to the next character beyond the last one successfully converted.

`do_out` and `do_in` must be called under the following pre-conditions:

- `from <= from_end`
- `to <= to_end`
- `state` is either initialized to the beginning of a sequence or equal to the result of the previous conversion on the sequence.

In the case where no conversion is required, `from_next` is set to `from` and `to_next` set to `to`.

`do_out` and `do_in` return one the following:

Return Value	Meaning
`ok`	`completed the conversion`
`partial`	`not all source characters converted`
`error`	`encountered a source character it could not convert`
`noconv`	`no conversion was needed`

If either function returns `partial` and `(from == from_end)`, then one of two conditions prevail:

- The destination sequence has not accepted all the converted characters, or
- Additional source characters are needed before another destination element can be assembled.

```
virtual int
do_length(const stateT&, const externT* from,
          const externT* end,
          size_t max) const;
```

Determines the largest number `<= max` of `internT` characters that can be produced from the sequence `[from,end)`, and returns the number of `externT` characters that would be consumed from `[from,end)` in order to produce this number of `internT` characters.

`do_length` must be called under the following pre-conditions:

- `from <= from_end`
- `state` is either initialized to the beginning of a sequence or equal to the result of the previous conversion on the sequence.

©Copyright 1999 Rogue Wave Software, Inc.

```
virtual int
do_max_length() const throw();
```

Returns the maximum value that `do_length` can return for any valid combination of its first three arguments, with the fourth argument (`max`) set to `1`.

```
virtual result
do_out(stateT& state,
       const internT* from,
       const internT* from_end,
       const internT*& from_next,
       externT* to, externT* to_limit,
       externT*& to_next) const;
```

See `do_in` above.

```
virtual result
do_unshift(stateT& state, externT* to, externT* to_limit,
           externT*& to_next) const;
```

Determines the sequence of `externT` characters that should be appended to a sequence whose state is given by `state`, in order to terminate the sequence (that is, to return it to the default or initial or unshifted state). Stores the terminating sequence starting at `to`, proceeding no farther than `to_limit`. Sets `to_end` to point past the last `externT` character stored.

`do_unshift` must be called under the following pre-conditions:

- `from <= from_end`
- `state` is either initialized to the beginning of a sequence or equal to the result of the previous conversion on the sequence.

The return value from `do_unshift` is as follows:

Return Value	Meaning
ok	terminating sequence was stored successfully
partial	only part of the sequence was stored
error	the state is invalid
noconv	no terminating sequence is needed for this state

Example

```
//
// codecvt.cpp
//
#include <sstream>
#include "codecvte.h"

int main ()
{
  using namespace std;

  mbstate_t state;

  // A string of ISO characters and buffers to hold
  // conversions
  string ins("\xfc \xcc \xcd \x61 \xe1 \xd9 \xc6 \xf5");
  string ins2(ins.size(),'.');
  string outs(ins.size(),'.');

  // Print initial contents of buffers
  cout << "Before:\n" << ins << endl;
  cout << ins2 << endl;
  cout << outs << endl << endl;

  // Initialize buffers
  string::iterator in_it = ins.begin();
  string::iterator out_it = outs.begin();

  // Create a user defined codecvt fact
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // This facet converts from ISO Latin
  // Alphabet No. 1 (ISO 8859-1) to
  // U.S. ASCII code page 437
  // This facet replaces the default for
  // codecvt<char,char,mbstate_t>
  locale loc(locale(),new ex_codecvt);

  // Now get the facet from the locale
  const codecvt<char,char,mbstate_t>& cdcvt =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
    use_facet<codecvt<char,char,mbstate_t> >(loc);
#else
    use_facet(loc,(codecvt<char,char,mbstate_t>*)0);
#endif

  // convert the buffer
  cdcvt.in(state,ins.begin(),ins.end(),in_it,
           outs.begin(),outs.end(),out_it);

  cout << "After in:\n" << ins << endl;
  cout << ins2 << endl;
  cout << outs << endl << endl;

  // Lastly, convert back to the original codeset
  in_it = ins.begin();
  out_it = outs.begin();
  cdcvt.out(state, outs.begin(),outs.end(),out_it,
            ins2.begin(),ins2.end(),in_it);

  cout << "After out:\n" << ins << endl;
  cout << ins2 << endl;
  cout << outs << endl;

  return 0;
}
```

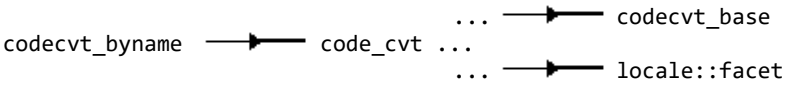
See Also

locale, *facets*, *codecvt_byname*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

codecvt_byname



Summary

A facet that includes code set conversion classification facilities based on the named locales.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT> class codecvt_byname;
```

Description

The ***codecvt_byname*** template includes the same functionality as the ***codecvt*** template, but specific to a particular named locale. For a description of the member functions of ***codecvt_byname,*** see the reference for ***codecvt.*** Only the constructor is described here.

Interface

```
template <class fromT, class toT, class stateT>
class codecvt_byname : public codecvt<fromT, toT, stateT> {
public:
  explicit codecvt_byname(const char*, size_t refs = 0);
protected:
  ~codecvt_byname();  // virtual
  virtual result do_out(stateT&,
                        const internT*,
                        const internT*,
                        const internT*&,
                        externT*, externT*,
                        externT*&) const;
  virtual result do_in(stateT&,
                       const externT*,
                       const externT*,
                       const externT*&,
                       internT*, internT*,
                       internT*&) const;
  virtual result do_unshift(stateT&,
                            externT*, externT*,
                            externT*&) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  virtual bool do_always_noconv() const throw();
  virtual int do_length(const stateT&, const externT*,
                        const externT*, size_t max) const;
  virtual int do_max_length() const throw();
  virtual int do_encoding() const throw();
};
```

Constructors

```
explicit codecvt_byname(const char* name, size_t refs = 0);
```

Construct a ***codecvt_byname*** facet. The facet provides codeset conversion relative to the named locale specified by the `name` argument. If the `refs` argument is `0`, destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, the object must be explicitly deleted: the locale does not do so.

See Also

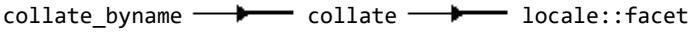
locale, ***facets***, ***codecvt***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

collate, collate_byname



Summary

A string collation, comparison, and hashing facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
string_type

Member Functions

compare()
do_compare()
do_hash()
do_transform()
hash()
transform()

Synopsis

```
#include <locale>
template <class charT> class collate;
template <class charT> class collate_byname;
```

Description

The ***collate*** and ***collate_byname*** facets allow for string collation, comparison, and hashing. Use the ***collate*** facet for the "C" locale, and use the ***collate_byname*** for named locales.

Interface

```
template <class charT>
class collate : public locale::facet {
public:
  typedef charT               char_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef basic_string<charT> string_type;
  explicit collate(size_t refs = 0);
  int compare(const charT*, const charT*,
              const charT*, const charT*) const;
  string_type transform(const charT*, const charT*) const;
  long hash(const charT*, const charT*) const;
  static locale::id id;
protected:
  ~collate();  // virtual
  virtual int do_compare(const charT*, const charT*,
                         const charT*, const charT*) const;
  virtual string_type do_transform(const charT*,
                                   const charT*) const;
  virtual long do_hash (const charT*, const charT*) const;
};

template <class charT>
class collate_byname : public collate<charT> {
public:
  typedef basic_string<charT> string_type;
  explicit collate_byname(const char*, size_t = 0);
protected:
  ~collate_byname();  // virtual
  virtual int do_compare(const charT*, const charT*,
                         const charT*, const charT*) const;
  virtual string_type do_transform(const charT*,
                                   const charT*) const;
  virtual long do_hash(const charT*, const charT*) const;
};
```

Types

`char_type`

Type of character the facet is instantiated on.

`string_type`

Type of character string returned by member functions.

Constructors

`explicit` **`collate`**`(size_t refs = 0)`

Construct a ***collate*** facet. If the `refs` argument is `0`, destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, the object must be explicitly deleted: the locale does not do so.

`explicit` **`collate_byname`**`(const char* name, size_t refs = 0);`

Construct a ***collate_byname*** facet. Use the named locale specified by the name argument. The `refs` argument serves the same purpose as it does for the ***collate*** constructor.

Destructors

```
~collate();  // virtual and protected
~collate_byname();  // virtual and protected
```

Destroy the facet.

Facet ID

`static locale::id` **`id`**`;`

Unique identifier for this type of facet.

Public Member Functions

©Copyright 1999 Rogue Wave Software, Inc.

The public members of the ***collate*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `compare` function simply calls its protected cousin `do_compare`.

```
int
compare(const charT* low1, const charT* high1,
        const charT* low2, const charT* high2) const;
long
hash(const charT* low, const charT* high) const;
string_type
transform(const charT* low, const charT* high) const;
```

Each of these public member functions `xxx` simply call the corresponding protected `do_xxx` function.

Protected Member Functions

```
virtual int
do_compare(const charT* low1, const charT* high1,
           const charT* low2, const charT* high2) const;
```

Returns `1` if the character string represented by the range `[low1,high1)` is greater than the character string represented by the range `[low2,high2)`, `-1` if first string is less than the second, or `0` if the two are equal. The default instantiations, `collate<char>` and `collate<wchar_t>`, perform a lexicographical comparison.

```
virtual long
do_hash( const charT* low, const charT* high)
```

Generates a hash value from a string defined by the range of characters `[low,high)`. Given two strings that compare equal (in other words, `do_compare` returns `0`), `do_hash` returns an integer value that is the same for both strings. For differing strings the probability that the return value is equal is approximately:

```
1.0/numeric_limits<unsigned long>::max()
```

```
virtual string_type
do_transform(const charT* low, const charT* high) const;
```

Returns a string that yields the same result in a lexicographical comparison with another string returned from transform as does the `do_compare` function applied to the original strings. In other words, the result of applying a lexicographical comparison to two strings returned from `transform` is the same as applying `do_compare` to the original strings passed to transform.

Example

```
//
// collate.cpp
//
#include <iostream>

int main ()
{
  using namespace std;
  locale loc;
  string s1("blue");
  string s2("blues");
  // Get a reference to the collate<char> facet
  const collate<char>& co =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
      use_facet<collate<char> >(loc);
#else
      use_facet(loc,(collate<char>*)0);
#endif
  // Compare two strings
  cout << co.compare(s1.begin(),s1.end(),
                     s2.begin(),s2.end()-1) << endl;
  cout << co.compare(s1.begin(),s1.end(),
                     s2.begin(),s2.end()) << endl;
  // Retrieve hash values for two strings
  cout << co.hash(s1.begin(),s1.end()) << endl;
  cout << co.hash(s2.begin(),s2.end()) << endl;
  return 0;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

See Also

locale, *facets*, *ctype*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

compare

Summary

A binary function or a function object that returns `true` or `false`. ***compare*** objects are typically passed as template parameters, and used for ordering elements within a container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

See Also

binary_function, ***Function Objects***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

complex

Complex Number Library

Summary

C++ complex number library

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

abs()	log10()	operator+=()	pow()
arg()	norm()	operator-()	real()
conj()	operator!=()	operator-=()	sin()
cos()	operator>>()	operator/()	sinh()
cosh()	operator<<()	operator/=()	sqrt()
exp()	operator*()	operator=()	tan()
imag()	operator*=()	operator==()	tanh()
log()	operator+()	polar()	

Specializations

```
complex <float>
complex <double>
complex <long double>
```

Synopsis

```
#include <complex>
template <class T>
class complex;
class complex<float>;
class complex<double>;
class complex<long double>;
```

Description

©Copyright 1999 Rogue Wave Software, Inc.

complex<T> is a class that supports complex numbers. A complex number has a real part and an imaginary part. The ***complex*** class supports equality, comparison and basic arithmetic operations. In addition, mathematical functions such as exponents, logarithms, powers, and square roots are also available.

Interface

```
template <class T>
class complex {

public:
   typedef T value_type;

   complex (const T& re = T(), const T& im = T());
   complex (const complex&);
   template <class X> complex
    (const complex<X>&);

   T real () const;
   T imag () const;

   complex<T>& operator= (const T&);
   complex<T>& operator+=(const T&);
   complex<T>& operator-=(const T&);
   complex<T>& operator*=(const T&);
   complex<T>& operator/=(const T&);

   template <class X>
    complex<T>& operator= (const complex<X>&);

   template <class X>
    complex<T>& operator+= (const complex<X>&);
   template <class X>
    complex<T>& operator-= (const complex<X>&);
   template <class X>
    complex<T>& operator*= (const complex<X>&);
   template <class X>
    complex<T>& operator/= (const complex<X>&);
};
// Non-member Operators

template<class T>
 complex<T> operator+ (const complex<T>&,
                       const complex<T>&);
template<class T>
 complex<T> operator+ (const complex<T>&, T&);
template<class T>
 complex<T> operator+ (T, const complex<T>&);

template<class T>
 complex<T> operator- (const complex<T>&,
                       const complex<T>&);
template<class T>
 complex<T> operator- (const complex<T>&, T&);
template<classT>
 complex<T> operator- (T, const complex<T>&);

template<class T>
 complex<T> operator* (const complex<T>&,
                       const complex<T>&);
template<class T>
 complex<T> operator* (const complex<T>&, T&);
template<class T>
 complex<T> operator* (T, const complex<T>&);

template<class T>
 complex<T> operator/ (const complex<T>&,
                       const complex<T>&);
template<class T>
 complex<T> operator/ (const complex<T>&, T&);
template<class T>
 complex<T> operator/ (T, const complex<T>&);

template<class T>
 complex<T> operator+ (const complex<T>&);
template<class T>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 complex<T> operator- (const complex<T>&);

template<class T>
 bool operator== (const complex<T>&, const complex<T>&);
template<class T>
 bool operator== (const complex<T>&, T&);
template<class T>
 bool operator== (T, const complex<T>&);

template<class T>
 bool operator!= (const complex<T>&, const complex<T>&);
template<class T>
 bool operator!= (const complex<T>&, T&);
template<class T>
 bool operator!= (T, const complex<T>&);

template <class T, class charT, class traits>
 basic_istream<charT, traits>& operator>>
              (istream&, complex<T>&);
template <class T, class charT, class traits>
 basic_ostream<charT, traits>& operator<<
              (ostream&, const complex<T>&);

// Values
template<class T> T real (const complex<T>&);
template<class T> T imag (const complex<T>&);

template<class T> T abs (const complex<T>&);
template<class T> T arg (const complex<T>&);
template<class T> T norm (const complex<T>&);

template<class T> complex<T> conj (const complex<T>&);
template<class T> complex<T> polar (const T&, const T&);

// Transcendentals
template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp (const complex<T>&);
template<class T> complex<T> log (const complex<T>&);

template<class T> complex<T> log10 (const complex<T>&);

template<class T> complex<T> pow (const complex<T>&, int);
template<class T> complex<T> pow (const complex<T>&, T&);
template<class T> complex<T> pow (const complex<T>&,
                                  const complex<T>&);
template<class T> complex<T> pow (const T&,
                                  const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);
template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);
```

Constructors

```
complex
(const T& re_arg = T(), const T& im_arg = T());
```

Constructs an object of class ***complex***, initializing `re_arg` to the real part and `im_arg` to the imaginary part.

```
template <class X> complex
(const complex<X>&);
```

Constructs a complex number from another complex number.

Assignment Operators

```
complex<T>& operator=(const T& v);
```

Assigns `v` to the real part of itself, setting the imaginary part to `0`.

```
complex<T>& operator+=(const T& v);
```

©Copyright 1999 Rogue Wave Software, Inc.

Adds v to the real part of itself, then returns the result.

```
complex<T>& operator-=(const T& v);
```

Subtracts v from the real part of itself, then returns the result.

```
complex<T>& operator*=(const T& v);
```

Multiplies v by the real part of itself, then returns the result.

```
complex<T>& operator/=(const T& v);
```

Divides v by the real part of itself, then returns the result.

```
template <class X>
complex<T>
operator=(const complex<X>& c);
```

Assigns c to itself.

```
template <class X>
complex<T>
operator+=(const complex<X>& c);
```

Adds c to itself, then returns the result.

```
template <class X>
complex<T>
operator-=(const complex<X>& c);
```

Subtracts c from itself, then returns the result.

```
template <class X>
complex<T>
operator*=(const complex<X>& c);
```

Multiplies itself by c, then returns the result.

```
template <class X>
complex<T>
operator/=(const complex<X>& c);
```

Divides itself by c, then returns the result.

Member Functions

```
T
imag() const;
```

Returns the imaginary part of the complex number.

```
T
real() const;
```

Returns the real part of the complex number.

Non-member Operators

```
template<class T> complex<T>
operator+(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator+(const complex<T>& lhs, const T& rhs);
template<class T> complex<T>
operator+(const T& lhs, const complex<T>& rhs);
```

Returns the sum of lhs and rhs.

```
template<class T> complex<T>
operator-(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator-(const complex<T>& lhs, const T& rhs);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class T> complex<T>
operator-(const T& lhs, const complex<T>& rhs);
```

Returns the difference of `lhs` and `rhs`.

```
template<class T> complex<T>
operator*(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator*(const complex<T>& lhs, const T& rhs);
template<class T> complex<T>
operator* (const T& lhs, const complex<T>& rhs);
```

Returns the product of `lhs` and `rhs`.

```
template<class T> complex<T>
operator/(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator/(const complex<T>& lhs, const T& rhs);
template<class T> complex<T>
operator/(const T& lhs, const complex<T>& rhs);
```

Returns the quotient of `lhs` divided by `rhs`.

```
template<class T> complex<T>
operator+(const complex<T>& rhs);
```

Returns `rhs`.

```
template<class T> complex<T>
operator-(const complex<T>& lhs);
```

Returns `complex<T>(-lhs.real(), -lhs.imag())`.

```
template<class T> bool
operator==(const complex<T>& x, const complex<T>& y);
```

Returns `true` if the real and imaginary parts of `x` and `y` are equal.

```
template<class T> bool
operator==(const complex<T>& x, const T& y);
```

Returns `true` if `y` is equal to the real part of `x` and the imaginary part of `x` is equal to `0`.

```
template<class T> bool
operator==(const T& x, const complex<T>& y);
```

Returns `true` if `x` is equal to the real part of `y` and the imaginary part of `y` is equal to `0`.

```
template<class T> bool
operator!=(const complex<T>& x, const complex<T>& y);
```

Returns `true` if either the real or the imaginary part of `x` and `y` are not equal.

```
template<class T> bool
operator!=(const complex<T>& x, const T& y);
```

Returns `true` if `y` is not equal to the real part of `x` or the imaginary part of `x` is not equal to `0`.

```
template<class T> bool
operator!=(const T& x, const complex<T>& y);
```

Returns `true` if `x` is not equal to the real part of `y` or the imaginary part of `y` is not equal to `0`.

```
template <class T, class charT, class traits>
          basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

Reads a complex number `x` into the input stream `is`. `x` may be of the form `u`, `(u)`, or `(u,v)` where `u` is the real part and `v` is the imaginary part. If bad input is encountered, `is.setstate(ios::failbit)` is called.

```
template <class T, class charT, class traits>
          basic_ostream<charT, traits>&
```

©Copyright 1999 Rogue Wave Software, Inc.

```
operator<<(basic_ostream<charT, traits>& os,
           const complex<T>& x);
```

Returns `os << "(" << x.real() << "," << x.imag() << ")"`.

Non-member Functions

```
template<class T> T
abs(const complex<T>& c);
```

Returns the absolute value or magnitude of `c` (the square root of the norm).

```
template<class T> T
arg(const complex<T>& x);
```

Returns the phase angle of `x` or `atan2(imag(x), real(x))`.

```
template<class T> complex<T>
conj(const complex<T>& c);
```

Returns the conjugate of `c`.

```
template<class T> complex<T>
cos(const complex<T>& c);
```

Returns the cosine of `c`.

```
template<class T> complex<T>
cosh(const complex<T>& c);
```

Returns the hyperbolic cosine of `c`.

```
template<class T> complex<T>
exp(const complex<T>& x);
```

Returns `e` raised to the `x` power.

```
template<class T> T
imag(const complex<T>& c) const;
```

Returns the imaginary part of `c`.

```
template<class T> complex<T>
log(const complex<T>& x);
```

Returns the complex natural (base e) logarithm of `x`, in the range of a strip mathematically unbounded along the real axis and in the interval [-i times pi, i times pi] along the imaginary axis. When `x` is a nega- tive real number, `imag(log(x))` is pi.

The branch cuts are along the negative real axis.

```
template<class T> complex<T>
log10(const complex<T>& x);
```

Returns the complex common (base 10) logarithm of `x`, defined as log(x)/log(10).

The branch cuts are along the negative real axis.

```
template<class T> T
norm(const complex<T>& c);
```

Returns the squared magnitude of `c`. (The sum of the squares of the real and imaginary parts.)

```
template<class T> complex<T>
polar(const T& m, const T& a = 0);
```

Returns the complex value of a complex number whose magnitude is `m` and phase angle is `a`, measured in radians.

```
template<class T> complex<T>
pow(const complex<T>& x, int y);
template<class T> complex<T>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
pow(const complex<T>& x, const T& y);
template<class T> complex<T>
pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T>
pow(const T& x, const complex<T>& y);
```

Returns `x` raised to the `y` power; or, if called with `(0, 0)`, returns `complex <T>(1,0)`. The branch cuts are along the negative real axis.

```
template<class T> T
real(const complex<T>& c);
```

Returns the real part of `c`.

```
template<class T> complex<T>
sin(const complex<T>& c);
```

Returns the sine of `c`.

```
template<class T> complex<T>
sinh(const complex<T>& c);
```

Returns the hyperbolic sine of `c`.

```
template<class T> complex<T>
sqrt(const complex<T>& x);
```

Returns the complex square root of `x`, in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis. The branch cuts are along the negative real axis.

```
template<class T> complex<T>
tan(const complex<T>& x);
```

Returns the tangent of `x`.

```
template<class T> complex<T>
tanh(const complex<T>& x);
```

Returns the hyperbolic tangent of `x`.

Example

```
//
// complex.cpp
//
 #include <complex>
 #include <iostream>
 using namespace std;

 int main()
 {
   complex<double> a(1.2, 3.4);
   complex<double> b(-9.8, -7.6);

   a += b;
   a /= sin(b) * cos(a);
   b *= log(a) + pow(b, a);

   cout << "a = " << a << ", b = " << b << endl;

   return 0;
 }
```

Program Output

```
a = (1.42804e-06,-0.0002873), b = (58.2199,69.7354)
```

Warnings

On compilers that don't support member function templates, the arithmetic operators do not work on any arbitrary type; they work only on float, double and long doubles. Also, you can perform binary arithmetic only on types that are the

©Copyright 1999 Rogue Wave Software, Inc.

same.

Compilers that don't support non-converting constructors permit unsafe downcasts (for example, long double to double, double to float, long double to float).

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Containers

Summary

A standard template library (STL) collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

Within the standard template library, collection classes are often described as containers. A container stores a collection of other objects and includes basic functions that support the use of generic algorithms. Containers come in two types: sequences and associative containers. They are further distinguished by the type of iterator they support.

A *sequence* supports a linear arrangement of single elements. ***vector***, ***list***, ***deque***, ***bitset***, and ***string*** fall into this category. *Associative containers* map values onto keys, which allows for retrieval of the values based on the keys. The STL includes the ***map***, ***multimap***, ***set***, and ***multiset*** associative containers. ***map*** and ***multimap*** store the value and the key separately, and allow for fast retrieval of the value, based upon fast retrieval of the key. ***set*** and ***multiset*** store only keys allowing fast retrieval of the key itself.

Container Requirements

Containers within the STL must meet the following requirements. Sequences and associative containers must also meet their own separate sets of requirements. The requirements for containers are:

- A container allocates all storage for the objects it holds.
- A container `X` of objects of type `T` includes the following types:

`X::value_type`	a `T`
`X::reference`	lvalue of `T`
`X::const_reference`	const lvalue of `T`
`X::iterator`	an iterator type pointing to `T`. `X::iterator` cannot be an output iterator
`X::const_iterator`	an iterator type pointing to const `T`. `X::iterator` cannot be an output iterator
`X::difference_type`	a signed integral type (must be the same as the distance type for `X::iterator` and `X::const_iterator`)
`X::size_type`	an unsigned integral type representing any non-negative value of `difference_type`
`X::allocator_type`	type of allocator used to obtain storage for elements stored in the container

- A container includes a default constructor, a copy constructor, an assignment operator, and a full complement of comparison operators (==, !=, <, >, <=, >=).
- A container includes the following member functions:

©Copyright 1999 Rogue Wave Software, Inc.

`begin()`	Returns an `iterator` or a `const_iterator` pointing to the first element in the collection
`end()`	Returns an iterator or a const_iterator pointing just beyond the last element in the collection
`swap(container)`	Swaps elements between this container and the swap's argument
`clear()`	Deletes all the elements in the container
`size()`	Returns the number of elements in the collection as a `size_type`
`max_size()`	Returns the largest possible number of elements for this type of container as a `size_type`
`empty()`	Returns `true` if the container is empty, `false` otherwise
`get_allocator()`	Returns the allocator used by this container

Reversible Containers

A container may be reversible. Essentially, a reversible container includes a reverse iterator that allows traversal of the collection in a direction opposite that of the default iterator. A reversible container must meet the following requirements in addition to those listed above:

- A reversible container includes the following types:

`X::reverse_iterator`	An iterator type pointing to `T`
`X::const_reverse_iterator`	An iterator type pointing to `const T`

- A reversible container includes the following member functions:

`rbegin()`	Returns a `reverse_iterator` or a `const_reverse_iterator` pointing past the end of the collection
`rend()`	Returns a `reverse_iterator` or a `const_reverse_iterator` pointing to the first element in the collection

Sequences

In addition to the requirements for containers, the following requirements hold for sequences:

- `iterator` and `const_iterator` must be forward iterators, bidirectional iterators or random access iterators.
- A sequence includes the following constructors:

`X(n, t)`	Constructs a container with `n` copies of `t`
`X(i, j)`	Constructs a container with elements from the range `[i,j)`

- A sequence includes the following member functions:

`insert(p,t)`	Inserts the element `t` in front of the position identified by the iterator `p`
`insert(p,n,t)`	Inserts `n` copies of `t` in front of the position identified by the iterator `p`
`insert(p,i,j)`	Inserts elements from the range `[i,j)` in front of the position identified by the iterator `p`
`erase(q)`	Erases the element pointed to by the iterator `q`
`erase(q1,q2)`	Erases the elements in the range `[q1,q2)`

- A sequence may also include the following member functions if they can be implemented with constant time complexity.

`front()`	Returns the element pointed to by `begin()`
`back()`	Returns the element pointed to by `end() - 1`
`push_front(x)`	Inserts the element `x` at `begin()`
`push_back(x)`	Inserts the element `x` at `end()`
`pop_front()`	Erases the element at `begin()`
`pop_back()`	Erases the element at `end() - 1`
`operator[](n)`	Returns the element at `a.begin() + n`
`at(n)`	Returns the element at `a.begin() + n`; throws ***out_of_range*** if `n` is invalid

Associative Containers

©Copyright 1999 Rogue Wave Software, Inc.

In addition to the requirements for a container, the following requirements hold for associative containers:

- For an associative container `iterator` and `const_iterator` must be bidirectional iterators. Associative containers are inherently sorted. Their iterators proceed through the container in the non-descending order of keys (where non-descending order is defined by the comparison object that was used to construct the container).
- An associative container includes the following types:

`X::key_type`	the type of the `Key`
`X::key_compare`	the type of the comparison to use to put the keys in order
`X::value_compare`	the type of the comparison used on values

- The default constructor and copy constructor for associative containers use the template parameter comparison class.
- An associative container includes the following additional constructors:

`X(c)`	Constructs an empty container using `c` as the comparison object
`X(i,j,c)`	Constructs a container with elements from the range `[i,j)` and the comparison object `c`
`X(i, j)`	Constructs a container with elements from the range [i,j) using the template parameter comparison object

- An associative container includes the following member functions:

`key_comp()`	Returns the comparison object used in constructing the associative container
`value_comp()`	Returns the value comparison object used in constructing the associative container
`insert(t)`	If the container does NOT support redundant key values, then this function only inserts `t` if there is no key present that is equal to the key of `t`. If the container DOES support redundant keys, then this function always inserts the element `t`. Returns a `pair<iterator,bool>`. The `bool` component of the returned pair indicates the success or failure of the operation and the `iterator` component points to the element with key equal to key of `t`.
`insert(p,t)`	If the container does NOT support redundant key values, then this function only inserts `t` if there is no key present that is equal to the key of `t`. If the container DOES support redundant keys, then this function always inserts the element `t`. The iterator `p` serves as a hint of where to start searching, allowing for some optimization of the insertion. It does not restrict the algorithm from inserting ahead of that location if necessary.
`insert(i,j)`	Inserts elements from the range `[i,j)`. A prerequisite is that `i` and `j` cannot be iterators into the container.
`erase(k)`	Erases all elements with key equal to `k`. Returns number of erased elements.
`erase(q)`	Erases the element pointed to by `q`
`erase(q1,q2)`	Erases the elements in the range `[q1,q2)`
`find(k)`	Returns an iterator pointing to an element with key equal to `k` or `end()`, if such an element is not found
`count(k)`	Returns the number of elements with key equal to `k`
`lower_bound(k)`	Returns an iterator pointing to the first element with a key greater than or equal to `k`
`upper_bound(k)`	Returns an iterator pointing to the first element with a key greater than `k`
`equal_range(k)`	Returns a pair of iterators such that the first element of the pair is equivalent to `lower_bound(k)` and the second element equivalent to `upper_bound(k)`

See Also

bitset, *deque*, *list*, *map*, *multimap*, *multiset*, *priority_queue*, *queue*, *set*, *stack*, *vector*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

copy, copy_backward

`Algorithm`

Summary

Copies a range of elements.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator, class OutputIterator>
 OutputIterator copy(InputIterator first,
                     InputIterator last,
                     OutputIterator result);
template <class BidirectionalIterator1,
          class BidirectionalIterator2>
 BidirectionalIterator2
  copy_backward(BidirectionalIterator1 first,
                BidirectionalIterator1 last,
                BidirectionalIterator2 result);
```

Description

The ***copy*** algorithm copies values from the range specified by `[first, last)` to the range specified by `[result, result + (last - first))`. ***copy*** can be used to copy values from one container to another, or to copy values from one location in a container to another location in the *same* container, as long as `result` is not within the range `[first-last)`. ***copy*** returns `result + (last - first)`. For each non-negative integer `n < (last - first)`, ***copy*** assigns `*(first + n)` to `*(result + n)`. The result of ***copy*** is undefined if `result` is in the range `[first, last)`.

Unless `result` is an insert iterator, ***copy*** assumes that at least as many elements follow `result` as are in the range `[first, last)`.

The ***copy_backward*** algorithm copies elements in the range specified by `[first, last)` into the range specified by `[result - (last - first), result)`, starting from the end of the sequence (`last-1`) and progressing to the front (`first`). Note that ***copy_backward*** does *not* reverse the order of the elements, it simply reverses the order of transfer. ***copy_backward*** returns `result - (last - first)`. You should use ***copy_backward*** instead of ***copy*** when `last` is in the range `[result - (last - first), result)`. For each positive integer `n <= (last - first)`, ***copy_backward*** assigns `*(last - n)` to `*(result - n)`. The result of ***copy_backward*** is undefined if `result` is in the range `[first, last)`.

Unless `result` is an insert iterator, ***copy_backward*** assumes that there are at least as many elements ahead of `result` as are in the range `[first, last)`.

©Copyright 1999 Rogue Wave Software, Inc.

Complexity

Both ***copy*** and ***copy_backward*** perform exactly `last - first` assignments.

Example

```
  //
  // stdlib/examples/manual/copyex.cpp
  //
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
  int d1[4] = {1,2,3,4};
  int d2[4] = {5,6,7,8};

  // Set up three vectors
  //
  vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4), v3(d2,d2 + 4);
  //
  // Set up one empty vector
  //
  vector<int> v4;
  //
  // Copy v1 to v2
  //
  copy(v1.begin(),v1.end(),v2.begin());
  //
  // Copy backwards v1 to v3
  //
  copy_backward(v1.begin(),v1.end(),v3.end());
  //
  // Use insert iterator to copy into empty vector
  //
  copy(v1.begin(),v1.end(),back_inserter(v4));
  //
  // Copy all four to cout
  //
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;
  copy(v3.begin(),v3.end(),out);
  cout << endl;
  copy(v4.begin(),v4.end(),out);
  cout << endl;

  return 0;
}
```

Program Output

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector <int, allocator<int> >
```

instead of:

```
vector <int>
```

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

count, count_if

`Algorithm`

Summary

Count the number of elements in a container that satisfy a given condition.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template<class InputIterator, class T>
 typename iterator_traits<InputIterator>::difference_type
 count(InputIterator first, InputIterator last,
       const T& value);
template <class InputIterator, class T, class Size>
 void count(InputIterator first, InputIterator last,
            const T& value, Size& n);
template<class InputIterator, class Predicate>
 typename iterator_traits<InputIterator>::difference_type
 count_if(InputIterator first, InputIterator last,
          Predicate pred);
template <class InputIterator, class Predicate, class Size>
 void count_if(InputIterator first, InputIterator last,
               Predicate pred, Size& n);
```

Description

The ***count*** algorithm compares `value` to elements in the sequence defined by iterators `first` and `last`. The first version of ***count*** returns the number of matches. The second version increments a counting value `n` each time it finds a match. In other words, ***count*** returns (or adds to `n`) the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
*i == value
```

Type `T` must be EqualityComparable.

Complexity

The ***count_if*** algorithm lets you specify a predicate, and returns the number of times an element in the sequence satisfies the predicate (or increments `n` that number of times). That is, ***count_if*** returns (or adds to `n`) the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
pred(*i) == true.
```

©Copyright 1999 Rogue Wave Software, Inc.

Both ***count*** and ***count_if*** perform exactly `last-first` applications of the corresponding predicate.

Example

```
//
// count.cpp
//
// Does not demonstrate the partial specialization versions
// of count and count_if
//
 #include <vector>
 #include <algorithm>
 #include <iostream>
 using namespace std;

 int main()
 {
   int sequence[10] = {1,2,3,4,5,5,7,8,9,10};
   int i=0,j=0,k=0;
   //
   // Set up a vector
   //
   vector<int> v(sequence,sequence + 10);

   count(v.begin(),v.end(),5,i);  // Count fives
   count(v.begin(),v.end(),6,j);  // Count sixes
   //
   // Count all less than 8
   // I=2, j=0
   //
   count_if(v.begin(),v.end(),bind2nd(less<int>(),8),k);
   // k = 7

   cout << i << " " << j << " " << k << endl;
   return 0;
 }
```

Program Output

```
2 0 7
```

Warnings

If your compiler does not support partial specialization, then the first version of both ***count*** and ***count_if*** (the one that returns the count) is not available.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector <int, allocator<int> >
```

instead of:

```
vector <int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

cout

`Pre-defined stream`

Summary

Controls output to a stream buffer associated with the object `stdout` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern ostream cout;
ostream cout;
```

Description

The object `cout` controls output to a stream buffer associated with the object `stdout` declared in `<cstdio>`. By default the standard C and C++ streams are synchronized, but performance improvement can be achieved by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

After the object `cin` is initialized, `cin.tie()` returns `&cout`, which implies that `cin` and `cout` are synchronized.

Formatting

The formatting is done through member functions or manipulators.

Manipulators	Member functions
`showpos`	`setf(ios_base::showpos)`
`noshowpos`	`unsetf(ios_base::showpos)`
`showbase`	`setf(ios_base::showbase)`
`noshowbase`	`unsetf(ios_base::showbase)`
`uppercase`	`setf(ios_base::uppercase)`
`nouppercase`	`unsetf(ios_base::uppercase)`
`showpoint`	`setf(ios_base::showpoint)`
`noshowpoint`	`unsetf(ios_base::showpoint)`

©Copyright 1999 Rogue Wave Software, Inc.

boolalpha	setf(ios_base::boolalpha)
noboolalpha	unsetf(ios_base::boolalpha)
unitbuf	setf(ios_base::unitbuf)
nounitbuf	unsetf(ios_base::unitbuf)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)
resetiosflags (ios_base::fmtflags flag)	setf(0,flag)
setiosflags (ios_base::fmtflags flag)	setf(flag)
setbase(int base)	see above
setfill(char_type c)	fill(c)
setprecision(int n)	precision(n)
setw(int n)	width(n)
endl	
ends	
flush	flush( )

Description

showpos	Generates a + sign in non-negative generated numeric output.
showbase	Generates a prefix indicating the numeric base of generated integer output
uppercase	Replaces certain lowercase letters with their uppercase equivalents in generated output
showpoint	Generates a decimal-point character unconditionally in generated floating-point output
boolalpha	Inserts and extracts bool type in alphabetic format
unitbuf	Flushes output after each output operation
internal	Adds fill characters at a designated internal point in certain generated output. If no such point is designated, it's identical to `right`.
left	Adds fill characters on the right (final positions) of certain generated output
right	Adds fill characters on the left (initial positions) of certain generated output
dec	Converts integer input or generates integer output in decimal base
hex	Converts integer input or generates integer output in hexadecimal base
oct	Converts integer input or generates integer output in octal base
fixed	Generates floating-point output in fixed-point notation
scientific	Generates floating-point output in scientific notation
resetiosflags (ios_base::fmtflags flag)	Resets the `fmtflags` field `flag`
setiosflags	

©Copyright 1999 Rogue Wave Software, Inc.

`(ios_base::fmtflags flag)`	Sets up the flag `flag`
`setbase(int base)`	Converts integer input or generates integer output in base `base`. The parameter `base` can be 8, 10 or 16.
`setfill(char_type c)`	Sets the character used to pad (fill) an output conversion to the specified field width
`setprecision(int n)`	Sets the precision (number of digits after the decimal point) to generate on certain output conversions
`setw(int n)`	Sets the field with (number of characters) to generate on certain output conversions
`endl`	Inserts a newline character into the output sequence and flush the output buffer.
`ends`	Inserts a null character into the output sequence.
`flush`	Flush the output buffer.

Default Values

```
precision()           6
width()               0
fill()                the space character
flags()               skipws | dec
getloc()              locale::locale()
```

Example

```
//
// cout example #1
//
#include<iostream>
#include<iomanip>

void main ( )
{
  using namespace std;

  int i;
  float f;

  // read an integer and a float from stdin
  cin >> i >> f;
  // output the integer and goes at the line
  cout << i << endl;

  // output the float and goes at the line
  cout << f << endl;

  // output i in hexa
  cout << hex << i << endl;

  // output i in octal and then in decimal
  cout << oct << i << dec << i << endl;

  // output i preceded by its sign
  cout << showpos << i << endl;

  // output i in hexa
  cout << setbase(16) << i << endl;

  // output i in dec and pad to the left with character
  // @ until a width of 20
  // if you input 45 it outputs 45@@@@@@@@@@@@@@@@@@
  cout << setfill('@') << setw(20) << left << dec << i;
  cout << endl;

  // output the same result as the code just above
  // but uses member functions rather than manipulators
  cout.fill('@');
  cout.width(20);
  cout.setf(ios_base::left, ios_base::adjustfield);
  cout.setf(ios_base::dec, ios_base::basefield);
  cout << i << endl;

  // outputs f in scientific notation with
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // a precision of 10 digits
  cout << scientific << setprecision(10) << f << endl;

  // change the precision to 6 digits
  // equivalents to cout << setprecision(6);
  cout.precision(6);

  // output f and goes back to fixed notation
  cout << f << fixed << endl;

}
//
// cout example #2
//
#include <iostream>

void main ( )
{
  using namespace std;

  char p[50];

  cin.getline(p,50);

  cout << p;
}
//
// cout example #3
//
#include <iostream>
#include <fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  ifstream in("file_name.txt");

  // output the all file to stdout
  if ( in )
    cout << in.rdbuf();
  else
    {
      cout << "Error while opening the file";
      cout << endl;
    }
}
```

Warnings

Keep in mind that the manipulator `endl` flushes the stream buffer. Therefore it is recommended to use `\n' if your only intent is to go at the line. It greatly improves performance when C and C++ streams are not synchronized.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cin***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***wclog***(3C++), ***iomanip***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.1

Standards Conformance

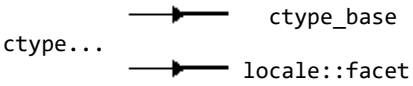
ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ctype



Summary

A facet that includes character classification facilities.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id

Member Functions

do_is()	do_widen()	
do_narrow()	is()	
do_scan_is()	narrow()	toupper()
do_scan_not()	scan_is()	widen()
do_tolower()	scan_not()	
do_toupper()	tolower()	

Synopsis

```
#include <locale>
class ctype_base;
template <class charT> class ctype;
```

Specializations

```
class ctype<char>;
```

Description

©Copyright 1999 Rogue Wave Software, Inc.

ctype<charT> is a facet that allows you to classify characters and perform simple conversions. ***ctype<charT>*** also converts upper to lower and lower to upper case, and converts between `charT` and `char`. ***ctype<charT>*** relies on `ctype_base` for a set of masks that identify the various classes of characters. These classes are:

alnum
alpha
cntrl
digit
graph
lower
print
punct
space
upper
xdigit

The masks are passed to member functions of ***ctype*** to obtain the classifications of a character or range of characters.

Interface

```
class ctype_base {
public:
  enum mask {
    space, print, cntrl, upper, lower,
    alpha, digit, punct, xdigit,
    alnum=alpha|digit, graph
  };
};

template <class charT>
class ctype : public locale::facet, public ctype_base {
  public:
    typedef charT char_type;
    explicit ctype(size_t refs = 0);
    bool         is(mask, charT) const;
    const charT* is(const charT*,
                    const charT*, mask*) const;
    const charT* scan_is(mask, const charT*,
                         const charT*) const;
    const charT* scan_not(mask, const charT*,
                          const charT*) const;
    charT        toupper(charT) const;
    const charT* toupper(charT*, const charT*) const;
    charT        tolower(charT) const;
    const charT* tolower(charT*, const charT*) const;
    charT        widen(char) const;
    const char*  widen(const char*,
                       const char*, charT*) const;
    char         narrow(charT, char) const;
    const charT* narrow(const charT*, const charT*,
                        char, char*) const;
    static locale::id id;

  protected:
     ~ctype();  // virtual
    virtual bool         do_is(mask, charT) const;
    virtual const charT* do_is(const charT*,
                               const charT*,
                               mask*) const;
    virtual const charT* do_scan_is(mask,
                                    const charT*,
                                    const charT*) const;
    virtual const charT* do_scan_not(mask,
                                     const charT*,
                                     const charT*) const;
    virtual charT        do_toupper(charT) const;
    virtual const charT* do_toupper(charT*,
                                    const charT*) const;
    virtual charT        do_tolower(charT) const;
    virtual const charT* do_tolower(charT*,
                                    const charT*) const;
    virtual charT        do_widen(char) const;
    virtual const char*  do_widen(const char*,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                                const char*,
                                charT*) const;
    virtual char         do_narrow(charT, char) const;
    virtual const charT* do_narrow(const charT*,
                                   const charT*,
                                   char, char*) const;
};
```

Types

char_type

Type of character the facet is instantiated on.

Constructors

```
explicit ctype(size_t refs = 0)
```

Construct a ***ctype*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

Destructors

```
~ctype();  // virtual and protected
```

Destroy the facet.

Public Member Functions

The public members of the ***ctype*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the public `widen` function simply calls its protected cousin `do_widen`.

```
bool
is(mask m, charT c) const;
const charT*
is(const charT* low,
   const charT* high, mask* vec) const;
```

Returns `do_is(m,c)` or `do_is(low,high,vec)`.

```
char
narrow(charT c, char dfault) const;
const charT*
narrow(const charT* low, const charT*, char dfault,
       char* to) const;
```

Returns `do_narrow(c,dfault)` or `do_narrow(low,high,dfault,to)`.

```
const charT*
scan_is(mask m, const charT*, const charT* high) const;
```

Returns `do_scan_is(m,low,high)`.

```
const charT*
scan_not(mask m, const charT* low, const charT* high) const;
```

Returns `do_scan_not(m,low,high)`.

```
charT
tolower(charT c) const;
const charT*
tolower(charT* low, const charT* high) const;
```

Returns `do_tolower(c)` or `do_tolower(low,high)`.

```
charT
toupper(charT) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
const charT*
toupper(charT* low, const charT* high) const;
```

Returns `do_toupper(c)` or `do_toupper(low,high)`.

```
charT
widen(char c) const;
const char*
widen(const char* low, const char* high, charT* to) const;
```

Returns `do_widen(c)` or `do_widen(low,high,to)`.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Protected Member Functions

```
virtual bool
do_is(mask m, charT c) const;
```

Returns `true` if `c` matches the classification indicated by the mask `m`, where `m` is one of the values available from ***ctype_base***. For instance, the following call returns `true` since `` `a' `` is an alphabetic character:

```
ctype<char>().is(ctype_base::alpha,'a');
```

See ***ctype_base*** for a description of the masks.

```
virtual const charT*
do_is(const charT* low, const charT* high,
      mask* vec) const;
```

Fills `vec` with every mask from ***ctype_base*** that applies to the range of characters indicated by `[low,high)`. See ***ctype_base*** for a description of the masks. For instance, after the following call, the first five entries in the array `v` would each contain the `mask` value: `alpha|lower|print|alnum|graph`:

```
char a[] = "abcde";
ctype_base::mask v[12];
ctype<char>().is(a,a+5,v);
```

Returns `high`.

```
virtual char
do_narrow(charT, char dfault) const;
```

Returns the appropriate `char` representation for `c`, if such exists. Otherwise `do_narrow` returns `dfault`.

```
virtual const charT*
do_narrow(const charT* low, const charT* high,
          char dfault, char* dest) const;
```

Converts each character in the range `[low,high)` to its `char` representation, if such exists. If a `char` representation is not available, then the character is converted to `dfault`. Returns `high`.

```
virtual const charT*
do_scan_is(mask m, const charT* low, const charT* high) const;
```

Finds the first character in the range `[low,high)` that matches the classification indicated by the mask `m`.

```
virtual const charT*
do_scan_not(mask m, const charT* low, const charT* high) const;
```

Finds the first character in the range `[low,high)` that does not match the classification indicated by the mask `m`.

```
virtual charT
do_tolower(charT) const;
```

Returns the lower case representation of `c`, if such exists. Otherwise returns `c`.

©Copyright 1999 Rogue Wave Software, Inc.

```
virtual const charT*
do_tolower(charT* low, const charT* high) const;
```

Converts each character in the range `[low,high)` to its lower case representation, if such exists. If a lower case representation does not exist, then the character is not changed. Returns `high`.

```
virtual charT
do_toupper(charT c) const;
```

Returns the upper case representation of `c`, if such exists. Otherwise returns `c`.

```
virtual const charT*
do_toupper(charT* low, const charT* high) const;
```

Converts each character in the range `[low,high)` to its upper case representation, if such exists. If an upper case representation does not exist, then the character is not changed. Returns `high`.

```
virtual charT
do_widen(char c) const;
```

Returns the appropriate `charT` representation for `c`.

```
virtual const char*
do_widen(const char* low, const char* high, charT* dest) const;
```

Converts each character in the range `[low,high)` to its `charT` representation. Returns `high`.

Example

```
//
// ctype.cpp
//

#include <iostream>

int main ()
{
  using namespace std;

  locale loc;
  string s1("blues Power");

  // Get a reference to the ctype<char> facet
  const ctype<char>& ct =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
      use_facet<ctype<char> >(loc);
#else
      use_facet(loc,(ctype<char>*)0);
#endif

  // Check the classification of the 'a' character
  cout << ct.is(ctype_base::alpha,'a') << endl;
  cout << ct.is(ctype_base::punct,'a') << endl;

  // Scan for the first upper case character
  cout << (char)*(ct.scan_is(ctype_base::upper,
                          s1.begin(),s1.end())) << endl;

  // Convert characters to upper case
  ct.toupper(s1.begin(),s1.end());
  cout << s1 << endl;

  return 0;
}
```

See Also

locale, *facets*, *collate*, *ctype<char>*, *ctype_byname*

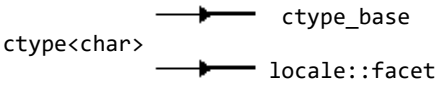
©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ctype<char>



Summary

A specialization of the ***ctype*** facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id

Member Functions

classic_table()
do_tolower()
do_toupper()
is()
narrow()
scan_is()
scan_not()
table()
tolower()
toupper()
widen()

Synopsis

```
#include <locale>
class ctype<char>;
```

Description

This specialization of the ***ctype<charT>*** template includes inline versions of ***ctype***'s member functions. The facet has the same public interface and uses the same set of masks as the ***ctype*** template.

Interface

```
template <>
class ctype<char> : public locale::facet, public ctype_base {
  public:
    typedef char char_type;
    explicit ctype(const mask* = 0, bool = false,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                  size_t = 0);
    bool         is(mask, char) const;
    const char* is(const char*,
                   const char*, mask*) const;
    const char* scan_is(mask,
                        const char*,
                        const char*) const;
    const char* scan_not(mask,
                         const char*,
                         const char*) const;
    char        toupper(char) const;
    const char* toupper(char*, const char*) const;
    char        tolower(char) const;
    const char* tolower(char*, const char*) const;
    char        widen(char) const;
    const char*  widen(const char*,
                       const char*, char*) const;
    char         narrow(char, char) const;
    const char* narrow(const char*, const char*,
                       char, char*) const;
    static locale::id id;
    static const size_t table_size = 256;

  protected:
    const mask* table() const throw();
    static const mask* classic_table() throw();

~ctype();  // virtual
    virtual char        do_toupper(char) const;
    virtual const char* do_toupper(char*,
                                   const char*) const;
    virtual char        do_tolower(char) const;
    virtual const char* do_tolower(char*,
                                   const char*) const;
};
```

Types

char_type

Type of character the facet is instantiated on.

Constructors

```
explicit ctype(const mask* tbl = 0, bool del = false,
               size_t refs = 0)
```

Construct a *ctype* facet. The three parameters set up the following conditions:

- The `tbl` argument must be either `0` or an array of at least `table_size` elements. If `tbl` is non-zero, then the supplied table is used for character classification.
- If `tbl` is non zero, and `del` is `true`, then the `tbl` array is deleted by the destructor, so the calling program need not concern itself with the lifetime of the table.
- If the `refs` argument is `0`, then destruction of the object itself is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

Destructors

```
~ctype();  // virtual and protected
```

Destroy the facet. If the constructor was called with a non-zero `tbl` argument and a `true del` argument, then the array supplied by the `tbl` argument is deleted.

Public Member Functions

The public members of the *ctype<char>* facet specialization do not all serve the same purpose as the functions in the template. In many cases these functions implement functionality, rather than just forwarding a call to a protected

©Copyright 1999 Rogue Wave Software, Inc.

implementation function.

```
static const mask*
classic_table() throw();
```

Returns a pointer to a table_size character array that represents the classifications of characters in the "C" locale.

```
bool
is(mask m, char c) const;
```

Determines if the character `c` has the classification indicated by the mask `m`. Returns `table()[(unsigned char)c]` and `m`.

```
const char*
is(const char* low,
   const char* high, mask* vec) const;
```

Fills `vec` with every mask from ***ctype_base*** that applies to the range of characters indicated by `[low,high)`. See ***ctype_base*** for a description of the masks. For instance, after the following call, the first five elements of `v` would contain: `alpha|lower|print|xdigit|graph}`:

```
char a[] = "abcde";
ctype_base::mask v[12];
ctype<char>().do_is(a,a+5,v);
```

Returns `high`.

```
char
narrow(char c, char dfault) const;
```

Returns `c`.

```
const char*
narrow(const char* low, const char*, char dfault,
       char* to) const;
```

Performs `::memcpy(to,low,high-low)`. Returns `high`.

```
const char*
scan_is(mask m, const char*, const char* high) const;
```

Finds the first character in the range `[low,high)` that matches the classification indicated by the mask `m`. The classification is matched by checking for `table()[(unsigned char) p] & m`, where `p` is in the range `[low,high)`. Returns the first `p` that matches, or `high` if none do.

```
const char*
scan_not(mask m, const char* low, const char* high) const;
```

Finds the first character in the range `[low,high)` that does not match the classification indicated by the mask `m`. The classification is matched by checking for `!(table()[(unsigned char) p] & m)`, where `p` is in the range [low,high). Returns the first `p` that matches, or `high` if none do.

```
const mask*
table() const throw();
```

If the `tbl` argument that was passed to the constructor was non-zero, then this function returns that argument. Otherwise it returns `classic_table()`.

```
char
tolower(char c) const;
const char*
tolower(char* low, const char* high) const;
```

Returns `do_tolower(c)` or `do_tolower(low,high)`.

```
char
toupper(char) const;
const char*
toupper(char* low, const char* high) const;
```

Returns `do_toupper(c)` or `do_toupper(low,high)`.

©Copyright 1999 Rogue Wave Software, Inc.

```
char
widen(char c) const;
```

Returns `c`.

```
const char*
widen(const char* low, const char* high, char* to) const;
```

Performs `::memcpy(to,low,high-low)`. Returns `high`.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Protected Member Functions

```
virtual char
do_tolower(char) const;
```

Returns the lower case representation of `c`, if such exists. Otherwise returns `c`.

```
virtual const char*
do_tolower(char* low, const char* high) const;
```

Converts each character in the range `[low,high)` to its lower case representation, if such exists. If a lower case representation does not exist, then the character is not changed. Returns `high`.

```
virtual char
do_toupper(char c) const;
```

Returns the upper case representation of `c`, if such exists. Otherwise returns `c`.

```
virtual const char*
do_toupper(char* low, const char* high) const;
```

Converts each character in the range `[low,high)` to its upper case representation, if such exists. If an upper case representation does not exist, then the character is not changed. Returns `high`.

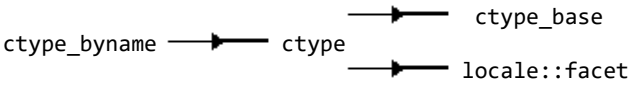
See Also

locale, *facets*, *collate*, *ctype<char>*, *ctype_byname*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

ctype_byname



Summary

A facet that includes character classification facilities based on the named locales.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT> class ctype_byname;
template <> class ctype_byname<char>;
```

Description

ctype_byname<charT> template and ***ctype_byname<char>*** specialization include the same functions as the ***ctype<charT>*** template, but specific to a particular named locale. For a description of the member functions of ***ctype_byname***, see the reference for ***ctype<charT>***. Only the constructor is described here.

Interface

```
template <class charT>
class ctype_byname : public ctype<charT> {
public:
  explicit ctype_byname(const char*, size_t = 0);
protected:
  ~ctype_byname();  // virtual
  virtual bool         do_is(mask, charT) const;
  virtual const charT* do_is(const charT*, const charT*,
                             mask*) const;
  virtual const char*  do_scan_is(mask,
                                  const charT*,
                                  const charT*) const;
  virtual const char*  do_scan_not(mask,
                                   const charT*,
                                   const charT*) const;
  virtual charT        do_toupper(charT) const;
  virtual const charT* do_toupper(charT*,
                                  const charT*) const;
  virtual charT        do_tolower(charT) const;
  virtual const charT* do_tolower(charT*,
                                  const charT*) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  virtual charT        do_widen(char) const;
  virtual const char*  do_widen(const char*, const char*,
                                charT*) const;
  virtual char         do_narrow(charT, char) const;

  virtual const charT* do_narrow(const charT*, const charT*,
                                 char, char*) const;
};

class ctype_byname<char> : public ctype<char> {
public:
  explicit ctype_byname(const char*, size_t = 0);
protected:
  ~ctype_byname();  // virtual
  virtual char        do_toupper(char) const;
  virtual const char* do_toupper(char*, const char*) const;
  virtual char        do_tolower(char) const;
  virtual const char* do_tolower(char*, const char*) const;
};
```

Constructors

```
explicit ctype_byname(const char* name, size_t refs = 0);
```

Constructs a ***ctype_byname*** facet. The facet classifies characters relative to the named locale specified by the `name` argument. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

See Also

locale, *facets*, *collate*, *ctype<char>*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

deque

`Container`

Summary

A sequence that supports random access iterators and efficient insertion/deletion at both beginning and end.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

allocator

Member Functions

assign()	erase()	operator>=()	pop_front()
at()	front()	operator<()	push_back()
back()	get_allocator()	operator<=()	push_front()
begin()	insert()	operator=()	rbegin()
clear()	max_size()	operator==()	rend()
empty()	operator!=()	operator[]()	resize()
end()	operator>()	pop_back()	size()
			swap()

Synopsis

```
#include <deque>

template <class T, class Allocator = allocator<T> >
 class deque;
```

Description

deque<T, Allocator> is a type of sequence that supports random access iterators. It supports constant time insert and erase operations at the beginning or the end of the container. Insertion and erase in the middle take linear time. Storage management is handled by the `Allocator` template parameter.

©Copyright 1999 Rogue Wave Software, Inc.

Any type used for the template parameter `T` must include the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

`Copy constructors`	T(t) and `T(u)`
`Destructor`	t.~T()
`Address of`	&t and `&u` yielding `T*` and `const T*` respectively
`Assignment`	t = a where `a` is a (possibly `const`) value of `T`

Interface

```
template <class T, class Allocator = allocator<T> >
 class deque {

public:

 // Types

   class iterator;
   class const_iterator;

   typedef T value_type;
   typedef Allocator allocator_type;
   typedef typename
           Allocator::reference        reference;
   typedef typename
           Allocator::const_reference  const_reference;
   typedef typename
           Allocator::size_type        size_type;
   typedef typename
           Allocator::difference_type  difference_type;
   typedef typename
           std::reverse_iterator<iterator> reverse_iterator;
   typedef typename
           std::reverse_iterator<const_iterator>
                                const_reverse_iterator;

 // Construct/Copy/Destroy

   explicit deque (const Allocator& = Allocator());
   explicit deque (size_type);
   deque (size_type, const T& value,
          const Allocator& = Allocator ());
   deque (const deque<T,Allocator>&);
   template <class InputIterator>
    deque (InputIterator, InputIterator,
           const Allocator& = Allocator ());
   ~deque ();
   deque<T,Allocator>& operator=
                       (const deque<T,Allocator>&);
   template <class InputIterator>
    void assign (InputIterator, InputIterator);
   void assign (size_type, const T&);
   allocator_type get allocator () const;

 // Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize (size_type, T);
   bool empty () const;

// Element access
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   reference operator[] (size_type);
   const_reference operator[] (size_type) const;
   reference at (size_type);
   const_reference at (size_type) const;
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

 // Modifiers

   void push_front (const T&);
   void push_back (const T&);
   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
   template <class InputIterator>
    void insert (iterator, InputIterator, InputIterator);

   void pop_front ();
   void pop_back ();

   iterator erase (iterator);
   iterator erase (iterator, iterator);
   void swap (deque<T, Allocator>&);
   void clear();
};

 // Non-member Operators

template <class T, class Allocator>
 bool operator== (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);

template <class T, class Allocator>
 bool operator!= (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);


template <class T, class Allocator>
 bool operator< (const deque<T, Allocator>&,
                 const deque<T, Allocator>&);

template <class T, class Allocator>
 bool operator> (const deque<T, Allocator>&,
                 const deque<T, Allocator>&);

template <class T, class Allocator>
 bool operator<= (const deque<T, Allocator>&,
                 const deque<T, Allocator>&);

template <class T, class Allocator>
 bool operator>= (const deque<T, Allocator>&,
                 const deque<T, Allocator>&);


// Specialized Algorithms

template <class T, class Allocator>
 voice swap (deque<T, Allocator>&, deque<T, Allocator>&);
```

Constructors

```
explicit
deque(const Allocator& alloc = Allocator());
```

The default constructor. Creates a deque of zero elements. The deque uses the allocator `alloc` for all storage management.

```
explicit
deque(size_type n);
```

Creates a list of length `n`, containing `n` copies of the default value for type `T`. `T` must have a default constructor. The deque uses the allocator Allocator() for all storage management.

©Copyright 1999 Rogue Wave Software, Inc.

```
deque(size_type n, const T& value,
       const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of `value`. The deque uses the allocator `alloc` for all storage management.

```
deque(const deque<T, Allocator>& x);
```

Creates a copy of `x`.

```
template <class InputIterator>
deque(InputIterator first, InputIterator last,
       const Allocator& alloc = Allocator());
```

Creates a deque of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The deque uses the allocator `alloc` for all storage management.

Destructors

```
~deque();
```

Releases any allocated memory for self.

Allocators

```
allocator
allocator_type get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator begin();
```

Returns a random access iterator that points to the first element.

```
const_iterator begin() const;
```

Returns a constant random access iterator that points to the first element.

```
iterator end();
```

Returns a random access iterator that points to the past-the-end value.

```
const_iterator end() const;
```

Returns a constant random access iterator that points to the past-the-end value.

```
reverse_iterator rbegin();
```

Returns a random access `reverse_iterator` that points to the past-the-end value.

```
const_reverse_iterator rbegin() const;
```

Returns a constant random access reverse iterator that points to the past-the-end value.

```
reverse_iterator rend();
```

Returns a random access `reverse_iterator` that points to the first element.

```
const_reverse_iterator rend() const;
```

Returns a constant random access reverse iterator that points to the first element.

Assignment Operators

```
deque<T, Allocator>&
operator=(const deque<T, Allocator>& x);
```

©Copyright 1999 Rogue Wave Software, Inc.

Erases all elements in self, then inserts into self a copy of each element in `x`. Returns a reference to self.

Reference Operators

```
reference operator[](size_type n);
```

Returns a `reference` to element `n` of self. The result can be used as an lvalue. The index `n` must be between `0` and the `size() - 1`..

```
const_reference operator[](size_type n) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between `0` and the `size() - 1`.

Member Functions

```
template <class InputIterator>
void
assign(InputIterator first, InputIterator last);
```

Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

```
void
assign(size_type n, const T& t);
```

Erases all elements contained in self, then inserts `n` instances of the value of `t`.

```
reference
at(size_type n);
```

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between `0` and the `size() - 1`.

```
const_reference
at(size_type) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between `0` and the `size() - 1`.

```
reference
back();
```

Returns a reference to the last element.

```
const_reference
back() const;
```

Returns a constant reference to the last element.

```
void
clear();
```

Erases all elements from the self.

```
bool
empty() const;
```

Returns `true` if the size of self is zero.

```
reference
front();
```

Returns a reference to the first element.

```
const_reference
front() const;
```

Returns a constant reference to the first element.

```
iterator
erase(iterator first, iterator last);
```

©Copyright 1999 Rogue Wave Software, Inc.

Deletes the elements in the range (`first, last`). Returns an iterator pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
iterator
erase(iterator position);
```

Removes the element pointed to by `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if there were no elements after the deleted range.

```
iterator
insert(iterator position, const T& x);
```

Inserts `x` before `position`. The return value points to the inserted `x`.

```
void
insert(iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` before `position`.

```
template <class InputIterator>
void
insert(iterator position, InputIterator first,
       InputIterator last);
```

Inserts copies of the elements in the range `(first, last]` before `position`.

```
size_type
max_size() const;
```

Returns `size()` of the largest possible deque.

```
void
pop_back();
```

Removes the last element. Note that this function does not return the element.

```
void
pop_front();
```

Removes the first element. Note that this function does not return the element.

```
void
push_back(const T& x);
```

Appends a copy of `x` to the end.

```
void
push_front(const T& x);
```

Inserts a copy of `x` at the front.

```
void
resize(size_type sz);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` copies of the default value of type `T` are inserted at the end of the deque. If the new size is smaller than the current capacity, then the deque is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken. Type `T` must have a default constructor.

```
void
resize(size_type sz, T c);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` `c`'s are inserted at the end of the deque. If the new size is smaller than the current capacity, then the deque is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken.

```
size_type
size() const;
```

Returns the number of elements.

©Copyright 1999 Rogue Wave Software, Inc.

```
void
swap(deque<T,Allocator>& x);
```

Exchanges self with x.

Non-member Functions

```
template <class T, class Allocator>
bool operator==(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Equality operator. Returns true if x is the same as y.

```
template <class T, class Allocator>
bool operator!=(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns true if x is not the same as y.

```
template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x,
               const deque<T, Allocator>& y);
```

Returns true if the elements contained in x are lexicographically less than the elements contained in y.

```
template <class T, class Allocator>
bool operator>(const deque<T, Allocator>& x,
               const deque<T, Allocator>& y);
```

Returns true if the elements contained in x are lexicographically greater than the elements contained in y.

```
template <class T, class Allocator>
bool operator<=(const deque<T, Allocator>& x,
               const deque<T, Allocator>& y);
```

Returns true if the elements contained in x are lexicographically less than or equal to the elements contained in y.

```
template <class T, class Allocator>
bool operator>=(const deque<T, Allocator>& x,
               const deque<T, Allocator>& y);
```

Returns true if the elements contained in x are lexicographically greater than or equal to the elements contained in y.

```
template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x,
               const deque<T, Allocator>& y);
```

Returns true if the elements contained in x are lexicographically less than the elements contained in y.

Specialized Algorithms

```
template <class T, class Allocator>
void swap(deque<T, Allocator>& a, deque<T, Allocator>& b);
```

Swaps the contents of a and b.

Example

```
//
// deque.cpp
//
 #include <deque>
 #include <string>
 #include <iostream>
 using namespace std;

 deque<string, allocator> deck_of_cards;
 deque<string, allocator> current_hand;

 void initialize_cards(deque<string, allocator>& cards) {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  cards.push_front("aceofspades");
  cards.push_front("kingofspades");
  cards.push_front("queenofspades");
  cards.push_front("jackofspades");
  cards.push_front("tenofspades");
  // etc.
}

template <class It, class It2>
void print_current_hand(It start, It2 end)
{
  while (start < end)
  cout << *start++ << endl;
}


template <class It, class It2>
void deal_cards(It, It2 end) {
  for (int i=0;i<5;i++) {
    current_hand.insert(current_hand.begin(),*end);
    deck_of_cards.erase(end++);
  }
}

void play_poker() {
  initialize_cards(deck_of_cards);
  deal_cards(current_hand.begin(),deck_of_cards.begin());
}

int main()
{
  play_poker();
  print_current_hand(current_hand.begin(),current_hand.end());
  return 0;
}
```

Program Output

```
aceofspades
kingofspades
queenofspades
jackofspades
tenofspades
```

Warnings

Member function templates are used in all containers in by the Standard Template Library. An example of this is the constructor for ***deque<T, Allocator>***, which takes two templatized iterators:

```
template <class InputIterator>
 deque (InputIterator, InputIterator);
```

deque also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a ***deque*** in the following two ways:

```
int intarray[10];
deque<int> first_deque(intarray, intarray + 10);
deque<int> second_deque(first_deque.begin(),
                        first_deque.end());
```

But not this way:

```
deque<long> long_deque(first_deque.begin(),
                                  first_deque.end());
```

since the `long_deque` and `first_deque` are not the same type.

©Copyright 1999 Rogue Wave Software, Inc.

Additionally, many compilers do not support default template arguments. If your compiler is one of these, you always need to supply the `Allocator` template argument. For instance, you have to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

distance

`Iterator Operation`

Summary

Computes the distance between two iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class ForwardIterator>
 iterator_traits<ForwardIterator>::difference_type
 distance (ForwardIterator first,
                ForwardIterator last);

template <class ForwardIterator, class Distance>
 void distance (ForwardIterator first,
                ForwardIterator last,
                Distance& n);
```

Description

The ***distance*** template function computes the distance between two iterators. The first version returns that value, while the second version increments `n` by that value. The last iterator must be reachable from the first iterator.

Note that the second version of this function is obsolete. It is included for backward compatibility and to support compilers that do not include partial specialization. The first version of the function is not available with compilers that do not support partial specialization, since it depends on `iterator_traits`, which itself depends on that particular language feature.

Example

```
//
// distance.cpp
//

 #include <iterator>
 #include <vector>
 #include <iostream>
 using namespace std;
 int main()
 {
   //
   //Initialize a vector using an array
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  //
  int arr[6] = {3,4,5,6,7,8};
  vector<int> v(arr,arr+6);
  //
  //Declare a list iterator, s.b. a ForwardIterator
  //
  vector<int>::iterator itr = v.begin()+3;
  //
  //Output the original vector
  //
  cout << "For the vector: ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  cout << "When the iterator is initialized to point to "
       << *itr << endl;
  //
  // Use of distance
  //
  vector<int>::difference_type dist = 0;
  distance(v.begin(), itr, dist);
  cout << "The distance between the beginning and itr is "
       << dist << endl;
  return 0;
}
```

Program Output

```
For the vector: 3 4 5 6 7 8
When the iterator is initialized to point to 6
The distance between the beginning and itr is 3
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector <int, allocator,int> >
```

instead of:

```
vector <int>
```

If your compiler does not support partial specialization, then you can't use the version of ***distance*** that returns the distance. Instead you have to use the version that increments a reference parameter.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Sequences, ***Random Access Iterators***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

__distance_type

`Iterator primitive`

Summary

Determines the type of distance used by an iterator. This function is now obsolete. It is retained in order to include backward compatibility and support compilers that do not include partial specialization.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class Category, class T, class Distance,
          class Pointer, class Reference>
inline Distance* __distance_type(const iterator<Category, T,
                 Distance, Pointer, Reference>&);
template <class T>
inline ptrdiff_t* __distance_type (const T*);
```

Description

The ***__distance_type*** family of function templates return a pointer to a value that is of the same type as that used to represent a distance between two iterators. The first of these take an iterator of a particular type and return a pointer to a default value of the ***difference_type*** for that iterator. The `T*` form of the function returns `ptrdiff_t*`.

Generic algorithms use this function to create local variables of the correct type. The ***__distance_type*** functions are typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,__distance_type(first));
}

template <class Iterator, class Distance>
void __foo(Iterator first, Iterator last, Distance*>
{
  Distance d = Distance();
  distance(first,last,d);
  ...
}
```

The auxiliary function template allows the algorithm to extract a distance type from the first iterator and then use that type to perform some useful work.

See Also

Other iterator primitives: ***__value_type***, ***__iterator_category***, ***distance***, ***advance***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

divides

`Function Object`

Summary

Returns the result of dividing its first argument by its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct divides;
```

Description

divides is a binary function object. Its `operator()` returns the result of dividing `x` by `y`. You can pass a ***divides*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***divides*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vecResult.begin(),
          divides<int>());
```

After this call to ***transform***, `vecResult[n]` contains `vec1[n]` divided by `vec2[n]`.

Interface

```
template <class T>
   struct divides : binary_function<T, T, T>
{
   T operator() (const T&, const T&) const;
};
```

See Also

binary_function, ***Function Objects***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

equal

`Algorithm`

Summary

Compares two ranges for equality.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2>
 bool equal(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2);

template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
 bool equal(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, BinaryPredicate
            binary_pred);
```

Description

The ***equal*** algorithm does a pairwise comparison of all of the elements in one range with all of the elements in another range to see if they match. The first version of ***equal*** uses the equal operator (==) as the comparison function, and the second version allows you to specify a binary predicate as the comparison function. The first version returns `true` if all of the corresponding elements are equal to each other. The second version of ***equal*** returns `true` if for each pair of elements in the two ranges, the result of applying the binary predicate is `true`. In other words, ***equal*** returns `true` if both of the following are `true`:

1. There are at least as many elements in the second range as in the first;
2. For every iterator `i` in the range `[first1, last1)` the following corresponding conditions hold:

   ```
   *i == *(first2 + (i - first1))
   ```

   or

   ```
   binary_pred(*i, *(first2 + (i - first1))) == true
   ```

Otherwise, ***equal*** returns `false`.

©Copyright 1999 Rogue Wave Software, Inc.

This algorithm assumes that there are at least as many elements available after `first2` as there are in the range `[first1, last1)`.

Complexity

equal performs at most `last1-first1` comparisons or applications of the predicate.

Example

```
//
// equal.cpp
//
 #include <algorithm>
 #include <vector>
 #include <functional>
 #include <iostream>
 using namespace std;

 int main()
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,2,4,3};
   //
   // Set up two vectors
   //
   vector<int> v1(d1+0, d1 + 4), v2(d2+0, d2 + 4);

   // Check for equality
   bool b1 = equal(v1.begin(),v1.end(),v2.begin());
   bool b2 = equal(v1.begin(),v1.end(),
                   v2.begin(),equal_to<int>());

   // Both b1 and b2 are false
   cout << (b1 ? "TRUE" : "FALSE")  << " "
        << (b2 ? "TRUE" : "FALSE") << endl;
   return 0;
 }
```

Program Output

```
FALSE FALSE
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

equal_range

`Algorithm`

Summary

Find the largest subrange in a collection into which a given value can be inserted without violating the ordering of the collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
 pair<ForwardIterator, ForwardIterator>
 equal_range(ForwardIterator first, ForwardIterator last,
             const T& value);

 template <class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
  equal_range(ForwardIterator first, ForwardIterator last,
              const T& value, Compare comp);
```

Description

The ***equal_range*** algorithm performs a binary search on an ordered container to determine where the element `value` can be inserted without violating the container's ordering. The library includes two versions of the algorithm. The first version uses the less than operator (`operator <`) to search for the valid insertion range, and assumes that the sequence was sorted using the less than operator. The second version allows you to specify a function object of type `Compare`, and assumes that `Compare` was the function used to sort the sequence. The function object must be a binary predicate.

equal_range returns a pair of iterators, `i` and `j`, that define a range containing elements equivalent to `value`, in other words, the first and last valid insertion points for `value`. If `value` is not an element in the container, `i` and `j` are equal. Otherwise, `i` points to the first element not "less" than `value`, and `j` points to the first element greater than value. In the second version, "less" is defined by the comparison object. Formally, ***equal_range*** returns a sub-range `[i, j)` such that `value` can be inserted at any iterator `k` within the range. Depending upon the version of the algorithm used, `k` must satisfy one of the following conditions:

```
!(*k < value) && !(value < *k)
```

or

```
comp(*k,value) == false && comp(value, *k) == false
```

©Copyright 1999 Rogue Wave Software, Inc.

The range `[first,last)` is assumed to be sorted. Type `T` must be LessThanComparable.

Complexity

equal_range performs at most `2 * log(last - first) + 1` comparisons.

Example

```
//
// eqlrange.cpp
//
 #include <vector>
 #include <algorithm>
 #include <functional>
 #include <iostream>
 using namespace std;
 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};
   //
   // Set up a vector
   //
   vector<int> v1(d1+0, d1 + 11);
   //
   // Try equal_range variants
   //
   pair<iterator,iterator> p1 =
       equal_range(v1.begin(),v1.end(),3);
   // p1 = (v1.begin() + 4,v1.begin() + 5)
   pair<iterator,iterator> p2 =
       equal_range(v1.begin(),v1.end(),2,less<int>());
   // p2 = (v1.begin() + 4,v1.begin() + 5)
   // Output results
   cout << endl  << "The equal range for 3 is: "
        << "( " << *p1.first << " , "
        << *p1.second << " ) " << endl << endl;
   cout << endl << "The equal range for 2 is: "
        << "( " << *p2.first << " , "
        << *p2.second << " ) " << endl;
   return 0;
 }
```

Program Output

```
The equal range for 3 is: ( 3 , 4 )
The equal range for 2 is: ( 2 , 3 )
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***lower_bound***, ***upper_bound***

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

equal_to

`Function Object`

Summary

A binary function object that returns `true` if its first argument equals its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct equal_to;
```

Description

equal_to is a binary function object. Its `operator()` returns `true` if `x` is equal to `y`. You can pass an ***equal_to*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***equal_to*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vecResult.begin(),
          equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` contains a `1` if `vec1(n)` was equal to `vec2(n)` or a `0` if `vec1(n)` was not equal to `vec2(n)`.

Interface

```
template <class T>
    struct equal_to : binary_function<T, T, bool>
```

See Also

binary_function, ***Function Objects***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

exception

`Standard Exception`

Summary

A class that supports logic and runtime errors.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator=()
what()

Synopsis

```
#include <exception>
class exception;
```

Description

The class ***exception*** defines the base class for the types of objects thrown as exceptions by Standard C++ Library components and some expressions. This class is used to report errors detected during program execution. Users can also use these exceptions to report errors in their own programs.

Interface

```
class exception {

   public:
     exception () throw();
     exception (const exception&) throw();
     exception& operator= (const exception&) throw();
     virtual ~exception () throw();
     virtual const char* what () const throw();
 };

 class logic_error : public exception {
   public:
     explicit logic_error (const string& what_arg);
 };

 class domain_error : public logic_error {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  public:
    explicit domain_error (const string& what_arg);
};

class invalid_argument : public logic_error {
  public:
    explicit invalid_argument (const string& what_arg);
};

class length_error : public logic_error {
  public:
    explicit length_error (const string& what_arg);
};


class out_of_range : public logic_error {
  public:
    explicit out_of_range (const string& what_arg);
};

class runtime_error : public exception {
  public:
    explicit runtime_error (const string& what_arg);
};

class range_error : public runtime_error {
  public:
    explicit range_error (const string& what_arg);
};

class overflow_error : public runtime_error {
  public:
    explicit overflow_error (const string& what_arg);
};

class underflow_error : public runtime_error {
  public:
    explicit underflow_error (const string& what_arg);
};
```

Constructors

```
exception()
throw();
```

Constructs an object of class ***exception***.

```
exception(const exception&)
throw();
```

Copies an `exception` object.

Destructors

```
virtual
~exception()
throw();
```

Destroys an object of class ***exception***.

Operators

```
exception&
operator=(const exception&)
throw();
```

Copies an `exception` object.

Member Functions

```
virtual const char*
what()const
```

©Copyright 1999 Rogue Wave Software, Inc.

```
throw();
```

Returns an implementation-defined, null-terminated byte string representing a human-readable message describing the exception. The message may be a null-terminated multibyte string, suitable for conversion and display as a `wstring`.

Constructors for Derived Classes

```
logic_error::logic_error(const string& what_arg);
```

Constructs an object of class `logic_error`.

```
domain_error::domain_error(const string& what_arg);
```

Constructs an object of class `domain_error`.

```
invalid_argument::invalid_argument(const string& what_arg);
```

Constructs an object of class `invalid_argument`.

```
length_error::length_error(const string& what_arg);
```

Constructs an object of class `length_error`.

```
out_of_range::out_of_range(const string& what_arg);
```

Constructs an object of class `out_of_range`.

```
runtime_error::runtime_error(const string& what_arg);
```

Constructs an object of class `runtime_error`.

```
range_error::range_error(const string& what_arg);
```

Constructs an object of class `range_error`.

```
overflow_error::overflow_error(const string& what_arg);
```

Constructs an object of class `overflow_error`.

```
underflow_error::underflow_error(
                    const string& what_arg);
```

Constructs an object of class `underflow_error`.

Example

```
//
// except.cpp
//
#include <iostream>
#include <stdexcept>
using namespace std;

static void f() { throw runtime_error("a runtime error"); }

int main ()
{
   //
   // By wrapping the body of main in a try-catch block
   // we can be assured that we'll catch all exceptions
   // in the exception hierarchy. You can simply catch
   // exception as is done below, or you can catch each
   // of the exceptions in which you have an interest.
   //
   try
   {
       f();
   }
   catch (const exception& e)
   {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        cout << "Got an exception: " << e.what() << endl;
    }
    return 0;
}
```

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

facets

`Localization classes`

Summary

A family of classes used to encapsulate categories of locale functionality.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

The Standard C++ Localization library includes a locale interface that contains a collection of diverse facets. Each facet has localization facilities for some specific area, such as character classification or numeric formatting. Each facet also falls into one or more broad categories. These categories are defined in the locale class, and the standard facets fit into these categories as follows.

Category	Facets
`collate`	`collate, collate_byname`
`ctype`	`ctype, codecvt, ctype_byname, codecvt_byname`
`monetary`	`moneypunct, moneypunct_byname, money_put, money_get`
`numeric`	`numpunct, numpunct_byname, num_put, num_get`
`time`	`time_put, time_put_byname, time_get, time_get_byname`
`messages`	`messages, messages_byname`

A facet must satisfy two properties. First, it must be derived from the base class `locale::facet`, either directly or indirectly (for example, `facet -> ctype<char> -> my_ctype`). Second, it must contain a member of type `locale::id`. This ensures that the locale class can manage its collection of facets properly.

See Also

locale, specific facet reference sections

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

fill, fill_n

`Algorithm`

Summary

Initializes a range with a given value.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last,
            const T& value);

template <class OutputIterator, class Size, class T>
  void fill_n(OutputIterator first, Size n, const T& value);
```

Description

The ***fill*** and ***fill_n*** algorithms are used to assign a value to the elements in a sequence. ***fill*** assigns the value to all the elements designated by iterators in the range `[first, last)`.

The ***fill_n*** algorithm assigns the value to all the elements designated by iterators in the range `[first, first + n)`. ***fill_n*** assumes that there are at least `n` elements following `first`, unless `first` is an insert iterator.

Type `T` must be Assignable, and `Size` must be convertible to an integral type.

Complexity

fill makes exactly `last - first` assignments, and ***fill_n*** makes exactly `n` assignments.

Example

```
//
// fill.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
{
  int d1[4] = {1,2,3,4};
  //
  // Set up two vectors
  //
  vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
  //
  // Set up one empty vector
  //
  vector<int> v3;
  //
  // Fill all of v1 with 9
  //
  fill(v1.begin(),v1.end(),9);


  //
  // Fill first 3 of v2 with 7
  //
  fill_n(v2.begin(),3,7);

  //
  // Use insert iterator to fill v3 with 5 11's
  //
  fill_n(back_inserter(v3),5,11);
  //
  // Copy all three to cout
  //
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;
  copy(v3.begin(),v3.end(),out);
  cout << endl;
  //
  // Fill cout with 3 5's
  //
  fill_n(ostream_iterator<int,char>(cout," "),3,5);
  cout << endl;

  return 0;
}
```

Program Output

```
9 9 9 9
7 7 7 4
11 11 11 11 11
5 5 5
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

find

`Algorithm`

Summary

Finds an occurrence of value in a sequence.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator, class T>
  InputIterator find(InputIterator first,
                     InputIterator last,
                     const T& value);
```

Description

The ***find*** algorithm lets you search for the first occurrence of a particular value in a sequence. ***find*** returns the first iterator `i` in the range `[first, last)` for which the following condition holds:

```
*i == value.
```

If ***find*** does not find a match for `value`, it returns the iterator `last`.

Type `T` must be EqualityComparable.

Complexity

find performs at most `last-first` comparisons.

Example

```
//
// find.cpp
//
 #include <vector>
 #include <algorithm>
 #include <functional>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
int main()
{
  typedef vector<int>::iterator iterator;
  int d1[10] = {0,1,2,2,3,4,2,2,6,7};

  // Set up a vector
  vector<int> v1(d1,d1 + 10);

  // Try find
  iterator it1 = find(v1.begin(),v1.end(),3);
  // it1 = v1.begin() + 4;

  // Try find_if
  iterator it2 =
  find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
  // it2 = v1.begin() + 4

  // Try both adjacent_find variants
  iterator it3 = adjacent_find(v1.begin(),v1.end());
  // it3 = v1.begin() +2


  iterator it4 =
     adjacent_find(v1.begin(),v1.end(),equal_to<int>());
  // v4 = v1.begin() + 2

  // Output results
  cout << *it1 << " " << *it2 << " " << *it3 << " "
       << *it4 << endl;

  return 0;
}
```

Program Output

```
3 3 2 2
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

adjacent_find, *find_first_of*, *find_if*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

find_end

`Algorithm`

Summary

Finds the last occurrence of a sub-sequence in a sequence.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1,
                          ForwardIterator1 last1,
                          ForwardIterator2 first2,
                          ForwardIterator2 last2);
template <class Forward Iterator1, class ForwardIterator2,
          class BinaryPredicate>
  ForwardIterator1 find_end(ForwardIterator1 first1,
                            ForwardIterator1 last1,
                            ForwardIterator2 first2,
                            ForwardIterator2 last2,
                            BinaryPredicate pred);
```

Description

The ***find_end*** algorithm finds the last occurrence of a sub-sequence, indicated by `[first2, last2)`, in a sequence, `[first1,last1)`. The algorithm returns an iterator pointing to the first element of the found sub-sequence, or `last1` if no match is found.

More precisely, the ***find_end*** algorithm returns the last iterator `i` in the range `[first1, last1 - (last2-first2))` such that for any non-negative integer `n < (last2-first2)`, the following corresponding conditions hold:

```
*(i+n)  ==  *(first2+n),
pred(*(i+n),*(first2+n)) == true.
```

Or returns `last1` if no such iterator is found.

Two versions of the algorithm exist. The first uses the equality operator as the default binary predicate, and the second allows you to specify a binary predicate.

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

At most `(last2-first2)*(last1-first1-(last2-first2)+1)` applications of the corresponding predicate are done.

Example

```
//
// find_end.cpp
//
#include<vector>
#include<iterator>
#include<algorithm>
#include<functional>
#include<iostream>
using namespace std;

int main()
{
   typedef vector<int>::iterator iterator;
   int d1[10] = {0,1,6,5,3,2,2,6,5,7};
   int d2[4] = {6,5,0,0}
   //
   // Set up two vectors.
   //
   vector<int> v1(d1+0, d1+10), v2(d2+0, d2+2);
   //
   // Try both find_first_of variants.
   //
   iterator it1 = find_first_of (v1.begin(), v1.end(),
                              v2.begin(), v2.end());
   iterator it2 = find_first_of (v1.begin(), v1.end(),
                              v2.begin(),
                                  v2.end(), equal_to<int>());
   //
   // Try both find_end variants.
   //
   iterator it3 = find_end (v1.begin(), v1.end(),
                            v2.begin(), v2.end());
   iterator it4 = find_end (v1.begin(), v1.end(),
                            v2.begin(),
                            v2.end(), equal_to<int>());
   //
   // Output results of find_first_of.
    // Iterator now points to the first element that matches
   // one of a set of values
   //
   cout << "For the vectors: ";
   copy (v1.begin(), v1.end(),
         ostream_iterator<int>(cout," "));
   cout << " and ";
   copy (v2.begin(), v2.end(),
         ostream_iterator<int>(cout," "));
   cout<< endl ,, endl
       << "both versions of find_first_of point to: "
       << *it1 << endl;

   //
   //Output results of find_end.
   //Iterator now points to the first element of the last
   //find sub-sequence.
   //
   cout << endl << endl
        << "both versions of find_end point to: "
        << *it3 << endl;

   return 0;
}
```

Program Output

```
For the vectors: 0 1 6 5 3 2 2 6 5 7  and 6 5
both versions of find_first_of point to: 6
both versions of find_end point to: 6
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Algorithms, *find*, *find_if*, *adjacent_find*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

find_first_of

`Algorithm`

Summary

Finds the first occurrence of any value from one sequence in another sequence.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of (ForwardIterator1 first1,
                                ForwardIterator1 last1,
                                ForwardIterator2 first2,
                                ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
ForwardIterator1 find_first_of (ForwardIterator1 first1,
                                ForwardIterator1 last1,
                                ForwardIterator2 first2,
                                ForwardIterator2 last2,
                                BinaryPredicate pred);
```

Description

The ***find_first_of*** algorithm finds the first occurrence of a value from a sequence, specified by `first2, last2`, in a sequence specified by `first1, last1`. The algorithm returns an iterator in the range `[first1, last1)` that points to the first matching element. If the first sequence `[first1, last1)` does not contain any of the values in the second sequence, ***find_first_of*** returns `last1`.

In other words, ***find_first_of*** returns the first iterator `i` in the `[first1, last1)`such that for some integer `j` in the range `[first2, last2)`, the following conditions hold:

`*i == *j, pred(*i,*j) == true.`

Or ***find_first_of*** returns `last1` if no such iterator is found.

Two versions of the algorithm exist. The first uses the equality operator as the default binary predicate, and the second allows you to specify a binary predicate.

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

At most `(last1 - first1)*(last2 - first2)` applications of the corresponding predicate are done.

Example

```
//
// find_f_o.cpp
//
 #include <vector>
 #include <algorithm>
 #include <functional>
 #include <iostream>
 using namespace std;

 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[10] = {0,1,2,2,3,4,2,2,6,7};
   int d2[2] = {6,4};
   //
   // Set up two vectors
   //
   vector<int> v1(d1,d1 + 10), v2(d2,d2 + 2);
   //
   // Try both find_first_of variants
   //
   iterator it1 =
     find_first_of(v1.begin(),v1.end(),v2.begin(),v2.end());
   find_first_of(v1.begin(),v1.end(),v2.begin(),v2.end(),
                 equal_to<int>());
   //
   // Output results
   //
   cout << "For the vectors: ";
   copy(v1.begin(),v1.end(),
        ostream_iterator<int,char>(cout," " ));
   cout << " and ";
   copy(v2.begin(),v2.end(),
        ostream_iterator<int,char>(cout," " ));
   cout << endl << endl
       << "both versions of find_first_of point to: "
       << *it1;

   return 0;
 }
```

Program Output

```
For the vectors: 0 1 2 2 3 4 2 2 6 7  and 6 4
both versions of find_first_of point to: 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Algorithms, *adjacent_find*, *find*, *find_if*, *find_end*

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

find_if

`Algorithm`

Summary

Finds an occurrence of a value in a sequence that satisfies a specified predicate.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first,
                        InputIterator last,
                        Predicate pred);
```

Description

The ***find_if*** algorithm allows you to search for the first element in a sequence that satisfies a particular condition. The sequence is defined by iterators `first` and `last`, while the condition is defined by the third argument: a predicate function that returns a boolean value. ***find_if*** returns the first iterator `i` in the range `[first, last)` for which the following condition holds:

```
pred(*i) == true.
```

If no such iterator is found, ***find_if*** returns `last`.

Complexity

find_if performs at most `last-first` applications of the corresponding predicate.

Example

```
/
// find.cpp
//
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[10] = {0,1,2,2,3,4,2,2,6,7};

   // Set up a vector
   vector<int> v1(d1,d1 + 10);

   // Try find
   iterator it1 = find(v1.begin(),v1.end(),3);
   // it1 = v1.begin() + 4;

   // Try find_if
   iterator it2 =
    find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
   // it2 = v1.begin() + 4

   // Try both adjacent_find variants
   iterator it3 = adjacent_find(v1.begin(),v1.end());
   // it3 = v1.begin() +2

   iterator it4 =
      adjacent_find(v1.begin(),v1.end(),equal_to<int>());
   // v4 = v1.begin() + 2

   // Output results
   cout << *it1 << " " << *it2 << " " << *it3 << " "
        << *it4 << endl;

   return 0;
 }
```

Program Output

```
3 3 2 2
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

adjacent_find, *Algorithms*, *find*,*find_end*, *find_first_of*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

for_each

`Algorithm`

Summary

Applies a function to each element in a range.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator, class Function>
 void for_each(InputIterator first, InputIterator last,
               Function f);
```

Description

The ***for_each*** algorithm applies function `f` to all members of the sequence in the range `[first, last)`, where `first` and `last` are iterators that define the sequence. Since this a non-mutating algorithm, the function `f` cannot make any modifications to the sequence, but it can achieve results through side effects (such as copying or printing). If `f` returns a result, the result is ignored.

Complexity

The function `f` is applied exactly `last - first` times.

Example

```
//
// for_each.cpp
//
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

 // Function class that outputs its argument times x
 template <class Arg>
 class out_times_x :  private unary_function<Arg,void>
 {
   private:
      Arg multiplier;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  public:
     out_times_x(const Arg& x) : multiplier(x) { }
     void operator()(const Arg& x)
        { cout << x * multiplier << " " << endl; }
};

int main()
{
  int sequence[5] = {1,2,3,4,5};


  // Set up a vector
  vector<int> v(sequence,sequence + 5);


  // Setup a function object
  out_times_x<int> f2(2);

  for_each(v.begin(),v.end(),f2);   // Apply function

  return 0;
}
```

Program Output

```
2
4
6
8
10
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Algorithms, *Function Objects*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Forward Iterators

`Iterator`

Summary

A forward-moving iterator that can both read and write.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

NOTE:For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Forward iterators are forward moving, and have the ability to both read and write data. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a and b`	values of type `X`
`n`	value representing a `distance` between two iterators
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Forward Iterators

The following expressions must be valid for forward iterators:

`X u`	u might have a singular value
`X()`	X() might be singular
`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++`	returns `T&`

©Copyright 1999 Rogue Wave Software, Inc.

Forward iterators have the condition that `a == b` implies `*a == *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

See Also

Iterators, *Bidirectional Iterators*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

fpos

Summary

Maintains position information fort he ***iostream*** classes.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

state_type

Member Functions

good()
long()
offset()
pos()
state()

Synopsis

```
#include <rw/iotraits>
template<class stateT = mbstate_t>
class fpos
```

Description

The template class ***fpos<stateT>*** is used by the ***iostream*** classes to maintain position information. It maintains three kinds of information: the absolute position, the conversion state and the validity of the stored position. Streams instantiated on tiny characters use ***streampos*** as their positioning type, whereas streams instantiated on wide characters use ***wstreampos***, but both are defined as ***fpos<mbstate_t>***.

Interface

```
template <class stateT = mbstate_t>
class fpos {

 public:

  typedef stateT  state_type;

  fpos(long off = 0);
  fpos(state_type);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  bool good();
  operator long();

  long offset() cont;
  long offset(long);

  state_type   state(state_type);
  state_type   state () const;

  long pos() const;
  long pos(long);

};
```

Types

state_type

The type `state_type` holds the conversion state, and is compatible with the function `locale::codecvt()`. By default it is defined as `mbstate_t`.

Public Constructors

`fpos(long off =0);`

Constructs an `fpos` object, initializing its position with `off` and its conversion state with the default `stateT` constructor. This function is not described in the C++ standard.

`fpos(state_type st);`

Constructs an `fpos` object, initializing its conversion state with `st`, its position with the start position, and its status to `good`.

Public Member Functions

```
state_type
state() const;
```

Returns the conversion state stored in the `fpos` object.

```
state_type
state(state_type st);
```

Stores `st` as the new conversion state in the `fpos` object and returns its previous value.

`bool good();`

Returns the status of the `fpos` object. Offset(`-1`) indicates an invalid value and returns `false`.

`operator long();`

Converts `fpos` object to absolute displacement. This operator is handy when used with functions such as `tellp()` or `tellg()` to get absolute displacement.

```
long
offset() const;
```

Returns the signed displacement in the fpos object.

```
long
offset(long off);
```

Stores `off` as the new signed displacement in the `fpos` object and returns its previous value.

```
long
pos() const;
```

Returns the absolute position in the fpos object.

©Copyright 1999 Rogue Wave Software, Inc.

```
long
offset(long pos);
```

Stores `pos` as the new absolute position in the `fpos` object and returns its previous value.

Valid Operations

In the following,

- `P` refers to type `fpos<stateT>`
- `p` and `q` refer to a value of type `fpos<stateT>`
- `O` refers to the offset type (`streamoff`, `wstreamoff`, `long`)
- `o` refers to a value of the offset type
- `i` refers to a value of type `int`

Valid operations:

`P p( I );`	Constructs from `int`
`P p = i;`	Assigns from `int`
`P( o )`	Converts from offset
`O( p )`	Converts to offset
`p == q`	Tests for equality
`p != q`	Tests for inequality
`q = p + o`	Adds offset
`p += o`	Adds offset
`q = p -o`	Subtracts offset
`q -= o`	Subtracts offset
`o = p - q`	Returns offset

See Also

iosfwd(3C++), ***char_traits***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.4.

Amendment 1 to the C Standard.

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

front_insert_iterator, front_inserter

`Insert Iterator`

Summary

An insert iterator used to insert items at the beginning of a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

container_type

Member Functions

front_inserter()
operator*()
operator++()
operator=()

Synopsis

```
#include <iterator>
template <class Container>
class front_insert_iterator ;
```

Description

Insert iterators let you *insert* new elements into a collection rather than copy a new element's value over the value of an existing element. The class ***front_insert_iterator*** is used to insert items at the beginning of a collection. The function `front_inserter` creates an instance of a ***front_insert_iterator*** for a particular collection type. A ***front_insert_iterator*** can be used with ***deque***s and ***list***s, but not with ***map***s or ***set***s.

Note that a ***front_insert_iterator*** makes each element that it inserts the new front of the container. This has the effect of reversing the order of the inserted elements. For example, if you use a ***front_insert_iterator*** to insert "1" then "2" then "3" onto the front of container `exmpl`, you find, after the three insertions, that the first three elements of `exmpl` are "3 2 1".

Interface

```
template <class Container>
 class front_insert_iterator : public
       iterator<output_iterator_tag,void,void,void,void> {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 protected:
   Container* container;
 public:
   typedef Container container_type;
   explicit front_insert_iterator (Container&);
   front_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   front_insert_iterator<Container>& operator* ();
   front_insert_iterator<Container>& operator++ ();
   front_insert_iterator<Container> operator++ (int);
};

 template <class Container>
  front_insert_iterator<Container>
                        front_inserter (Container&);
```

Types

container_type

The type of container acted on by this iterator.

Constructors

```
explicit
front_insert_iterator(Container& x);
```

Creates an instance of a ***front_insert_iterator*** associated with container `x`.

Operators

```
front_insert_iterator<Container>&
operator=(const typename Container::value_type& value);
```

Inserts a copy of `value` on the front of the container, and returns `*this`.

```
front_insert_iterator<Container>&
operator*();
```

Returns `*this` (the input iterator itself).

```
front_insert_iterator<Container>&
operator++();
front_insert_iterator<Container>
operator++(int);
```

Increments the insert iterator and returns `*this`.

Non-member Functions

```
template <class Container>
front_insert_iterator<Container>
front_inserter(Container& x)
```

Returns a ***front_insert_iterator*** that inserts elements at the beginning of container `x`. This function allows you to create front insert iterators inline.

Example

```
//
// ins_itr.cpp
//
#include <iterator>
#include <deque>
#include <iostream>
using namespace std;

 int main ()
 {
   //
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // Initialize a deque using an array.
   //
   int arr[4] = { 3,4,7,8 };
   deque<int> d(arr+0, arr+4);
   //
   // Output the original deque.
   //
   cout << "Start with a deque: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int>(cout," "));
   //
   // Insert into the middle.
   //
   insert_iterator<deque<int> > ins(d, d.begin()+2);
   *ins = 5; *ins = 6;
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use an insert_iterator: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int>(cout," "));
   //
   // A deque of four 1s.
   //
   deque<int> d2(4, 1);
   //
   // Insert d2 at front of d.
   //
   copy(d2.begin(), d2.end(), front_inserter(d));
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use a front_inserter: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int>(cout," "));
   //
   // Insert d2 at back of d.
   //
   copy(d2.begin(), d2.end(), back_inserter(d));
   //
   // Output the new deque.
   //
   cout << endl << endl;
   cout << "Use a back_inserter: " << endl << "     ";
   copy(d.begin(), d.end(),
        ostream_iterator<int>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
Start with a deque:
     3 4 7 8
Use an insert_iterator:
     3 4 5 6 7 8
Use a front_inserter:
     1 1 1 1 3 4 5 6 7 8
Use a back_inserter:
     1 1 1 1 3 4 5 6 7 8 1 1 1 1
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Insert Iterators

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Function Objects

Summary

Function objects are objects with an `operator()` defined. They are used as arguments to templatized algorithms, in place of pointers to functions.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
```

Description

Function objects are objects with an `operator()` defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an `operator()` defined, or a pointer to a function. The Standard C++ Library includes both a standard set of function objects, and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take one argument are called *unary function objects.* Unary function objects must include the typedefs `argument_type` and `result_type`. Similarly, function objects that take two arguments are called *binary function objects* and, as such, must include the typedefs `first_argument_type`, `second_argument_type`, and `result_type`.

The classes `unary_function` and `binary_function` make the task of creating templatized function objects easier. The necessary typedefs for a unary or binary function object are included by inheriting from the appropriate function object class.

The function objects in the standard library are listed below, together with a brief description of their operation. This class reference also includes an alphabetic entry for each function.

Name	Operation
`arithmetic functions`	
`plus`	`addition x + y`
`minus`	`subtraction x - y`
`multiplies`	`multiplication x * y`
`divides`	`division x / y`
`modulus`	`remainder x % y`
`negate`	`negation - x`
`comparison functions`	

©Copyright 1999 Rogue Wave Software, Inc.

```
equal_to      equality test x == y
not_equal_to  inequality test x != y
greater       greater comparison x > y
less          less-than comparison x < y
greater_equal greater than or equal comparison x >= y
less_equal    less than or equal comparison x <= y
logical functions
logical_and   logical conjunction x && y
logical_or    logical disjunction x || y
logical_not   logical negation ! x
```

Interface

```
    template <class Arg, class Result>
    struct unary_function{
         typedef Arg argument_type;
         typedef Result result_type;
    };

    template <class Arg1, class Arg2, class Result>
    struct binary_function{
         typedef Arg1 first_argument_type;
         typedef Arg2 second_argument_type;
         typedef Result result_type;
    };

 // Arithmetic Operations

   template<class T>
   struct plus : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct minus : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct multiplies : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct divides : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct modulus : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct negate : unary_function<T, T> {
        T operator() (const T&) const;
};

 // Comparisons

template <class T>
struct equal_to : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct not_equal_to : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 struct greater : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 template <class T>
 struct less : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 template <class T>
 struct greater_equal : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 template <class T>
 struct less_equal : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 // Logical Comparisons

 template <class T>
 struct logical_and : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 template <class T>
 struct logical_or : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};

 template <class T>
 struct logical_not : unary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};
```

Example

```
//
// funct_ob.cpp
//
 #include<functional>
 #include<deque>
 #include<vector>
 #include<algorithm>
 #include <iostream>
 using namespace std;

 //Create a new function object from unary_function
 template<class Arg>
 class factorial : public unary_function<Arg, Arg>
 {
   public:

   Arg operator()(const Arg& arg)
   {
     Arg a = 1;
     for(Arg i = 2; i <= arg; i++)
       a *= i;
     return a;
   }
 };

 int main()
 {
   //Initialize a deque with an array of ints
   int init[7] = {1,2,3,4,5,6,7};
   deque<int> d(init, init+7);

   //Create an empty vector to store the factorials
   vector<int> v((size_t)7);

   //Transform the numbers in the deque to their factorials
   //and store in the vector
   transform(d.begin(), d.end(), v.begin(),
             factorial<int>());
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   //Print the results
   cout << "The following numbers: " << endl << "     ";
   copy(d.begin(),d.end(),
        ostream_iterator<int,char>(cout," "));

   cout << endl << endl;
   cout << "Have the factorials: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));

   return 0;
 }
```

Program Output

```
The following numbers:
     1 2 3 4 5 6 7
Have the factorials:
     1 2 6 24 120 720 5040
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> > and deque<int, allocator<int> >
```

instead of:

```
vector<int> and deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***unary_function***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

generate, generate_n

`Algorithm`

Summary

Initialize a container with values produced by a value-generator class.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last,
                Generator gen);

template <class OutputIterator, class Size, class Generator>
  void generate_n(OutputIterator first, Size n, Generator gen);
```

Description

A value-generator function returns a value each time it is invoked. The algorithms ***generate*** and ***generate_n*** initialize (or reinitialize) a sequence by assigning the return value of the generator function `gen` to all the elements designated by iterators in the range `[first, last)` or `[first, first + n)`. The function `gen` takes no arguments. (`gen` can be a function or a class with an `operator ()` defined that takes no arguments.)

generate_n assumes that there are at least `n` elements following `first`, unless `first` is an insert iterator.

Complexity

The ***generate*** and ***generate_n*** algorithms invoke `gen` and assign its return value exactly `last - first` (or `n`) times.

Example

```
//
// generate.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// Value generator simply doubles the current value
// and returns it
template <class T>
class generate_val
{
  private:
     T val_;
  public:
     generate_val(const T& val) : val_(val) {}
     T& operator()() { val_ += val_; return val_; }
};

int main()
{
  int d1[4] = {1,2,3,4};
  generate_val<int> gen(1);

  // Set up two vectors
  vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
  // Set up one empty vector
  vector<int> v3;

  // Generate values for all of v1
  generate(v1.begin(),v1.end(),gen);

  // Generate values for first 3 of v2
  generate_n(v2.begin(),3,gen);

  // Use insert iterator to generate 5 values for v3
  generate_n(back_inserter(v3),5,gen);

  // Copy all three to cout
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;
  copy(v3.begin(),v3.end(),out);
  cout << endl;

  // Generate 3 values for cout
  generate_n(ostream_iterator<int>(cout," "),3,gen);
  cout << endl;

  return 0;
}
```

Program Output

```
2 4 8 16
2 4 8 4
2 4 8 16 32
2 4 8
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

get_temporary_buffer

`Memory Handling Primitive`

Summary

Pointer based primitive for handling memory

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <memory>
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer (ptrdiff_t);
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer (ptrdiff_t, T*);
```

Description

The ***get_temporary_buffer*** templatized function reserves from system memory the largest possible buffer that is less than or equal to the size requested (`n*sizeof(T)`), and returns a `pair<T*, ptrdiff_t>` containing the address and size of that buffer. The units used to describe the capacity are in `sizeof(T)`.

The first version of this function is not available if your compiler does not support functions templatized on return type only.

See Also

allocator, ***pair***, ***return_temporary_buffer***.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

greater

`Function Object`

Summary

A binary function object that returns `true` if its first argument is greater than its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct greater : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

Description

greater is a binary function object. Its `operator()` returns `true` if `x` is greater than `y`. You can pass a ***greater*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***greater*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vecResult.begin(), greater<int>());
```

Warnings

After this call to ***transform***, `vecResult(n)` contains a `1` if `vec1(n)` was greater than `vec2(n)` or a `0` if `vec1(n)` was less than or equal to `vec2(n)`.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

greater_equal

`Function Object`

Summary

A binary function object that returns `true` if its first argument is greater than or equal to its second

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct greater_equal ; : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

Description

greater_equal is a binary function object. Its `operator()` returns `true` if `x` is greater than or equal to `y`. You can pass a ***greater_equal*** object to any algorithm that requires a binary function. For example, the ***sort*** algorithm can accept a binary function as an alternate comparison object to sort a sequence. ***greater_equal*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
.
.
sort(vec1.begin(), vec1.end(),greater_equal<int>());
```

After this call to ***sort***, `vec1` is sorted in descending order.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

©Copyright 1999 Rogue Wave Software, Inc.

Function Objects

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

gslice

`Valarray helpers`

Summary

A numeric array class used to represent a generalized slice from an array.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

size()
start()
stride()

Synopsis

```
#include <valarray>
class gslice ;
```

Description

gslice represents a generalized slice from an array. A generalized slice contains a starting index, a set of lengths and a set of strides. The number of lengths and strides must be equal. Together the lengths and strides allow a slice from a multiple dimension array (with the dimension equal to the number of strides) to be represented on a one dimensional ***valarray***. The ***gslice*** maps a set of `n` indices `(ij)`, where `n` is equal to the number of strides, to a single index `k`.

When applied to a ***valarray*** using the ***gslice*** subscript operator (see ***valarray***) a ***gslice*** produces a ***gslice_array***. The ***gslice_array*** class creates a view into the original ***valarray*** that is tailored to match parameters of the ***gslice***. The elements in a ***gslice_array*** are references to the elements in the original array.

Interface

```
class gslice {
public:
  // constructors
  gslice();
  gslice(size_t, const valarray<size_t>&,
         const valarray<size_t>&);
  gslice (const gslice&);

  // Accessors
  size_t start() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  valarray<size_t> size() const;
  valarray<size_t> stride() const;
};
```

Constructors

gslice();

Default constructor creates a gslice specifying no elements.

```
gslice(size_t start, const valarray<size_t>& length,
       const valarray<size_t>& stride);
```

Creates a slice with starting index, length and stride as indicated by the arguments.

gslice(const gslice&)

Creates a slice with starting index, length and stride as indicated by the slice argument.

Accessors

size_t **start**();

Returns the starting index of the ***gslice***.

valarraysize_t> **size**();

Returns a ***valarray<size_t>*** containing the lengths of the ***gslice***.

Valarray<size_t> **stride**();

Returns a ***valarray<size_t>*** containing the strides of the ***gslice***.

Example

```
//
// gslice.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
 using namespace std;

int main(void)
{
  int ibuf[27] =
   {0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10};
  size_t len_buf[3] = {3,3,3};
  size_t stride_buf[3] = {9,3,1};

  // create a valarray of ints
  valarray<int>  vi(ibuf,27);

  // create length and stride valarrays
  valarray<size_t> len(len_buf,3);
  valarray<size_t> stride(stride_buf,3);

  // print out the valarray
  cout << vi << endl;

  // Print out all three dimensions (the entire valarray)
  cout << valarray<int>(vi[gslice(0,len,stride)]) << endl;

  // Print a two dimensional slice out of the middle
  valarray<size_t> len2(2);
  len2[0] = 3;
  len2[1] = 3;
  valarray<size_t> stride2(2);
  stride2[0] = 3;
  stride2[1] = 1;
  cout << valarray<int>(vi[gslice(9,len2,stride2)]) << endl;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // Print another two dimensional slice out of the middle
  // but orthogonal to one we just did
  stride2[0] = 9;
  stride2[1] = 1;
  cout << valarray<int>(vi[gslice(3,len2,stride2)]) << endl;

  // Print out the last plane in the middle,
  // (orthogonal to both of the previous ones)
  stride2[0] = 3;
  stride2[1] = 9;
  cout << valarray<int>(vi[gslice(1,len2,stride2)]) << endl;

  // Now how about a diagonal slice?
  // upper left front to lower right back
  stride2[0] = 3;
  stride2[1] = 10;
  cout << valarray<int>(vi[gslice(0,len2,stride2)]) << endl;

  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10]
[0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10]
[1,2,3,4,5,6,7,8,9]
[3,4,5,4,5,6,5,6,7]
[1,2,3,4,5,6,7,8,9]
[0,2,4,3,5,7,6,8,10]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, *slice_array*, *slice*, *gslice_array*, *mask_array*, *indirect_array*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

gslice_array

`Valarray helpers`

Summary

A numeric array class used to represent a BLAS-like slice from a ***valarray***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator%=()
operator&=() operator-=()
operator>>=() operator/=()
operator<<=() operator=()
operator*=() operator^=()
operator+=()

Synopsis

```
#include <valarray>
template <class T>
class gslice_array ;
```

Description

gslice_array<T> creates a ***gslice*** view into a ***valarray***. ***gslice_arrays*** are only produced by applying the ***gslice*** subscript operator to a ***valarray***. The elements in a ***gslice_array*** are references to selected elements in the ***valarray*** (so changing an element in the ***gslice_array*** really changes the corresponding element in the ***valarray***). A ***gslice_array*** does not itself hold any distinct elements. The template cannot be instantiated directly since all its constructors are private. However, you can copy a ***gslice_array*** to a ***valarray*** using either the ***valarray*** copy constructor or the assignment operator. Reference semantics are lost at that point.

Interface

```
template <class T> class gslice_array {
public:

  // types
  typedef T value_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // destructor
  ~gslice_array();

  // public assignment
  void operator= (const valarray<T>& array) const;
  // computed assignment
  void operator*= (const valarray<T>& array) const;
  void operator/= (const valarray<T>& array) const;
  void operator%= (const valarray<T>& array) const;
  void operator+= (const valarray<T>& array) const;
  void operator-= (const valarray<T>& array) const;
  void operator^= (const valarray<T>& array) const;
  void operator&= (const valarray<T>& array) const;
  void operator|= (const valarray<T>& array) const;
  void operator<<= (const valarray<T>& array) const;
  void operator>>= (const valarray<T>& array) const;

  // fill function
  void operator=(const T&);

private:
  // constructors
  gslice_array();
  gslice_array(const gslice_array<T>&);
  // operator =
  gslice_array<T>& operator= (const gslice_array<T>& array);
};
```

Constructors

```
gslice_array();
gslice_array(const gslice_array&);
```

All ***gslice_array*** constructors are private and cannot be called directly. This prevents copy construction of ***gslice_arrays***.

Assignment Operators

```
void operator=(const valarray<T>& x) const;
```

Assigns values from `x` to the selected elements of the ***valarray*** that self refers to. Remember that a ***gslice_array*** never holds any elements itself; it simply refers to selected elements in the ***valarray*** used to generate it.

```
gslice_array<T>&
operator=(const gslice-_array<T>& x);
```

Private assignment operator. Cannot be called directly, thus preventing assignment between ***gslice_arrays***.

Computed Assignment Operators

```
void operator*=(const valarray<T>& val) const;
void operator/=(const valarray<T>& val) const;
void operator%=(const valarray<T>& val) const;
void operator+=(const valarray<T>& val) const;
void operator-=(const valarray<T>& val) const;
void operator^=(const valarray<T>& val) const;
void operator&=(const valarray<T>& val) const;
void operator|=(const valarray<T>& val) const;
void operator<<=(const valarray<T>& val) const;
void operator>>=(const valarray<T>& val) const;
```

Applies the indicated operation using elements from `val` to the selected elements of the ***valarray*** that self refers to. Remember that a ***gslice_array*** never holds any elements itself; it simply refers to selected elements in the ***valarray*** used to generate it.

Member Functions

```
void operator=(const T& x) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Assigns x to the selected elements of the ***valarray*** that self refers to.

Example

```
//
// gslice_array.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
using namespace std;

int main(void)
{
  int ibuf[27] =
   {0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10};
  int buf13[9] = {13,13,13,13,13,13,13,13,13};
  size_t len_buf[3] = {3,3,3};
  size_t stride_buf[3] = {9,3,1};

  // create a valarray of ints
  valarray<int>  vi(ibuf,27);

  // print out the valarray
  cout << vi << endl;

  // Get a two dimensional diagonal slice out of the middle
  valarray<size_t> len2(2);
  len2[0] = 3;
  len2[1] = 3;
  valarray<size_t> stride2(2);
  stride2[0] = 3;
  stride2[1] = 10;
  gslice_array<int> gsl = vi[gslice(0,len2,stride2)];

  // print out the slice
  cout << gsl << endl;

  // Assign 13's to everything in the slice
  gsl = valarray<int>(buf13,9);

  // print out the slice and our original valarray
  cout << gsl << endl << vi <<  endl;

  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10]
[0,2,4,3,5,7,6,8,10]
[13,13,13,13,13,13,13,13,13]
[13,1,2,13,4,5,13,7,8,1,13,3,4,13,6,7,13,9,2,3,13,5,6,13,8,9,13]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, ***slice_array***, ***slice***, ***gslice***, ***mask_array***, ***indirect_array***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

has_facet

Locale Function

Summary

A function template used to determine if a locale has a given facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class Facet> bool has_facet(const locale&) throw();
```

Description

has_facet returns `true` if the requested facet is available in the locale. Otherwise it returns `false`. You specify the facet type by explicitly including the template parameter (see the example below).

Note that if your compiler cannot overload function templates on return type, then you need to use an alternative ***has_facet*** template. The alternative template takes an additional argument that's a pointer to the type of facet you want to check on. The declaration looks like this:

```
template <class Facet>
const bool has_facet(const locale&, Facet*) throw();
```

The example below shows the use of both variations of ***has_facet***.

Example

```
//
// hasfacet.cpp
//

#include <iostream>

int main ()
{
  using namespace std;

  locale loc;

#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  cout << has_facet<ctype<char> >(loc) << endl;
#else
  cout << has_facet(loc,(ctype<char>*)0) << endl;
#endif
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  return 0;
}
```

See Also

locale, *facets*, *use_facet*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

Heap Operations

`Algorithm`

See the entries for ***make_heap***, ***pop_heap***, ***push_heap*** and ***sort_heap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

includes

`Algorithm`

Summary

A basic set of operation for sorted sequences.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2>
 bool includes (InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2,
                InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
          class Compare>
 bool includes (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2,
              InputIterator2 last2, Compare comp);
```

Description

The ***includes*** algorithm compares two sorted sequences and returns `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. It returns `false` otherwise. ***include*** assumes that the sequences are sorted using the less than operator (`operator<`), unless an alternative comparison operator (`comp`) is included.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// includes.cpp
//
 #include <algorithm>
 #include <set>
 #include <iostream>
 using namespace std;

 int main()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
{

  //Initialize some sets
  int a1[10] = {1,2,3,4,5,6,7,8,9,10};
  int a2[6]  = {2,4,6,8,10,12};
  int a3[4]  = {3,5,7,8};
  set<int, less<int> > all(a1, a1+10), even(a2, a2+6),
                       small(a3,a3+4);

 //Demonstrate includes
 cout << "The set: ";
 copy(all.begin(),all.end(),
      ostream_iterator<int,char>(cout," "));
 bool answer = includes(all.begin(), all.end(),
              small.begin(), small.end());
 cout << endl
     << (answer ? "INCLUDES " : "DOES NOT INCLUDE ");
 copy(small.begin(),small.end(),
      ostream_iterator<int,char>(cout," "));
 answer = includes(all.begin(), all.end(),
                   even.begin(), even.end());
 cout << ", and" << endl
     << (answer ? "INCLUDES" : "DOES NOT INCLUDE ");
 copy(even.begin(),even.end(),
      ostream_iterator<int,char>(cout," "));
 cout << endl << endl;

 return 0;
}
```

Program Output

```
The set: 1 2 3 4 5 6 7 8 9 10
INCLUDES 3 5 7 8 , and
DOES NOT INCLUDE 2 4 6 8 10 12
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
set<int, less<int>, allocator<int> >
```

instead of:

```
set<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

set,*set_union*, *set_intersection*, *set_difference*, *set_symmetric_difference*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

indirect_array

`Valarray helpers`

Summary

A numeric array class used to represent elements selected from a ***valarray***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator%=()
operator&=() operator-=()
operator>>=() operator/=()
operator<<=() operator=()
operator*=() operator^=()
operator+=()

Synopsis

```
#include <valarray>
template <class T>
class indirect_array ;
```

Description

indirect_array<T> creates a selective view into a ***valarray***. ***Indirect_array***s are produced by applying the indirect subscript operator to a ***valarray***. The indirect array produced by this subscript contains only the elements of the ***valarray*** whose indices appear as values in the argument. The elements in an ***indirect_array*** are references to selected elements in the ***valarray*** (so changing an element in the ***indirect_array*** really changes the corresponding element in the ***valarray***). An ***indirect_array*** does not itself hold any distinct elements. The template cannot be instantiated directly since all its constructors are private. However, you can copy an ***indirect_array*** to a ***valarray*** using either the ***valarray*** copy constructor or the assignment operator. Reference semantics are lost at that point.

Interface

```
template <class T> class indirect_array {
public:

  // types
  typedef T value_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // destructor
  ~indirect_array();

  // public assignment
  void operator= (const valarray<T>& array) const;
  // computed assignment
  void operator*= (const valarray<T>& array) const;
  void operator/= (const valarray<T>& array) const;
  void operator%= (const valarray<T>& array) const;
  void operator+= (const valarray<T>& array) const;
  void operator-= (const valarray<T>& array) const;
  void operator^= (const valarray<T>& array) const;
  void operator&= (const valarray<T>& array) const;
  void operator|= (const valarray<T>& array) const;
  void operator<<= (const valarray<T>& array) const;
  void operator>>= (const valarray<T>& array) const;

  // fill function
  void operator=(const T&);

private:
  // constructors
  indirect_array();
  indirect_array(const indirect_array<T>&);
  // operator =
  indirect_array<T>&
     operator= (const indirect_array<T>& array);
};
```

Constructors

```
indirect_array();
indirect_array(const indirect_array&);
```

All ***indirect_array*** constructors are private and cannot be called directly. This prevents copy construction of ***indirect_arrays***.

Assignment Operators

```
void operator=(const valarray<T>& x) const;
```

Assigns values from `x` to the selected elements of the ***valarray*** that self refers to. Remember that an ***indirect_array*** never holds any elements itself; it simply refers to selected elements in the ***valarray*** used to generate it.

```
indirect_array<T>&
operator=(const indirect-_array<T>& x);
```

Private assignment operator. Cannot be called directly, thus preventing assignment between ***indirect_arrays***.

Computed Assignment Operators

```
void operator*=(const valarray<T>& val) const;
void operator/=(const valarray<T>& val) const;
void operator%=(const valarray<T>& val) const;
void operator+=(const valarray<T>& val) const;
void operator-=(const valarray<T>& val) const;
void operator^=(const valarray<T>& val) const;
void operator&=(const valarray<T>& val) const;
void operator|=(const valarray<T>& val) const;
void operator<<=(const valarray<T>& val) const;
void operator>>=(const valarray<T>& val) const;
```

Applies the indicated operation using elements from `val` to the selected elements of the ***valarray*** that self refers to. Remember that an ***indirect_array*** never holds any elements itself; it simply refers to selected elements in the ***valarray*** used to generate it.

Member Functions

©Copyright 1999 Rogue Wave Software, Inc.

```
void operator=(const T& x);
```

Assigns x to the selected elements of the valarray that self refers to.

Example

```
//
// indirect_array.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
using namespace std;

int main(void)
{
  int ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
  size_t sbuf[6] = {0,2,3,4,7,8};

  // create a valarray of ints
  valarray<int>         vi(ibuf,10);

  // create a valarray of indices for a selector
  valarray<size_t> selector(sbuf,6);

  // print out the valarray<int>
  cout << vi << endl;

  // Get a indirect_array
  // and assign that indirect to another valarray
  indirect_array<int> select = vi[selector];
  valarray<int> vi3 = select;

  // print out the selective array
  cout << vi3 << endl;

  // Double the selected values
  select += vi3;

  // print out vi1 again
  cout << vi << endl;

  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,9]
[0,2,3,4,7,8]
[0,1,4,6,8,5,6,14,16,9]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, *slice_array*, *slice*, *gslice*, *gslice_array*, *mask_array*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

inner_product

`Generalized Numeric Operation`

Summary

Computes the inner product `A X B` of two ranges `A` and `B`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <numeric>
template <class InputIterator1, class InputIterator2,
          class T>
T inner_product (InputIterator1 first1,
                 InputIterator1 last1,
                 InputIterator2 first2, T init);
template <class InputIterator1, class InputIterator2,
          class T,
          class BinaryOperation1,
          class BinaryOperation2>
T inner_product (InputIterator1 first1,
                 InputIterator1 last1,
                 InputIterator2 first2, T init,
                 BinaryOperation1 binary_op1,
                 BinaryOperation2 binary_op2);
```

Description

There are two versions of ***inner_product***. The first computes an inner product using the default multiplication and addition operators, while the second allows you to specify binary operations to use in place of the default operations.

The first version of the function computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with:

```
acc = acc + ((*i1) * (*i2))
```

for every iterator `i1` in the range `[first1, last1)` and iterator `i2` in the range `[first2, first2 + (last1 - first1))`. The algorithm returns `acc`.

The second version of the function initializes `acc` with `init`, then computes:

```
acc = binary_op1(acc, binary_op2(*i1, *i2))
```

for every iterator `i1` in the range `[first1, last1)` and iterator `i2` in the range `[first2, first2 + (last1 - first1))`.

©Copyright 1999 Rogue Wave Software, Inc.

Complexity

The ***inner_product*** algorithm computes exactly `(last1 - first1)` applications of either:

```
acc + (*i1) * (*i2)
```

or

```
binary_op1(acc, binary_op2(*i1, *i2)).
```

Example

```
//
// inr_prod.cpp
//
 #include <numeric>       //For inner_product
 #include <list>          //For list
 #include <vector>        //For vectors
 #include <functional>    //For plus and minus
 #include <iostream>
 using namespace std;
 int main()
 {
   //Initialize a list and an int using arrays of ints
   int a1[3] = {6, -3, -2};
   int a2[3] = {-2, -3, -2};
   list<int>   l(a1, a1+3);
   vector<int> v(a2, a2+3);
   //Calculate the inner product of the two sets of values
   int inner_prod =
        inner_product(l.begin(), l.end(), v.begin(), 0);
   //Calculate a wacky inner product using the same values
   int wacky =
         inner_product(l.begin(), l.end(), v.begin(), 0,
                       plus<int>(), minus<int>());
   //Print the output
   cout << "For the two sets of numbers: " << endl
        << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << " and  ";
   copy(l.begin(),l.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "," << endl << endl;
   cout << "The inner product is: " << inner_prod << endl;
   cout << "The wacky result is: " << wacky << endl;
   return 0;
 }
```

Program Output

```
For the two sets of numbers:
     -2 -3 -2
 and  6 -3 -2 ,
The inner product is: 1
The wacky result is: 8
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

`list<int, allocator<int> >` and `vector<int, allocator<int> >`

instead of

`list<int>` and `vector<int>`

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

inplace_merge

`Algorithm`

Summary

Merges two sorted sequences into one.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last,
                     Compare comp);
```

Description

The ***inplace_merge*** algorithm merges two sorted consecutive ranges `[first, middle)` and `[middle, last)`, and puts the result of the merge into the range `[first, last)`. The merge is stable, which means that if the two ranges contain equivalent elements, the elements from the first range always precede the elements from the second.

There are two versions of the ***inplace_merge*** algorithm. The first version uses the less than operator (`operator<`) as the default for comparison, and the second version accepts a third argument that specifies a comparison operator.

Complexity

When enough additional memory is available, ***inplace_merge*** does at most `(last - first) - 1` comparisons. If no additional memory is available, an algorithm with `O(NlogN)` complexity (where `N` is equal to `last-first`) may be used.

Example

```
//
// merge.cpp
//
 #include <algorithm>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <vector>
#include <functional>
#include <iostream>
using namespace std;

int main()
{
  int d1[4] = {1,2,3,4};
  int d2[8] = {11,13,15,17,12,14,16,18};

  // Set up two vectors
  vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);


  // Set up four destination vectors
  vector<int> v3(d2,d2 + 8),v4(d2,d2 + 8),
              v5(d2,d2 + 8),v6(d2,d2 + 8);
  // Set up one empty vector
  vector<int> v7;
  // Merge v1 with v2
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
        v3.begin());
  // Now use comparator
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v4.begin(),
        less<int>());
  // In place merge v5
  vector<int>::iterator mid = v5.begin();
  advance(mid,4);
  inplace_merge(v5.begin(),mid,v5.end());
  // Now use a comparator on v6
  mid = v6.begin();
  advance(mid,4);
  inplace_merge(v6.begin(),mid,v6.end(),less<int>());
  // Merge v1 and v2 to empty vector using insert iterator
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
        back_inserter(v7));
  // Copy all cout
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;
  copy(v3.begin(),v3.end(),out);
  cout << endl;
  copy(v4.begin(),v4.end(),out);
  cout << endl;
  copy(v5.begin(),v5.end(),out);
  cout << endl;
  copy(v6.begin(),v6.end(),out);
  cout << endl;
  copy(v7.begin(),v7.end(),out);
  cout << endl;

  // Merge v1 and v2 to cout
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
        ostream_iterator<int,char>(cout," "));
  cout << endl;
  return 0;
}
```

Program Output

```
1 2 3 4
1 2 3 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
11 12 13 14 15 16 17 18
11 12 13 14 15 16 17 18
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int, allocator,int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

merge

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Input Iterators

`Iterator`

Summary

A read-only, forward moving iterator.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

NOTE:For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Input iterators are read-only, forward moving iterators that satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirement descriptions listed below:

`a and b`	values of type `X`
`n`	value representing a `distance` between two iterators
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Input Iterators

The following expressions must be valid for input iterators:

`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	a == b implies `*a == *b`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++`	returns type `T`
`a -> m`	returns `(*a).m`

For input iterators, `a == b` does not imply that `++a == ++b`.

Algorithms using input iterators should be single pass algorithms. That is, they should not pass through the same iterator twice.

©Copyright 1999 Rogue Wave Software, Inc.

The value of type `T` does not have to be an `lvalue`.

See Also

Iterators, ***Output Iterators***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Insert Iterators

`Insert Iterator`

Summary

An iterator adaptor that allows an iterator to insert into a container rather than overwrite elements in the container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class Container>
class insert_iterator :
      iterator<output_iterator_tag,void,void,void,void> ;

template <class Container>
class back_insert_iterator;

template <class Container>
class front_insert_iterator;
```

Description

Insert iterators are iterator adaptors that let an iterator *insert* new elements into a collection rather than overwrite existing elements when copying to a container. There are several types of insert iterator classes.

- The class ***back_insert_iterator*** is used to insert items at the end of a collection. The function `back_inserter` can be used with an iterator inline, to create an instance of a ***back_insert_iterator*** for a particular collection type.
- The class ***front_insert_iterator*** is used to insert items at the start of a collection. The function `front_inserter` creates an instance of a ***front_insert_iterator*** for a particular collection type.
- An ***insert_iterator*** inserts new items into a collection at a location defined by an iterator supplied to the constructor. Like the other insert iterators, ***insert_iterator*** has a helper function called `inserter`, which takes a collection and an iterator into that collection, and creates an instance of the ***insert_iterator***.

Interface

```
template <class Container>
 class insert_iterator : public
       iterator<output_iterator_tag,void,void,void,void> ; {

public:
   typedef Container container_type;
   insert_iterator (Container&, typename
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                    Container::iterator);
   insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   insert_iterator<Container>& operator* ();
   insert_iterator<Container>& operator++ ();
   insert_iterator<Container>& operator++ (int);
};

template <class Container>
 class back_insert_iterator : public
       iterator<output_iterator_tag,void,void,void,void> ; {

public:
   typedef Container container_type;
   explicit back_insert_iterator (Container&);
   back_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   back_insert_iterator<Container>& operator* ();
   back_insert_iterator<Container>& operator++ ();
   back_insert_iterator<Container> operator++ (int);
};

template <class Container>
 class front_insert_iterator : public
       iterator<output_iterator_tag,void,void,void,void> ; {

public:
   typedef Container container_type;
   explicit front_insert_iterator (Container&);
   front_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   front_insert_iterator<Container>& operator* ();
   front_insert_iterator<Container>& operator++ ();
   front_insert_iterator<Container> operator++ (int);
};

 template <class Container, class Iterator>
 insert_iterator<Container> inserter (Container&, Iterator);

 template <class Container>
 back_insert_iterator<Container> back_inserter (Container&);

 template <class Container>
 front_insert_iterator<Container>
                       front_inserter (Container&);
```

See Also

back_insert_iterator, *front_insert_iterator*, *insert_iterator*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

insert_iterator, inserter

```
Insert Iterator
```


Summary

An insert iterator used to insert items into a collection rather than overwrite the collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

container_type

Member Functions

inserter()
operator*()
operator++()
operator=()

Synopsis

```
#include <iterator>
template <class Container>
class insert_iterator;
```

Description

Insert iterators let you *insert* new elements into a collection rather than copy a new element's value over the value of an existing element. The class ***insert_iterator*** is used to insert items into a specified location of a collection. The function `inserter` creates an instance of an ***insert_iterator*** given a particular collection type and iterator. An ***insert_iterator*** can be used with ***vector***s, ***deque***s, ***list***s, ***map***s and ***set***s.

Interface

```
template <class Container>
class insert_iterator : public
      iterator<output_iterator_tag,void,void,void,void> ;  {
protected:
   Container* container;
public:
   typedef Container container_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   insert_iterator (Container&, typename Container::iterator);
   insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   insert_iterator<Container>& operator* ();
   insert_iterator<Container>& operator++ ();
   insert_iterator<Container>& operator++ (int);
};

template <class Container, class Iterator>
insert_iterator<Container> inserter (Container&, Iterator)
```

Types

container_type

The type of container acted on by the iterator.

Constructors

```
insert_iterator(Container& x,
                typename Container::iterator i);
```

Creates an instance of an ***insert_iterator*** associated with container `x` and iterator `i`.

Operators

```
insert_iterator<Container>&
operator=(const typename Container::value_type& value);
```

Inserts a copy of `value` into the container at the location specified by the `insert_iterator`, increments the iterator, and returns `*this`.

```
insert_iterator<Container>&
operator*();
```

Returns `*this` (the input iterator itself).

```
insert_iterator<Container>&
operator++();
insert_iterator<Container>&
operator++(int);
```

Increments the insert iterator and returns `*this`.

Non-member Functions

```
template <class Container, class Iterator>
insert_iterator<Container>
inserter(Container& x, Iterator i);
```

Returns an ***insert_iterator*** that inserts elements into container `x` at location `i`. This function allows you to create insert iterators inline.

Example

```
 #include <iterator>
 #include <vector>
 #include <iostream>
 using namespace std;
 int main()
 {
   //Initialize a vector using an array
   int arr[4] = {3,4,7,8};
   vector<int> v(arr,arr+4);
   //Output the original vector
   cout << "Start with a vector: " << endl << "     ";
   copy(v.begin(),v.end(),
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        ostream_iterator<int,char>(cout," "));
   //Insert into the middle
   insert_iterator<vector<int> >  ins(v, v.begin()+2);
   *ins = 5;
   *ins = 6;
   //Output the new vector
   cout << endl << endl;
   cout << "Use an insert_iterator: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   return 0;
 }
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

back_insert_iterator, *front_insert_iterator*, *Insert Iterators*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

ios_base

`Base Class`

Summary

Defines member types and maintains data for classes that inherit from it.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

event_callback
fmtflags
iostate
openmode
seekdir

Member Functions

boolalpha()	is_sync()	oct()	
copyfmt()	iword()	precision()	skipws()
dec()	left()	pword()	sync_with_stdio()
failure()	noboolalpha()	register_callback()	unitbuf()
fixed()	noshowbase()	right()	unsetf()
flags()	noshowpoint()	scientific()	uppercase()
getloc()	noshowpos()	setf()	what()
hex()	noskipws()	showbase()	width()
imbue()	nounitbuf()	showpoint()	xalloc()
internal()	nouppercase()	showpos()	

Synopsis

```
#include <ios>
class ios_base;
```

Description

The class ***ios_base*** defines several member types:

- A class `failure` derived from `exception`.

©Copyright 1999 Rogue Wave Software, Inc.

- A class `Init`.
- Three bitmask types: `fmtflags`, `iostate`, and `openmode`.
- Two enumerated types: `seekdir` and `event`.

It maintains several kinds of data:

- Control information that influences how to interpret (format) input sequences and how to generate (format) output sequences.
- Locale object used within the stream classes.
- Additional information that is stored by the program for its private use.

Interface

```
class ios_base {

 public:

   class failure : public exception {

      public:

          explicit failure(const string& msg);
          virtual ~failure() throw();
          virtual const char* what() const throw();
    };

   typedef int      fmtflags;

   enum fmt_flags {
                    boolalpha   = 0x0001,
                    dec         = 0x0002,
                    fixed       = 0x0004,
                    hex         = 0x0008,
                    internal    = 0x0010,
                    left        = 0x0020,
                    oct         = 0x0040,
                    right       = 0x0080,
                    scientific  = 0x0100,
                    showbase    = 0x0200,
                    showpoint   = 0x0400,
                    showpos     = 0x0800,
                    skipws      = 0x1000,
                    unitbuf     = 0x2000,
                    uppercase   = 0x4000,
                    adjustfield = left | right | internal,
                    basefield   = dec | oct | hex,
                    floatfield  = scientific | fixed
                  };

   typedef int      iostate;

   enum io_state {
                   goodbit     = 0x00,
                   badbit      = 0x01,
                   eofbit      = 0x02,
                   failbit     = 0x04
                 };

   typedef int      openmode;

   enum open_mode {
                    app         = 0x01,
                    binary      = 0x02,
                    in          = 0x04,
                    out         = 0x08,
                    trunc       = 0x10,
                    ate         = 0x20
                  };

   typedef int      seekdir;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   enum seek_dir {
                     beg         = 0x0,
                     cur         = 0x1,
                     end         = 0x2
                 };


   class Init;

   fmtflags flags() const;
   fmtflags flags(fmtflags fmtfl);
   fmtflags setf(fmtflags fmtfl);
   fmtflags setf(fmtflags fmtfl, fmtflags mask);
   void unsetf(fmtflags mask);

   streamsize precision() const;
   streamsize precision(streamsize prec);
   streamsize width() const;
   streamsize width(streamsize wide);

   locale imbue(const locale& loc);
   locale getloc() const

   static int xalloc();
   long&  iword(int index);
   void*& pword(int index);

   ios_base& copyfmt(const ios_base& rhs);

   enum event     {
                    erase_event   = 0x0001,
                    imbue_event   = 0x0002,
                    copyfmt_event = 0x004
                  };

   typedef void (*event_callback) (event, ios_base&,
                  int index);
   void register_callback(event_callback fn, int index);

   bool synch_with_stdio(bool sync = true);
   bool is_synch();

  protected:

   ios_base();
   virtual ~ios_base();

  private:

   union ios_user_union {
                           long  lword;
                           void* pword;
                        };

   union ios_user_union *userwords_;
};

 ios_base& boolalpha(ios_base&);
 ios_base& noboolalpha(ios_base&);
 ios_base& showbase(ios_base&);
 ios_base& noshowbase(ios_base&);
 ios_base& showpoint(ios_base&);
 ios_base& noshowpoint(ios_base&);
 ios_base& showpos(ios_base&);
 ios_base& noshowpos(ios_base&);
 ios_base& skipws(ios_base&);
 ios_base& noskipws(ios_base&);
 ios_base& uppercase(ios_base&);
 ios_base& nouppercase(ios_base&);
 ios_base& internal(ios_base&);
 ios_base& left(ios_base&);
 ios_base& right(ios_base&);
 ios_base& dec(ios_base&);
 ios_base& hex(ios_base&);
 ios_base& oct(ios_base&);
 ios_base& fixed(ios_base&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
ios_base& scientific(ios_base&);
ios_base& unitbuf(ios_base&);
ios_base& nounitbuf(ios_base&);
```

Types

fmtflags

The type `fmtflags` is a bitmask type. Setting its elements has the following effects:

`showpos`	Generates a + sign in non-negative generated numeric output.
`showbase`	Generates a prefix indicating the numeric base of generated integer output
`uppercase`	Replaces certain lowercase letters with their uppercase equivalents in generated output
`showpoint`	Generates a decimal-point character unconditionally in generated floating-point output
`boolalpha`	Inserts and extracts bool type in alphabetic format
`unitbuf`	Flushes output after each output operation
`internal`	Adds fill characters at a designated internal point in certain generated output. If no such point is designated, it's identical to `right`.
`left`	Adds fill characters on the right (final positions) of certain generated output
`right`	Adds fill characters on the left (initial positions) of certain generated output
`dec`	Converts integer input or generates integer output in decimal base
`hex`	Converts integer input or generates integer output in hexadecimal base
`oct`	Converts integer input or generates integer output in octal base
`fixed`	Generates floating-point output in fixed-point notation
`scientific`	Generates floating-point output in scientific notation
`skipws`	Skips leading white space before certain input operation.

iostate

The type `iostate` is a bitmask type. Setting its elements has the following effects:

`badbit`	Indicates a loss of integrity in an input or output sequence.
`eofbit`	Indicates that an input operation reached the end of an input sequence.
`failbit`	Indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

openmode

The type `openmode` is a bitmask type. Setting its elements has the following effects:

`app`	Seeks to the end before writing.
`ate`	Opens and seeks to the end immediately after opening.
`binary`	Performs input and output in binary mode.
`in`	Opens for input.
`out`	Opens for output.
`trunc`	Truncates an existing stream when opening.

seekdir

The type `seekdir` is a bitmask type. Setting its elements has the following effects:

`beg`	Requests a seek relative to the beginning of the stream.
`cur`	Requests a seek relative to the current position within the sequence.
`end`	Requests a seek relative to the current end of the sequence.

event_callback

The type `event_callback` is the type of the callback function used as a parameter in the function `register_callback`. These functions allow you to use the `iword`, `pword` mechanism in an exception-safe environment.

©Copyright 1999 Rogue Wave Software, Inc.

Public Constructors

```
ios_base();
```

The `ios_base` members have an indeterminate value after construction.

Public Destructors

```
virtual
 ~ios_base();
```

Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` as `(*fn)(erase_event,*this, index)` at such a time that any `ios_base` member function called from within `fn` has well-defined results.

Public Member Functions

```
ios_base&
copyfmt(const ios_base& rhs);
```

Assigns to the member objects of `*this` the corresponding member objects of `rhs`. The contents of the union pointed to by `pword` and `iword` are copied, not the pointers themselves. Before copying any part of `rhs`, calls each registered callback pair `(fn,index)` as `(*fn)(erase_even,*this, index)`. After all parts have been replaced, calls each callback pair that was copied from `rhs` as `(*fn)(copy_event,*this,index)`.

```
fmtflags
flags() const;
```

Returns the format control information for both input and output.

```
fmtflags
flags(fmtflags fmtfl);
```

Saves the format control information, then sets it to `fmtfl` and returns the previously saved value.

```
locale
getloc() const;
```

Returns the imbued locale, which is used to perform locale-dependent input and output operations. The default locale, `locale::locale()`, is used if no other locale object has been imbued in the stream by a call to the `imbue` function.

```
locale
imbue(const locale& loc);
```

Saves the value returned by `getloc()`, then assigns `loc` to a private variable and calls each registered callback pair `(fn, index)` as `(*fn)(imbue_event,*this, index)`. It then returns the previously saved value.

```
bool
is_sync();
```

Returns `true` if the C++ standard streams and the standard C streams are synchronized. Otherwise returns `false`. This function is not part of the C++ standard.

```
long&
iword(int idx);
```

Returns `userwords_[idx].iword`. If `userwords_` is a null pointer, allocates a union of `long` and `void*` of unspecified size and stores a pointer to its first element in `userwords_`. The function then extends the union pointed to by `userwords_` to include the element `userwords_[idx]`. Each newly allocated element of the union is initialized to zero. The reference returned may become invalid after another call to the object's `iword` or `pword` member with a different index, after a call to its `copyfmt` member, or when the object is destroyed.

```
streamsize
precision() const;
```

Returns the precision (number of digits after the decimal point) to generate on certain output conversions.

```
streamsize
precision(streamsize prec);
```

©Copyright 1999 Rogue Wave Software, Inc.

Saves the precision, then sets it to `prec` and returns the previously saved value.

```
void*&
pword(int idx);
```

Returns `userword_[idx].pword`. If `userwords_` is a null pointer, allocates a union of `long` and `void*` of unspecified size and stores a pointer to its first element in `userwords_`. The function then extends the union pointed to by `userwords_` to include the element `userwords_[idx]`. Each newly allocated element of the array is initialized to zero. The reference returned may become invalid after another call to the object's `pword` or `iword` member with a different index, after a call to its `copyfmt` member, or when the object is destroyed.

```
void
register_callback(event_callback fn, int index);
```

Registers the pair `(fn, index)` such that during calls to `imbue()`, `copyfmt()`, or `~ios_base()`, the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event. Identical pairs are not merged; a function registered twice is called twice per event.

```
fmtflags
setf(fmtflags fmtfl);
```

Saves the format control information, then sets it to `fmtfl` and returns the previously saved value.

```
fmtflags
setf(fmtflags fmtfl, fmtflags mask);
```

Saves the format control information, then clears `mask` in `flags()`, sets `fmtfl & mask in flags()` and returns the previously saved value.

```
bool
sync_with_stdio(bool sync = true);
```

When called with a `false` argument, allows the C++ standard streams to operate independently of the standard C streams, which greatly improves performance. When called with a `true` argument, restores the default synchronization. The return value of the function is the status of the synchronization at the time of the call.

```
void
unsetf(fmtflags mask);
```

Clears `mask` in `flags()`.

```
streamsize
width() const;
```

Returns the field width (number of characters) to generate on certain output conversions.

```
streamsize
width(streamsize wide);
```

Saves the field width, then sets it to `wide` and returns the previously saved value.

```
static int
xalloc();
```

Returns the next static index that can be used with `pword` and `iword`. This is useful if you want to share data between several stream objects.

Class failure

The class ***failure*** defines the base class for the types of all objects thrown as exceptions by functions in the ***iostreams*** library. It reports errors detected during stream buffer operations.

```
explicit failure(const string& msg);
```

Constructs an object of class `failure`, initializing the base class with `exception(msg)`.

```
const char*
what() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns the message `msg` with which the exception was created.

Class Init

The class ***Init*** describes an object whose construction ensures the construction of the eight objects declared in `<iostream>`, which associate file stream buffers with the standard C streams.

Non-member Functions

```
ios_base&
boolalpha(ios_base& str);
```

Calls `str.setf(ios_base::boolalpha)` and returns `str`.

```
ios_base&
dec(ios_base& str);
```

Calls `str.setf(ios_base::dec, ios_base::basefield)` and returns `str`.

```
ios_base&
fixed(ios_base& str);
```

Calls `str.setf(ios_base::fixed, ios_base::floatfield)` and returns `str`.

```
ios_base&
hex(ios_base& str);
```

Calls `str.setf(ios_base::hex, ios_base::basefield)` and returns `str`.

```
ios_base&
internal(ios_base& str);
```

Calls `str.setf(ios_base::internal, ios_base::adjustfield)` and returns `str`.

```
ios_base&
left(ios_base& str);
```

Calls `str.setf(ios_base::left, ios_base::adjustfield)` and returns `str`.

```
ios_base&
noboolalpha(ios_base& str);
```

Calls `str.unsetf(ios_base::boolalpha)` and returns `str`.

```
ios_base&
noshowbase(ios_base& str);
```

Calls `str.unsetf(ios_base::showbase)` and returns `str`.

```
ios_base&
noshowpoint(ios_base& str);
```

Calls `str.unsetf(ios_base::showpoint)` and returns `str`.

```
ios_base&
noshowpos(ios_base& str);
```

Calls `str.unsetf(ios_base::showpos)` and returns `str`.

```
ios_base&
noskipws(ios_base& str);
```

Calls `str.unsetf(ios_base::skipws)` and returns `str`.

```
ios_base&
nounitbuf(ios_base& str);
```

Calls `str.unsetf(ios_base::unitbuf)` and returns `str`.

©Copyright 1999 Rogue Wave Software, Inc.

```
ios_base&
nouppercase(ios_base& str);
```

Calls `str.unsetf(ios_base::uppercase)` and returns `str`.

```
ios_base&
oct(ios_base& str);
```

Calls `str.setf(ios_base::oct, ios_base::basefield)` and returns `str`.

```
ios_base&
right(ios_base& str);
```

Calls `str.setf(ios_base::right, ios_base::adjustfield)` and returns `str`.

```
ios_base&
scientific(ios_base& str);
```

Calls `str.setf(ios_base::scientific, ios_base::floatfield)` and returns `str`.

```
ios_base&
showbase(ios_base& str);
```

Calls `str.setf(ios_base::showbase)` and returns `str`.

```
ios_base&
showpoint(ios_base& str);
```

Calls `str.setf(ios_base::showpoint)` and returns `str`.

```
ios_base&
showpos(ios_base& str);
```

Calls `str.setf(ios_base::showpos)` and returns `str`.

```
ios_base&
skipws(ios_base& str);
```

Calls `str.setf(ios_base::skipws)` and returns `str`.

```
ios_base&
unitbuf(ios_base& str);
```

Calls `str.setf(ios_base::unitbuf)` and returns `str`.

```
ios_base&
uppercase(ios_base& str);
```

Calls `str.setf(ios_base::uppercase)` and returns `str`.

See Also

basic_ios(3C++), ***basic_istream***(3C++), ***basic_ostream***(3C++), ***char_traits***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.4.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

iosfwd

Summary

The header `iosfwd` forward declares the input/output library template classes and specializes them for wide and tiny characters. It also defines the positional types used in class `char_traits` instantiated on tiny and wide characters.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iosfwd>
// forward declare the traits class
template<class charT> struct char_traits;

// forward declare the positioning class
template<class stateT> class fpos;

// forward declare the state class
class mbstate_t;

// forward declare the allocator class
template<class T> class allocator;

// forward declare the iostreams template classes
template<class charT,class traits=char_traits<charT>>
    class basic_ios;
template<class charT,class traits=char_traits<charT>>
    class basic_streambuf;
template<class charT,class traits=char_traits<charT>>
    class basic_istream;
template<class charT,class traits=char_traits<charT>>
    class basic_ostream;
template<class charT,class traits=char_traits<charT>,
         class Allocator = allocator<charT> >
    class basic_stringbuf;
template<class charT,class traits=char_traits<charT>,
         class Allocator = allocator<charT> >
    class basic_istringstream;
template<class charT,class traits=char_traits<charT>,
         class Allocator = allocator<charT> >
    class basic_ostringstream;
template<class charT,class traits=char_traits<charT>>
    class basic_filebuf;
template<class charT,class traits=char_traits<charT>>
    class basic_ifstream;
template<class charT,class traits=char_traits<charT>>
    class basic_ofstream;
template<class charT,class traits=char_traits<charT>>
    class ostreambuf_iterator;
template<class charT,class traits=char_traits<charT>>
    class istreambuf_iterator;
template<class charT,class traits=char_traits<charT>>
    class basic_iostream;
template<class charT,class traits=char_traits<charT>,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        class Allocator = allocator<charT> >
    class basic_stringstream;
template<class charT,class traits=char_traits<charT>>
    class basic_fstream;

// specializations on tiny characters
typedef basic_ios<char>              ios;
typedef basic_streambuf<char>        streambuf;
typedef basic_istream<char>          istream;
typedef basic_ostream<char>          ostream;
typedef basic_stringbuf<char>        stringbuf;
typedef basic_istringstream<char>    istringstream;
typedef basic_ostringstream<char>    ostringstream;
typedef basic_filebuf<char>          filebuf;
typedef basic_ifstream<char>         ifstream;
typedef basic_ofstream<char>         ofstream;
typedef basic_iostream<char>         iostream;
typedef basic_stringstream<char>     stringstream;
typedef basic_fstream<char>          fstream;

// specializations on wide characters
typedef basic_ios<wchar_t>           wios;
typedef basic_streambuf<wchar_t>     wstreambuf;
typedef basic_istream<wchar_t>       wistream;
typedef basic_ostream<wchar_t>       wostream;
typedef basic_stringbuf<wchar_t>     wstringbuf;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_filebuf<wchar_t>       wfilebuf;
typedef basic_ifstream<wchar_t>      wifstream;
typedef basic_ofstream<wchar_t>      wofstream;
typedef basic_iostream<wchar_t>      wiostream;
typedef basic_stringstream<wchar_t>  wstringstream;
typedef basic_fstream<wchar_t>       wfstream;

// positional types used by char_traits
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;

typedef long            streamoff;
typedef long            wstreamoff;
```

See Also

fpos(3C++), ***char_traits***(3C++), ***basic_ios***(3C++), ***basic_streambuf***(3C++), ***basic_istream***(3C++), ***basic_ostream***(3C++), ***basic_iostream***(3C++), ***basic_stringbuf***(3C++), ***basic_istringstream***(3C++), ***basic_ostringstream***(3C++), ***basic_stringstream***(3C++), ***basic_filebuf***(3C++), ***basic_ifstream***(3C++), ***basic_ofstream***(3C++), ***basic_fstream***(3C++), ***istreambuf_iterator***(3C++), ***ostreambuf_iterator***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

isalnum

`Locale Convenience Function`

Summary

Determines if a character is alphabetic or numeric.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT>
bool isalnum (charT c, const locale& loc) const;
```

Description

The ***isalnum*** function returns `true` if the character passed as a parameter is either part of the alphabet specified by the locale parameter or a decimal digit. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

Example

```
//
// isalnum.cpp
//

#include <iostream>

int main ()
{
  using namespace std;

  locale loc;

  cout << isalnum('a',loc) << endl;
  cout << isalnum(',',loc) << endl;

  return 0;
}
```

See Also

other is functions, ***locale***, ***ctype***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

isalpha

`Locale Convenience Function`

Summary

Determines if a character is alphabetic.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isalpha (charT c, const locale& loc) const;
```

Description

The ***isalpha*** function returns `true` if the character passed as a parameter is part of the alphabet specified by the locale parameter. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

iscntrl

`Locale Convenience Function`

Summary

Determines if a character is a control character.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool iscntrl (charT c, const locale& loc) const;
```

Description

The ***iscntrl*** function returns `true` if the character passed as a parameter is a control character. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

isdigit

`Locale Convenience Function`

Summary

Determines if a character is a decimal digit.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isdigit (charT c, const locale& loc) const;
```

Description

The ***isdigit*** function returns `true` if the character passed as a parameter is a decimal digit. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

isgraph

`Locale Convenience Function`

Summary

Determines if a character is a graphic character.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isgraph (charT c, const locale& loc) const;
```

Description

The ***isgraph*** function returns `true` if the character passed as a parameter is a graphic character. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

islower

`Locale Convenience Function`

Summary

Determines whether a character is lower case.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool islower (charT c, const locale& loc) const;
```

Description

The ***islower*** function returns `true` if the character passed as a parameter is lower case. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

isprint

`Locale Convenience Function`

Summary

Determines if a character is printable.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isprint (charT c, const locale& loc) const;
```

Description

The ***isprint*** function returns `true` if the character passed as a parameter is printable. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

ispunct

`Locale Convenience Function`

Summary

Determines if a character is punctuation.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool ispunct (charT c, const locale& loc) const;
```

Description

The ***ispunct*** function returns `true` if the character passed as a parameter is punctuation. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

isspace

`Locale Convenience Function`

Summary

Determines if a character is a space.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isspace (charT c, const locale& loc) const;
```

Description

The ***isspace*** function returns `true` if the character passed as a parameter is a space character. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

istream_iterator

`Iterators`

Summary

A stream iterator that has iterator capabilities for istreams. This iterator allows generic algorithms to be used directly on streams.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
istream_type
traits_type
value_type

Member Functions

operator!=()
operator*()
operator++()
operator->()
operator==()

Synopsis

```
#include <iterator>
template <class T, class charT, class traits = ios_traits<charT>,
          class Distance = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag,
                                T,Distance>;
```

Description

Stream iterators are the standard iterator interface for input and output streams.

The class ***istream_iterator*** reads elements from an input stream using operator >>. A value of type `T` is retrieved and stored when the iterator is constructed and each time `operator++` is called. The iterator is equal to the end-of-stream iterator value if the end-of-file is reached. You can use the constructor with no arguments to create an end-of-stream iterator. The only valid use of this iterator is to compare to other iterators when checking for end of file. Do not attempt to

©Copyright 1999 Rogue Wave Software, Inc.

dereference the end-of-stream iterator; it plays the same role as the past-the-end iterator of the `end()` function of containers. Since an ***istream_iterator*** is an input iterator, you cannot assign to the value returned by dereferencing the iterator. This also means that ***istream_iterators*** can only be used for single pass algorithms.

Since a new value is read every time the `operator++` is used on an ***istream_iterator***, that operation is not equality-preserving. This means that `i == j` does *not* mean that `++i == ++j` (although two end-of-stream iterators are always equal).

Interface

```
template <class T, class charT, class traits = ios_traits<charT>
          class Distance = ptrdiff_t>
 class istream_iterator :
          public iterator<input_iterator_tag,T, Distance>
 {

 public:
    typedef T value_type;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT,traits> istream_type;

    istream_iterator();
    istream_iterator (istream_type&);
    istream_iterator
          (const stream_iterator<T,charT,traits,Distance>&);
    ~istream_itertor ();

    const T& operator*() const;
    const T* operator ->() const;
    istream_iterator <T,charT,traits,
                      Distance>& operator++();
    istream_iterator <T,charT,traits,Distance>
                      operator++ (int)
 };

// Non-member Operators

template <class T, class charT, class traits,class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>&,
          const istream_iterator<T,charT,traits,Distance>&);

template <class T, class charT, class traits,class Distance>
bool operator!=(const istream_iterator<T,charT,traits,Distance>&,
          const istream_iterator<T,charT,traits,Distance>&);
```

Types

`value_type;`

Type of value to stream in.

`char_type;`

Type of character the stream is built on.

`traits_type;`

Traits used to build the stream.

`istream_type;`

Type of stream this iterator is constructed on.

Constructors

`istream_iterator();`

Constructs an end-of-stream iterator. This iterator can be used to compare against an end-of-stream condition. Use it to provide end iterators to algorithms.

©Copyright 1999 Rogue Wave Software, Inc.

```
istream_iterator(istream& s);
```

Constructs an ***istream_iterator*** on the given stream.

```
istream_iterator(const istream_iterator& x);
```

Copy constructor.

Destructors

```
~istream_iterator();
```

Operators

```
const T&
operator*() const;
```

Returns the current value stored by the iterator.

```
const T*
operator->() const;
```

Returns a pointer to the current value stored by the iterator.

```
istream_iterator& operator++()
istream_iterator operator++(int)
```

Retrieves the next element from the input stream.

Non-member Operators

```
bool
operator==(const istream_iterator<T,charT,traits,
            Distance>& x,
            const
            istream_iterator<T,charT,traits,Distance>& y)
```

Returns `true` if `x` is the same as `y`.

```
bool
operator!=(const istream_iterator<T,charT,traits,
            Distance>& x,
            const
            istream_iterator<T,charT,traits,Distance>& y)
```

Returns `true` if `x` is not the same as `y`.

Example

```
//
// io_iter.cpp
//
 #include <iterator>
 #include <vector>
 #include <numeric>
 #include <iostream>
 using namespace std;

 int main ()
 {
   vector<int> d;
   int total = 0;
   //
   // Collect values from cin until end of file
   // Note use of default constructor to get ending iterator
   //
   cout << "Enter a sequence of integers (eof to quit): " ;
   copy(istream_iterator<int,char>(cin),
        istream_iterator<int,char>(),
        inserter(d,d.begin()));
   //
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // stream the whole vector and the sum to cout
  //
  copy(d.begin(),d.end()-1,
       ostream_iterator<int,char>(cout," + "));
  if (d.size())
    cout << *(d.end()-1) << " = " <<
         accumulate(d.begin(),d.end(),total) << endl;
  return 0;
}
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. You also have to include all parameters to the istream_iterator template. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

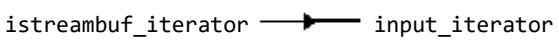
See Also

Iterators, *ostream_iterator*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

istreambuf_iterator



Summary

Reads successive characters from the stream buffer for which it was constructed.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
istream_type
streambuf_type
traits_type

Member Functions

equal()
operator*()
operator++()
operator==()

Synopsis

```
#include <streambuf>
template<class charT, class traits = char_traits<charT> >
class istreambuf_iterator
: public input_iterator
```

Description

The template class ***istreambuf_iterator*** reads successive characters from the stream buffer for which it was constructed. `operator*` gives access to the current input character, if any, and `operator++` advances to the next input character. If the end of stream is reached, the iterator becomes equal to the end of stream iterator value, which is constructed by the default constructor, `istreambuf_iterator()`. An ***istreambuf_iterator*** object can be used only for one-pass-algorithms.

Interface

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class charT, class traits = char_traits<charT> >
class istreambuf_iterator
: public input_iterator {

 public:

  typedef charT                          char_type;
  typedef typename traits::int_type      int_type;
  typedef traits                         traits_type;
  typedef basic_streambuf<charT, traits> streambuf_type;
  typedef basic_istream<charT, traits>   istream_type;

  class proxy;

  istreambuf_iterator() throw();
  istreambuf_iterator(istream_type& s)  throw();
  istreambuf_iterator(streambuf_type *s) throw();
  istreambuf_iterator(const proxy& p) throw();

  char_type operator*();
  istreambuf_iterator<charT, traits>& operator++();
  proxy operator++(int);
  bool equal(istreambuf_iterator<charT, traits>& b);

};



template<class charT, class traits>
bool operator==(istreambuf_iterator<charT, traits>& a,
                istreambuf_iterator<charT, traits>& b);

template<class charT, class traits>
bool operator !=(istreambuf_iterator<charT, traits>& a,
                istreambuf_iterator<charT, traits>& b);
```

Types

`char_type`

The type `char_type` is a synonym for the template parameter `charT`.

`int_type`

The type `int_type` is a synonym of type `traits::in_type`.

`istream_type`

The type `istream_type` is an instantiation of class `basic_istream` on types `charT` and `traits`:

```
typedef basic_istream<charT, traits> istream_type;
```

`streambuf_type`

The type `streambuf_type` is an instantiation of class `basic_streambuf` on types `charT` and `traits`:

```
typedef basic_streambuf<charT, traits> streambuf_type;
```

`traits_type`

The type `traits_type` is a synonym for the template parameter `traits`.

Nested Class Proxy

Class ***istreambuf_iterator<charT,traits>::proxy*** is a temporary placeholder for the return value of the post-increment operator. It keeps the character pointed to by the previous value of the iterator for some possible future access.

Constructors

`istreambuf_iterator``()`
```
  throw();
```

©Copyright 1999 Rogue Wave Software, Inc.

Constructs the end of stream iterator.

```
istreambuf_iterator(istream_type& s)
  throw();
```

Constructs an `istreambuf_iterator` that inputs characters using the `basic_streambuf` object pointed to by `s.rdbuf()`. If `s.rdbuf()` is a null pointer, the `istreambuf_iterator` is the end-of-stream iterator.

```
istreambuf_iterator(streambuf_type *s)
  throw();
```

Constructs an `istreambuf_iterator` that inputs characters using the `basic_streambuf` object pointed to by `s`. If `s` is a null pointer, the `istreambuf_iterator` is the end-of-stream iterator.

```
istreambuf_iterator(const proxy& p)
  throw();
```

Constructs an `istreambuf_iterator` that uses the `basic_streambuf` object embedded in the proxy object.

Member Operators

```
char_type
operator*();
```

Returns the character pointed to by the input sequence of the attached stream buffer. If no character is available, the iterator becomes equal to the end-of-stream iterator.

```
istreambuf_iterator<charT, traits>&
operator++();
```

Increments the input sequence of the attached stream buffer to point to the next character. If the current character is the last one, the iterator becomes equal to the end-of-stream iterator.

```
proxy
operator++(int);
```

Increments the input sequence of the attached stream buffer to point to the next character. If the current character is the last one, the iterator becomes equal to the end-of-stream iterator. The proxy object returned contains the character pointed to before carrying out the post-increment operator.

Public Member Functions

```
bool
equal(istreambuf_iterator<charT, traits>& b);
```

Returns `true` if and only if both iterators are at end of stream, or neither is at end of stream, regardless of what stream buffer object they are using.

Non-member Functions

```
template<class charT, class traits>
bool
operator==(istreambuf_iterator<charT, traits>& a,
                istreambuf_iterator<charT, traits>& b);
```

Returns `a.equal(b)`.

```
template<class charT, class traits>
bool
operator==(istreambuf_iterator<charT, traits>& a,
                istreambuf_iterator<charT, traits>& b);
```

Returns !(`a.equal(b)`).

Example

```
//
// stdlib/examples/manual/istreambuf_iterator.cpp
//
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // open the file is_iter.out for reading and writing
  ofstream out("is_iter.out",
               ios_base::out | ios_base::in );

  // output the example sentence into the file
  out << "Ceci est un simple example pour demontrer le"
      << endl;
  out << "fonctionement de istreambuf_iterator";

  // seek to the beginning of the file
  out.seekp(0);

  // construct an istreambuf_iterator pointing to
  // the ofstream object underlying stream buffer
  istreambuf_iterator<char> iter(out.rdbuf());

  // construct an end of stream iterator
  istreambuf_iterator<char> end_of_stream_iterator;

  cout << endl;

  // output the content of the file
  while( !iter.equal(end_of_stream_iterator) )

  // use both operator++ and operator*
  cout << *iter++;

  cout << endl;

}
```

See Also

basic_streambuf(3C++), ***basic_istream***(3C++), ***ostreambuf_iterator***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 24.5.3

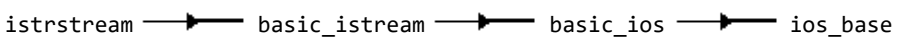
Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

istrstream



Summary

Reads characters from an array in memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
off_type
pos_type
traits

Member Functions

rdbuf()
str()

Synopsis

```
#include <strstream>
class istrstream
: public basic_istream<char>
```

Description

The class ***istrstream*** reads characters from an array in memory. It uses a private ***strstreambuf*** object to control the associated array object. It inherits from ***basic_istream<char>*** and therefore can use all the formatted and unformatted input functions.

This is a deprecated feature and might not be available in future versions.

Interface

```
class istrstream : public basic_istream<char> {

 public:

  typedef char_traits<char>             traits;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef char                          char_type;
  typedef typename traits::int_type     int_type;
  typedef typename traits::pos_type     pos_type;
  typedef typename traits::off_type     off_type;

  explicit istrstream(const char *s);
  istrstream(const char *s, streamsize n);
  explicit istrstream(char *s);
  istrstream(char *s, streamsize n);

  virtual ~istrstream();

  strstreambuf *rdbuf() const;
  char *str();

};
```

Types

char_type

The type char_type is a synonym of type char.

int_type

The type int_type is a synonym of type traits::in_type.

off_type

The type off_type is a synonym of type traits::off_type.

pos_type

The type pos_type is a synonym of type traits::pos_type.

traits

The type traits is a synonym of type char_traits<char>.

Constructors

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

Constructs an object of class istrstream, initializing the base class basic_istream<char> with the associated strstreambuf object. The strstreambuf object is initialized by calling strstreambuf(s,0), where s designates the first element of an NTBS (null terminated byte string).

```
explicit istrstream(const char* s, streamsize n);
explicit istrstream(char* s, streamsize n);
```

Constructs an object of class istrstream, initializing the base class basic_istream<char> with the associated strstreambuf object. The strstreambuf object is initialized by calling strstreambuf(s,n), where s designates the first element of an array whose length is n elements and n is greater than zero.

Destructors

```
virtual ~istrstream();
```

Destroys an object of class istrstream.

Member Functions

```
char*
str();
```

Returns a pointer to the underlying array object that may be null.

©Copyright 1999 Rogue Wave Software, Inc.

```
strstreambuf*
rdbuf() const;
```

Returns a pointer to the private `strstreambuf` object associated with the stream.

Example

```
//
// stdlib/examples/manual/istrstream.cpp
//
#include<iostream>
#include<strstream>

void main ( )
{
  using namespace std;

  const char* p="C'est pas l'homme qui prend la mer, ";
  const char* s="c'est la mer qui prend l'homme";

  // create an istrstream object and initialize
  // the underlying strstreambuf with p
  istrstream in_first(p);

  // create an istrstream object and initialize
  // the underlying strstreambuf with s
  istrstream in_next(s);

  // create an ostrstream object
  ostrstream out;

  // output the content of in_first and
  // in_next to out
  out << in_first.rdbuf() << in_next.str();

  // output the content of out to stdout
  cout << endl << out.rdbuf() << endl;


  // output the content of in_first to stdout
  cout << endl << in_first.str();

  // output the content of in_next to stdout
  cout << endl << in_next.rdbuf() << endl;

}
```

See Also

char_traits(3C++), *ios_base*(3C++), *basic_ios*(3C++), *strstreambuf*(3C++), *ostrstream*(3C++), *strstream*(3c++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Annex D Compatibility features Section D.6.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

isupper

`Locale Convenience Function`

Summary

Determines whether a character is upper case.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isupper (charT c, const locale& loc) const;
```

Description

The ***isupper*** function returns `true` if the character passed as a parameter is upper case. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

isxdigit

`Locale Convenience Function`

Summary

Determines whether a character is a hexadecimal digit.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>

template <class charT>
bool isxdigit (charT c, const locale& loc) const;
```

Description

The ***isxdigit*** function returns `true` if the character passed as a parameter is a hexadecimal digit. Otherwise the function returns `false`. The check is made using the `ctype` facet from the locale parameter.

See Also

other is functions, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

iter_swap

`Algorithm`

Summary

Exchanges values in two locations.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap (ForwardIterator1, ForwardIterator2);
```

Description

The ***iter_swap*** algorithm exchanges the values pointed to by the two iterators `a` and `b`.

Example

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

 int main ()
 {
   int d1[] = {6, 7, 8, 9, 10, 1, 2, 3, 4, 5};
   //
   // Set up a vector.
   //
   vector<int> v(d1+0, d1+10);
   //
   // Output original vector.
   //
   cout << "For the vector: ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // Swap the first five elements with the
   // last five elements.
   //
   swap_ranges(v.begin(), v.begin()+5, v.begin()+5);
   //
   // Output result.
   //
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   cout << endl << endl
  << "Swapping the first 5 elements with the last 5 gives: "
        << endl << "     ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // Now an example of iter_swap -- swap first and
   // last elements.
   //
   iter_swap(v.begin(), v.end()-1);
   //
   // Output result.
   //
   cout << endl << endl
        << "Swapping the first and last elements gives: "
        << endl << "     ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
For the vector: 6 7 8 9 10 1 2 3 4 5
Swapping the first five elements with the last five gives:
     1 2 3 4 5 6 7 8 9 10
Swapping the first and last elements gives:
     10 2 3 4 5 6 7 8 9 1
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Iterators*,*swap*, *swap_ranges

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

iterator

`Iterator base class`

Summary

A base iterator class.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class Category, class T,
          class Distance = ptrdiff_t, class Pointer = T*,
          class Reference = T&>
struct iterator
{
   typedef T value_type;
   typedef Distance difference_type;
   typedef Pointer pointer;
   typedef Reference reference;
   typedef Category iterator_category;
};
```

Description

The ***iterator*** structure is a base class from which all other iterator types can be derived. This structure defines an interface that consists of five public types: `value_type`, `difference_type`, `pointer`, `reference`, and `iterator_category`. These types are used primarily by classes derived from ***iterator*** and by the ***iterator_traits*** class.

See the ***iterators*** section for a description of iterators and the capabilities associated with various types.

See Also

iterator_traits

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

__iterator_category

`Iterator primitive`

Summary

Determines the category to which an iterator belongs. This function is now obsolete. It is included for backward compatibility and to support compilers that do not include partial specialization.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class Category, class T, class Distance,
          class Pointer, class Reference>
inline Category __iterator_category
  (const iterator<Category, T, Distance, Pointer,
   Reference>&);
template <class T>
inline random_access_iterator_tag __iterator_category
  (const T*)
```

Description

The ***__iterator_category*** family of function templates allows you to determine the category to which any iterator belongs. The first function takes an iterator of a specific type and returns the tag for that type. The last takes a `T*` and returns `random_access_iterator_tag`.

Tag Types

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

The ***__iterator_category*** function is particularly useful for improving the efficiency of algorithms. An algorithm can use this function to select the most efficient implementation an iterator is capable of handling without sacrificing the ability to work with a wide range of iterator types. For instance, both the `advance` and `distance` primitives use ***__iterator_category*** to maximize their efficiency by using the tag returned from ***iterator_category*** to select from one of several different auxiliary functions. Because this is a compile time selection, use of this primitive incurs no significant runtime overhead.

__iterator_category is typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
```

©Copyright 1999 Rogue Wave Software, Inc.

```
{
  __foo(begin,end,__iterator_category(first));
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           input_iterator_tag>
{
  // Most general implementation
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           bidirectional_iterator_tag>
{
  // Implementation takes advantage of bi-directional
  // capability of the iterators
}

...etc.
```

See the ***iterator*** section for a description of iterators and the capabilities associated with each type of iterator tag.

See Also

Other iterator primitives: ***__value_type***, ***__distance_type***, ***distance***, ***advance***, ***iterator***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

iterator_traits

`Iterator traits class`

Summary

Returns basic information about an iterator.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
template <class Iterator> struct iterator_traits
{
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::difference_type
                              difference_type;
   typedef typename Iterator::pointer pointer;
   typedef typename Iterator::reference reference;
   typedef typename Iterator::iterator_category
                              iterator_category;
};

// Specialization
template <class T> struct iterator_traits<T*>
{
   typedef T value_type;
   typedef ptrdiff_t difference_type;
   typedef T* pointer;
   typedef T& reference;
   typedef random_access_iterator_tag iterator_category;
};
```

Description

The ***iterator_traits*** template and specialization allows algorithms to access information about a particular iterator in a uniform way. The template requires either an iterator with a basic interface consisting of the types `value_type`, difference_type, `pointer`, `reference`, and `iterator_category`, or it requires a specialization for the iterator. The library includes one specialization (partial) to handle all pointer iterator types.

iterator_traits are used within algorithms to create local variables of either the type pointed to by the iterator or of the iterator's distance type. The traits also improve the efficiency of algorithms by making use of knowledge about basic iterator categories provided by the `iterator_category` member. An algorithm can use this "tag" to select the most efficient implementation an iterator is capable of handling without sacrificing the ability to work with a wide range of iterator types. For instance, both the `advance` and `distance` primitives use `iterator_category` to maximize their efficiency

©Copyright 1999 Rogue Wave Software, Inc.

by using the tag to select from one of several different auxiliary functions. The `iterator_category` must therefore be one of the iterator tags included by the library.

Tag Types

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

`iterator_traits::iterator_category` is typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,
        iterator_traits<Iterator>::iterator_category);
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           input_iterator_tag>
{
  // Most general implementation
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           bidirectional_iterator_tag>
{
  // Implementation takes advantage of bi-directional
  // capability of the iterators
}

...etc.
```

See the ***iterator*** section for a description of iterators and the capabilities associated with each type of iterator tag.

Warnings

If your compiler does not support partial specialization, then this template and specialization are not available to you. Instead you need to use the `__distance_type`, `__value_type`, and `__iterator_category` families of function templates. The Rogue Wave **Standard C++ Library** also includes alternate implementations of the distance, advance, and count functions when partial specialization is not supported by a particular compiler.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

__value_type, ***__distance_type***, ***__iterator_category***, ***distance***, ***advance***, ***iterator***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Iterators

Summary

Pointer generalizations for traversal and modification of collections.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. The illustration below displays the five iterator categories defined by the standard library, and shows their hierarchical relationship. Because standard library iterator categories are hierarchical, each category includes all the requirements of the categories above it.

Because iterators are used to traverse and access containers, the nature of the container determines the type of iterator it generates. Also, because algorithms require specific iterator types as arguments, it is iterators that, for the most part, determine which standard library algorithms can be used with which standard library containers.

To conform to the C++ standard, all container and sequence classes must include their own iterator types. Each iterator may be a class defined within the container or may be a simple pointer, whichever is appropriate.

Containers and sequences must also include `const` iterators that point to the beginning and end of their collections. These may be accessed using the class members, `begin()` and `end()`.

Because the semantics of iterators are a generalization of the semantics of C++ pointers, every template function that takes iterators also works using C++ pointers to contiguous memory sequences. For example, both of the following uses of the generic algorithm `count` are valid:

```
list<int> 1;
count(1.begin(), 1.end());
int buf[4]={1,2,3,4};
count(buf, buf+4);
```

Iterators may be constant or mutable depending upon whether the result of the `operator*` behaves as a reference or as a reference to a constant. Constant iterators cannot satisfy the requirements of an `output_iterator`.

Every iterator type guarantees that there is an iterator value that points past the last element of a corresponding container. This value is called the *past-the-end value*. No guarantee is made that this value is dereferenceable.

Every function included in an iterator is required to be realized in amortized constant time.

Key to Iterator Requirements

©Copyright 1999 Rogue Wave Software, Inc.

The following key pertains to the iterator requirements listed below:

`a and b`	values of type `X`
`n`	value representing a `distance` between two iterators
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Input Iterators

The following expressions must be valid for input iterators:

`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	a == b implies `*a == *b`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++`	returns type `T`

For input iterators, `a == b` does not imply that `++a == ++b`.

Algorithms using input iterators should be single pass algorithms. They should not pass through the same iterator twice.

The value of type `T` does not have to be an `lvalue`.

Requirements for Output Iterators

The following expressions must be valid for output iterators:

`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`*a = t`	result is not used
`++r`	returns `X&`
`r++`	return value convertible to `const X&`
`*r++ = t`	result is not used

The only valid use for the `operator*` is on the left hand side of the assignment statement.

Algorithms using output iterators should be single pass algorithms. They should not pass through the same iterator twice.

Requirements for Forward Iterators

The following expressions must be valid for forward iterators:

`X u`	u might have a singular value
`X()`	X() might be singular
`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to `const X&`

©Copyright 1999 Rogue Wave Software, Inc.

`*r++`	returns `T&`

Forward iterators have the condition that `a == b` implies `*a== *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

Requirements for Bidirectional Iterators

A bidirectional iterator must meet all the requirements for forward iterators. In addition, the following expressions must be valid:

`--r`	returns `X&`
`r--`	return value convertible to `const X&`
`*r--`	returns `T`

Requirements for Random Access Iterators

A random access iterator must meet all the requirements for bidirectional iterators. In addition, the following expressions must be valid:

`r += n`	Semantics of `--r` or `++r` `n` times depending on the sign of n
`a + n, n + a`	returns type `X`
`r -= n`	returns `X&`, behaves as `r += -n`
`a - n`	returns type `X`
`b - a`	returns `distance`
`a[n]`	*(a+n), return value convertible to `T`
`a < b`	total ordering relation
`a > b`	total ordering relation opposite to `<`
`a <= b`	!(a > b)
`a >= b`	!(a < b)

All relational operators return a value convertible to `bool`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

less

`Function Object`

Summary

A binary function object that returns `true` if its first argument is less than its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
template <class T>
struct less : public binary_function<T, T, bool>;
```

Description

less is a binary function object. Its `operator()` returns `true` if `x` is less than `y`. You can pass a ***less*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***less*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), less<int>());
```

After this call to ***transform***, `vecResult(n)` contains a `1` if `vec1(n)` was less than `vec2(n)` or a `0` if `vec1(n)` was greater than or equal to `vec2(n)`.

Interface

```
template <class T>
struct less : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

less_equal

`Function Object`

Summary

A binary function object that returns `true` if its first argument is less than or equal to its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
template <class T>
struct less_equal : public binary_function<T, T, bool>;
```

Description

less_equal is a binary function object. Its `operator()` returns `true` if `x` is less than or equal to `y`. You can pass a ***less_equal*** object to any algorithm that requires a binary function. For example, the ***sort*** algorithm can accept a binary function as an alternate comparison object to sort a sequence. ***less_equal*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
.
.
.
sort(vec1.begin(), vec1.end(),less_equal<int>());
```

After this call to ***sort***, `vec1` is sorted in ascending order.

Interface

```
template <class T>
struct less_equal : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

lexicographical_compare

`Algorithm`

Summary

Compares two ranges lexicographically.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2>
 bool
 lexicographical_compare(InputIterator1 first,
                         InputIterator2 last1,
                         InputIterator2 first2,
                         InputIterator last2);

template <class InputIterator1, class InputIterator2,
          class Compare>
 bool
 lexicographical_compare(InputIterator1 first,
                         InputIterator2 last1,
                         InputIterator2 first2,
                         InputIterator last2, Compare comp);
```

Description

The ***lexicographical_compare*** functions compare each element in the range `[first1, last1)` to the corresponding element in the range `[first2, last2)` using iterators `i` and `j`.

The first version of the algorithm uses `operator<` as the default comparison operator. It immediately returns `true` if it encounters any pair in which `*i` is less than `*j`, and immediately returns `false` if `*j` is less than `*i`. If the algorithm reaches the end of the first sequence before reaching the end of the second sequence, it also returns `true`.

The second version of the function takes an argument `comp` that defines a comparison function that is used in place of the default `operator<`.

The ***lexicographical_compare*** functions can be used with all the datatypes included in the standard library.

Complexity

lexicographical_compare performs at most `min((last1 - first1), (last2 - first2))` applications of the comparison function.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// lex_comp.cpp
//
 #include <algorithm>
 #include <vector>
 #include <functional>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   int d1[5] = {1,3,5,32,64};
   int d2[5] = {1,3,2,43,56};

   // set up vector
   vector<int> v1(d1,d1 + 5), v2(d2,d2 + 5);

   // Is v1 less than v2 (I think not)
   bool b1 = lexicographical_compare(v1.begin(),
              v1.end(), v2.begin(), v2.end());

   // Is v2 less than v1 (yup, sure is)
   bool b2 = lexicographical_compare(v2.begin(),
            v2.end(), v1.begin(), v1.end(), less<int>());
   cout << (b1 ? "TRUE" : "FALSE") << " "
        << (b2 ? "TRUE" : "FALSE") << endl;

   return 0;
 }
```

Program Output

```
FALSE TRUE
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

limits

Refer to the ***numeric_limits*** section of this reference guide.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

list

`Container`

Summary

A sequence that supports bidirectional iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

assign()	max_size()	pop_front()	
back()	merge()	push_back()	
begin()	operator!=()	push_front()	
clear()	operator>()	rbegin()	sort()
empty()	operator>=()	remove()	splice()
end()	operator<()	remove_if()	swap()
erase()	operator<=()	rend()	unique()
front()	operator=()	resize()	
get_allocator()	operator==()	reverse()	
insert()	pop_back()	size()	

Synopsis

```
#include <list>
template <class T, class Allocator = allocator<T> >
class list;
```

Description

list<T,Allocator> is a type of sequence that supports bidirectional iterators. A ***list<T,Allocator>*** allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Constant time random access is not supported.

Any type used for the template parameter `T` must include the following (where `T` is the type, `t` is a `value` of `T` and `u` is a `const value` of `T`):

©Copyright 1999 Rogue Wave Software, Inc.

Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

Interface

```
template <class T, class Allocator = allocator<T> >
 class list {

public:

// typedefs

   class iterator;
   class const_iterator;
   typedef typename
           Allocator::reference        reference;
   typedef typename
           Allocator::const_reference  const_reference;
   typedef typename
           Allocator::size_type        size_type;
   typedef typename
           Allocator::difference_type  difference_type;
   typedef T value_type;
   typedef Allocator allocator_type;

   typedef typename std::reverse_iterator<iterator>
                        reverse_iterator;
   typedef typename std::reverse_iterator<const_iterator>
                        const_reverse_iterator;



// Construct/Copy/Destroy

   explicit list (const Allocator& = Allocator());
   explicit list (size_type);
   list (size_type, const T&, const Allocator& =
         Allocator())
   template <class InputIterator>
   list (InputIterator, InputIterator,
         const Allocator& = Allocator());
   list(const list<T, Allocator>& x);
   ~list();
   list<T,Allocator>& operator= (const list<T,Allocator>&);
   template <class InputIterator>
    void assign (InputIterator, InputIterator);
   void assign (size_type n, const T&);

   allocator_type get allocator () const;

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   bool empty () const;
   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize (size_type, T);

// Element Access
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

// Modifiers

   void push_front (const T&);
   void pop_front ();
   void push_back (const T&);
   void pop_back ();

   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
   template <class InputIterator>
    void insert (iterator, InputIterator, InputIterator);

   iterator erase (iterator);
   iterator erase (iterator, iterator);

   void swap (list<T, Allocator>&);
   void clear ();

// Special mutative operations on list

   void splice (iterator, list<T, Allocator>&);
   void splice (iterator, list<T, Allocator>&, iterator);
   void splice (iterator, list<T, Allocator>&, iterator,
                iterator);

   void remove (const T&);
   template <class Predicate>
    void remove_if (Predicate);

   void unique ();
   template <class BinaryPredicate>
    void unique (BinaryPredicate);

   void merge (list<T, Allocator>&);
   template <class Compare>
    void merge (list<T, Allocator>&, Compare);

   void sort ();
   template <class Compare>
    void sort (Compare);

   void reverse();
};



// Non-member List Operators

template <class T, class Allocator>
 bool operator== (const list<T, Allocator>&,
                  const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator!= (const list<T, Allocator>&,
                  const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator< (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator> (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator<= (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator>= (const list<T, Allocator>&,
                 const list<T, Allocator>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// Specialized Algorithms

template <class T, class Allocator>
void swap (list<T,Allocator>&, list<T, Allocator>&);
```

Constructors

```
explicit list(const Allocator& alloc = Allocator());
```

Creates a list of zero elements. The list uses the allocator `alloc` for all storage management.

```
explicit list(size_type n);
```

Creates a list of length `n`, containing `n` copies of the default value for type `T`. `T` must have a default constructor. The list uses the allocator Allocator() for all storage management.

```
list(size_type n, const T& value,
      const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of `value`. The list uses the allocator `alloc` for all storage management.

```
template <class InputIterator>
list(InputIterator first, InputIterator last,
      const Allocator& alloc = Allocator());
```

Creates a list of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The list uses the allocator `alloc` for all storage management.

```
list(const list<T, Allocator>& x);
```

Creates a copy of `x`.

Destructors

```
~list();
```

Releases any allocated memory for this list.

Assignment Operators

```
list<T, Allocator>&
operator=(const list<T, Allocator>& x)
```

Erases all elements in self, then inserts into self a copy of each element in `x`. Returns a reference to `*this`.

Allocators

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

Returns a bidirectional iterator that points to the first element.

```
const_iterator
begin() const;
```

Returns a constant bidirectional iterator that points to the first element.

```
iterator
end();
```

Returns a bidirectional iterator that points to the past-the-end value.

©Copyright 1999 Rogue Wave Software, Inc.

```
const_iterator
end() const;
```

Returns a constant bidirectional iterator that points to the past-the-end value.

```
reverse_iterator
rbegin();
```

Returns a bidirectional iterator that points to the past-the-end value.

```
const_reverse_iterator
rbegin() const;
```

Returns a constant bidirectional iterator that points to the past-the-end value.

```
reverse_iterator
rend();
```

Returns a bidirectional iterator that points to the first element.

```
const_reverse_iterator
rend() const;
```

Returns a constant bidirectional iterator that points to the first element.

Member Functions

```
template <class InputIterator>
void
assign(InputIterator first, InputIterator last);
```

Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

```
void
assign(size_type n, const T& t);
```

Erases all elements contained in self, then inserts `n` instances of the `value` of `t`.

```
reference
back();
```

Returns a `reference` to the last element.

```
const_reference
back() const;
```

Returns a constant reference to the last element.

```
void
clear();
```

Erases all elements from the list.

```
bool
empty() const;
```

Returns `true` if the `size` is zero.

```
iterator
erase(iterator position);
```

Removes the element pointed to by `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
iterator
erase(iterator first, iterator last);
```

Removes the elements in the range (`first, last`). Returns an iterator pointing to the element following the element following the last deleted element, or `end()` if there were no elements after the deleted range.

©Copyright 1999 Rogue Wave Software, Inc.

```
reference
front();
```

Returns a reference to the first element.

```
const_reference
front() const;
```

Returns a constant reference to the first element.

```
iterator
insert(iterator position, const T& x);
```

Inserts `x` before `position`. Returns an iterator that points to the inserted `x`.

```
void
insert(iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` before `position`.

```
template <class InputIterator>
void
insert(iterator position, InputIterator first,
        InputIterator last);
```

Inserts copies of the elements in the range `[first, last)` before `position`.

```
size_type
max_size() const;
```

Returns `size()` of the largest possible list.

```
void
merge(list<T, Allocator>& x);
```

Merges a sorted `x` with a sorted self using `operator<`. For equal elements in the two lists, elements from self always precede the elements from `x`. The `merge` function leaves `x` empty.

```
template <class Compare>
void
merge(list<T, Allocator>& x, Compare comp);
```

Merges a sorted `x` with sorted self using a compare function object, `comp`. For identical elements in the two lists, elements from self always precede the elements from `x`. The `merge` function leaves `x` empty.

```
void
pop_back();
```

Removes the last element.

```
void
pop_front();
```

Removes the first element.

```
void
push_back(const T& x);
```

Appends a copy of `x` to the end of the list.

```
void
push_front(const T& x);
```

Appends a copy of `x` to the front of the list.

```
void
remove(const T& value);
template <class Predicate>
void
remove_if(Predicate pred);
```

©Copyright 1999 Rogue Wave Software, Inc.

Removes all elements in the list referenced by the list iterator `i` for which `*i == value` or `pred(*i) == true`, whichever is applicable. This is a stable operation. The relative order of list items that are not removed is preserved.

```
void
resize(size_type sz);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, `sz-size()` copies of the default value of type `T` are inserted at the end of the list. If the new size is smaller than the current capacity, then the list is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken. Type `T` must have a default constructor.

```
void
resize(size_type sz, T c);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, `sz-size() c`'s are inserted at the end of the list. If the new size is smaller than the current capacity, then the list is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken.

```
void
reverse();
```

Reverses the order of the elements.

```
size_type
size() const;
```

Returns the number of elements.

```
void
sort();
```

Sorts self according to the `operator<`. `sort` maintains the relative order of equal elements.

```
template <class Compare>
void
sort(Compare comp);
```

Sorts self according to a comparison function object, `comp`. This is also a stable sort.

```
void
splice(iterator position, list<T, Allocator>& x);
```

Inserts `x` before `position`, leaving `x` empty.

```
void
splice(iterator position, list<T, Allocator>& x,
        iterator i);
```

Moves the elements pointed to by iterator `i` in `x` to self, inserting it before `position`. The element is removed from `x`.

```
void
splice(iterator position, list<T, Allocator >& x,
        iterator first, iterator last);
```

Moves the elements in the range `[first, last)` in `x` to self, inserting them before `position`. The elements in the range `[first, last)` are removed from `x`.

```
void
swap(list <T, Allocator>& x);
```

Exchanges self with `x`.

```
void
unique();
```

Erases copies of consecutive repeated elements leaving the first occurrence.

```
template <class BinaryPredicate>
void
unique(BinaryPredicate binary_pred);
```

©Copyright 1999 Rogue Wave Software, Inc.

Erases consecutive elements matching a `true` condition of the `binary_pred`. The first occurrence is not removed.

Non-member Operators

```
template <class T, class Allocator>
bool operator==(const list<T, Allocator>& x,
                const list<T, Allocator>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool operator!=(const list<T, Allocator>& x,
                const list<T, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class T, class Allocator>
bool operator<(const list<T, Allocator>& x,
               const list<T,Allocator>& y);
```

Returns `true` if the sequence defined by the elements contained in `x` is lexicographically less than the sequence defined by the elements contained in `y`.

```
template <class T, class Allocator>
bool operator>(const list<T, Allocator>& x,
               const list<T,Allocator>& y);
```

Returns `y < x`.

```
template <class T, class Allocator>
bool operator<=(const list<T, Allocator>& x,
               const list<T,Allocator>& y);
```

Returns `!(y < x)`.

```
template <class T, class Allocator>
bool operator>=(const list<T, Allocator>& x,
               const list<T,Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class T, class Allocator>
void swap(list<T, Allocator>& a, list<T, Allocator>& b);
```

Swaps the contents of `a` and `b`.

Example

```
//
// list.cpp
//
 #include <list>
 #include <string>
 #include <iostream>
 using namespace std;
 // Print out a list of strings
 ostream& operator<<(ostream& out, const list<string>& l)
 {

   copy(l.begin(), l.end(),
        ostream_iterator<string,char>(cout," "));
   return out;
 }
 int main(void)
 {

   // create a list of critters
   list<string> critters;
   int i;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // insert several critters
   critters.insert(critters.begin(),"antelope");
   critters.insert(critters.begin(),"bear");
   critters.insert(critters.begin(),"cat");

   // print out the list
   cout << critters << endl;

   // Change cat to cougar
   *find(critters.begin(),critters.end(),"cat") = "cougar";
   cout << critters << endl;

   // put a zebra at the beginning
   // an ocelot ahead of antelope
   // and a rat at the end
   critters.push_front("zebra");
   critters.insert(find(critters.begin(),critters.end(),
                   "antelope"),"ocelot");
   critters.push_back("rat");
   cout << critters << endl;

   // sort the list (Use list's sort function since the
   // generic algorithm requires a random access iterator
   // and list only provides bidirectional)
   critters.sort();
   cout << critters << endl;

   // now let's erase half of the critters
   int half = critters.size() >> 1;
   for(i = 0; i < half; ++i) {
     critters.erase(critters.begin());
   }
   cout << critters << endl;
   return 0;
 }
```

Program Output

```
cat bear antelope
cougar bear antelope
zebra cougar bear ocelot antelope rat
antelope bear cougar ocelot rat zebra
ocelot rat zebra
```

Warnings

Member function templates are used in all containers included in the Standard Template Library. An example of this feature is the constructor for ***list<T, Allocator>*** that takes two templatized iterators:

```
template <class InputIterator>
list (InputIterator, InputIterator,
      const Allocator& = Allocator());
```

list also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a list in the following two ways:

```
int intarray[10];
list<int> first_list(intarray,intarray + 10);
list<int> second_list(first_list.begin(),first_list.end());
```

But not this way:

```
list<long> long_list(first_list.begin(),first_list.end());
```

since the `long_list` and `first_list` are not the same type.

©Copyright 1999 Rogue Wave Software, Inc.

Additionally, ***list*** includes a `merge` function of this type.

```
template <class Compare> void merge (list<T, Allocator>&,
  Compare);
```

This function allows you to specify a compare function object to be used in merging two lists. In this case, a substitute function is not included with the merge that uses the `operator<` as the default. Thus, if your compiler does not support member function templates, all list merges use `operator<`.

Also, many compilers do not support default template arguments. If your compiler is one of these, you always need to supply the `Allocator` template argument. For instance, you have to write:

```
list<int, allocator<int> >
```

instead of:

```
list<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Containers***, ***Iterators***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

locale

`Localization Container`

Summary

A localization class containing a polymorphic set of facets.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

category
facet
id

Member Functions

classic()
global()
name()
operator!=()
operator()()
operator=()
operator==()

Synopsis

```
#include <locale>
class locale;
```

Description

locale is a localization interface and a set of indexed facets, each of which covers one particular localization issue. The default locale object is constructed on the "C" locale. Locales can also be constructed on named locales.

A calling program can determine whether a particular facet is contained in a locale by using the `has_facet` function, and the program can obtain a reference to that facet with the `use_facet` function. These are not member functions, but instead take a locale object as an argument.

locale has several important characteristics.

©Copyright 1999 Rogue Wave Software, Inc.

First, successive calls to member functions always return the same result. This allows a calling program to safely cache the results of a call.

Only a locale constructed from a name, from parts of two named locales, or from a stream, has a name. All other locales are unnamed. Only named locales may be compared for equality. An unnamed locale is equal only to itself.

Interface

```
class locale {
public:
  // types:
  class facet;
  class id;
  typedef int category;
  static const category   none, collate, ctype, monetary,
                          numeric, time, messages,
                          all = collate | ctype | monetary |
                                numeric | time | messages;
  // construct/copy/destroy:
  locale() throw()
  locale(const locale&) throw()
  explicit locale(const char*);
  locale(const locale&, const char*, category);
  template <class Facet> locale(const locale&, Facet*);
  locale(const locale&, const locale&, category);
  ~locale() throw();  // non-virtual
  const locale& operator=(const locale&) throw();
  template <class Facet> locale combine (const locale&);
  // locale operations:
  basic_string<char>                  name() const;
  bool operator==(const locale&) const;
  bool operator!=(const locale&) const;
  template <class charT, class Traits, class Allocator>
    bool operator()(const basic_string<charT,Traits,
                                       Allocator>&,
                    const basic_string<charT,Traits,
                                       Allocator>&)
                    const;
  // global locale objects:
  static       locale global(const locale&);
  static const locale& classic();
};

class locale::facet {
protected:
  explicit facet(size_t refs = 0);
  virtual ~facet();
private:
  facet(const facet&);          // not defined
  void operator=(const facet&); // not defined
};

class locale::id {
public:
  id();
private:
  void operator=(const id&); // not defined
  id(const id&);             // not defined
};
```

Types

`category`

Standard facets fall into eight broad categories. These are: `none`, `collate`, `ctype`, `monetary`, `numeric`, `time`, `messages`, and `all`. `all` is a combination of all the other categories except `none`. Bitwise operations may be applied to combine or screen these categories. For instance, `all` is defined as:

```
(collate | ctype | monetary | numeric | time | messages)
```

locale member functions that take a category argument must be included with one of the above values or with one of the constants from the old C locale (for example, `LC_CTYPE`).

©Copyright 1999 Rogue Wave Software, Inc.

facet

Base class for facets. This class exists primarily to allow for reference counting services to derived classes. All facets must derive from it, either directly or indirectly (for example, `facet -> ctype<char> -> my_ctype`).

If the `refs` argument to the constructor is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

Copy construction and assignment of this class are not allowed.

id

Type used to index facets in the ***locale*** container. Every facet must contain a member of this type.

Copy construction and assignment of this type are not allowed.

Constructors

```
locale()
  throw()
```

Constructs a default ***locale*** object. This locale is the same as the last argument passed to `locale::global()`, or, if that function has not been called, the locale is the same as the classic "C" locale.

```
locale(const locale& other)
  throw()
```

Constructs a copy of the locale argument `other`.

```
explicit locale(const char* std_name);
```

Constructs a ***locale*** object on the named locale indicated by `std_name`. Throws a ***runtime_error*** exception if `std_name` is not a valid locale name.

```
locale(const locale& other, const char* std_name,
       category cat);
```

Constructs a ***locale*** object that is a copy of `other`, except for the facets that are in the category specified by `cat`. These facets are obtained from the named locale identified by `std_name`. Throws a `runtime_error` exception if `std_name` is not a valid locale name.

The resulting locale has a name only if `other` has a name.

```
template <class Facet>
locale(const locale& other, Facet* f);
```

Constructs a ***locale*** object that is a copy of `other`, except for the facet of type `Facet`. Unless `f` is null, it is used to supply the missing facet. Otherwise the facet comes from `other` as well.

Note that the resulting locale does not have a name.

```
locale(const locale& other, const locale& one,
       category cat);
```

Constructs a ***locale*** object that is a copy of `other`, except for facets that are in the category specified by category argument `cat`. These missing facets are obtained from the other locale argument, `one`.

Note that the resulting locale has a name only if both `other` and `one` have names.

Destructors

```
~locale();
```

Destroys the locale.

Public Member Operators

©Copyright 1999 Rogue Wave Software, Inc.

```
const locale&
operator=(const locale& other) throw();
```

Replaces *this with a copy of other. Returns *this.

```
template <class Facet>
locale combine(const locale& other);
```

Returns a locale object that is a copy of *this, except for the facet of type Facet, which is taken from other. If other does not contain a facet of type Facet, a ***runtime_error*** exception is thrown.

Note that the returned locale does not have a name.

```
bool
operator==(const locale& other) const;
```

Returns true if both other and *this are the same object, if one is a copy of another, or if both have the same name. Otherwise returns false.

```
bool
operator!=(const locale& other) const;
```

Returns !(*this == other)

```
template <class charT,Traits>
bool
operator()(const basic_string<charT,Traits>& s1,
           const basic_string<charT,Traits>& s2) const;
```

This operator allows a ***locale*** object to be used as a comparison object for comparing two strings. Returns the result of comparing the two strings using the compare member function of the collate<charT> facet contained in *this. Specifically, this function returns the following:

```
use_facet< collate<charT> >(*this).compare(s1.data(),
s1.data()+s1.size(), s2.data(),
s2.data()+s2.size()) < 0;
```

This allows a locale to be used with standard algorithms, such as sort, for localized comparison of strings.

Public Member Functions

```
basic_string<char>
name() const;
```

Returns the name of this locale, if it has one; otherwise returns the string "*".

Static Public Member Functions

```
static const locale&
```

```
classic();
```

Returns a locale with the semantics of the classic "C" locale.

```
static locale
```

```
global(const locale& loc);
```

Sets the global locale to loc. This causes future uses of the default constructor for locale to return a copy of loc. If loc has a name, this function has the further effect of calling std::setlocale(LC_ALL,loc.name().c_str());. Returns the previous value of locale().

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// locale.cpp
//

 #include <string>
 #include <vector>
 #include <iostream>
 #include "codecvte.h"

 int main ()
 {
  using namespace std;

  locale loc;  // Default locale

  // Construct new locale using default locale plus
  // user defined codecvt facet
  // This facet converts from ISO Latin
  // Alphabet No. 1 (ISO 8859-1) to
  // U.S. ASCII code page 437
  // This facet replaces the default for
  // codecvt<char,char,mbstate_t>
  locale my_loc(loc,new ex_codecvt);

  // imbue modified locale onto cout
  locale old = cout.imbue(my_loc);
  cout << "A \x93 jolly time was had by all" << endl;

  cout.imbue(old);
  cout << "A jolly time was had by all" << endl;

  // Create a vector of strings
  vector<string,allocator<void> > v;
  v.insert(v.begin(),"antelope");
  v.insert(v.begin(),"bison");
  v.insert(v.begin(),"elk");

  copy(v.begin(),v.end(),
       ostream_iterator<string,char,
                 char_traits<char> >(cout," "));
  cout << endl;

  // Sort the strings using the locale as a comparitor
  sort(v.begin(),v.end(),loc);

  copy(v.begin(),v.end(),
       ostream_iterator<string,char,
                char_traits<char> >(cout," "));

  cout << endl;
  return 0;
}
```

See Also

facets, *has_facet*, *use_facet*, ***specific facet reference sections***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

logical_and

`Function Object`

Summary

A binary function object that returns `true` if both of its arguments are `true`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct logical_and : public binary_function<T, T, bool>;
```

Description

logical_and is a binary function object. Its `operator()` returns `true` if both `x` and `y` are `true`. You can pass a ***logical_and*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***logical_and*** is used in that algorithm in the following manner:

```
vector<bool> vec1;
vector<bool> vec2;
vector<bool> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), logical_and<bool>());
```

After this call to ***transform***, `vecResult(n)` contains a `1` (`true`) if both `vec1(n)` and `vec2(n)` are `true` or a `0` (`false`) if either `vec1(n)` or `vec2(n)` is `false`.

Interface

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<bool, allocator<bool> >
```

instead of:

```
vector<bool>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

logical_not

`Function Object`

Summary

A unary function object that returns `true` if its argument is `false`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct logical_not : unary_function<T, bool> ;
```

Description

logical_not is a unary function object. Its `operator()` returns `true` if its argument is `false`. You can pass a ***logical_not*** object to any algorithm that requires a unary function. For example, the ***replace_if*** algorithm replaces an element with another value if the result of a unary operation is `true`. ***logical_not*** is used in that algorithm in the following manner:

```
vector<int> vec1;
.
.
.
void replace_if(vec1.begin(), vec1.end(),
                logical_not<int>(),1);
```

This call to ***replace_if*** replaces all zeros in the `vec1` with `1`.

Interface

```
template <class T>
struct logical_not : unary_function<T, bool> {
  bool operator() (const T&) const;
};
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you have to write:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects, ***unary_function***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

logical_or

`Function Object`

Summary

A binary function object that returns `true` if either of its arguments are `true`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct logical_or : binary_function<T, T, bool> ;
```

Description

logical_or is a binary function object. Its `operator()` returns `true` if either `x` or `y` are `true`. You can pass a ***logical_or*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***logical_or*** is used in that algorithm in the following manner:

```
vector<bool> vec1;
vector<bool> vec2;
vector<bool> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), logical_or<bool>());
```

After this call to ***transform***, `vecResult(n)` contains a `1` (`true`) if either `vec1(n)` or `vec2(n)` is `true` or a `0` (`false`) if both `vec1(n)` and `vec2(n)` are `false`.

Interface

```
template <class T>
struct logical_or : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<bool, allocator<bool> >
```

instead of:

```
vector<bool>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

lower_bound

`Algorithm`

Summary

Determine the first valid position for an element in a sorted container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
template <class ForwardIterator, class T>
 ForwardIterator lower_bound(ForwardIterator first,
                             ForwardIterator last,
                             const T& value);

 template <class ForwardIterator, class T, class Compare>
  ForwardIterator lower_bound(ForwardIterator first,
                              ForwardIterator last,
                              const T& value, Compare comp);
```

Description

The ***lower_bound*** algorithm compares a supplied `value` to elements in a sorted container and returns the first position in the container that `value` can occupy without violating the container's ordering. There are two versions of the algorithm. The first uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version lets you include a function object of type `Compare`, and assumes that `Compare` is the function used to sort the sequence. The function object must be a binary predicate.

lower_bound's return value is the iterator for the first element in the container that is *greater than or equal to* `value`, or, when the comparison operator is used, the first element that does not satisfy the comparison function. Formally, the algorithm returns an iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold:

```
*j < value
```

or

```
comp(*j, value) == true
```

Complexity

lower_bound performs at most `log(last - first) + 1` comparisons.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// ul_bound.cpp
//
 #include <vector>
 #include <algorithm>
 #include <iostream>
 using namespace std;

 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};

   // Set up a vector
   vector<int> v1(d1,d1 + 11);

   // Try lower_bound variants
   iterator it1 = lower_bound(v1.begin(),v1.end(),3);
   // it1 = v1.begin() + 4

   iterator it2 =
       lower_bound(v1.begin(),v1.end(),2,less<int>());
   // it2 = v1.begin() + 4

   // Try upper_bound variants
   iterator it3 = upper_bound(v1.begin(),v1.end(),3);
   // it3 = vector + 5

   iterator it4 =
      upper_bound(v1.begin(),v1.end(),2,less<int>());
   // it4 = v1.begin() + 5

   cout << endl << endl
        << "The upper and lower bounds of 3: ( "
        << *it1 << " , " << *it3 << " ]" << endl;

   cout << endl << endl
        << "The upper and lower bounds of 2: ( "
        << *it2 << " , " << *it4 << " ]" << endl;

   return 0;
 }
```

Program Output

```
The upper and lower bounds of 3: ( 3 , 4 ]
The upper and lower bounds of 2: ( 2 , 3 ]
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

upper_bound, *equal_range*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

make_heap

`Algorithm`

Summary

Creates a heap.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
  void
  make_heap(RandomAccessIterator first,
            RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  make_heap(RandomAccessIterator first,
            RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** algorithm, or a new element can be added by the ***push_heap*** algorithm, in `O(logN)` time.

These properties make heaps useful as priority queues.

The heap algorithms use less than (`operator<`) as the default comparison. In all of the algorithms, an alternate comparison operator can be specified.

The first version of the ***make_heap*** algorithm arranges the elements in the range `[first, last)` into a heap using less than (`operator<`) to perform comparisons. The second version uses the comparison operator `comp` to perform the comparisons. Since the only requirements for a heap are the two listed above, `make_heap` is not required to do anything within the range `(first, last - 1)`.

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

This algorithm makes at most `3 * (last - first)` comparisons.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

   // Make heaps
   make_heap(v1.begin(),v1.end());
   make_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z) and v2 = (4,x,y,z)
   // Note that x, y and z represent the remaining
   // values in the container (other than 4).
   // The definition of the heap and heap operations
   // does not require any particular ordering
   // of these values.

   // Copy both vectors to cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now let's pop
   pop_heap(v1.begin(),v1.end());
   pop_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (3,x,y,4) and v2 = (3,x,y,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // And push
   push_heap(v1.begin(),v1.end());
   push_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z) and v2 = (4,x,y,z)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now sort those heaps
   sort_heap(v1.begin(),v1.end());
   sort_heap(v2.begin(),v2.end(),less<int>());
   // v1 = v2 = (1,2,3,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   return 0;
 }
```

Program Output

©Copyright 1999 Rogue Wave Software, Inc.

```
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

pop_heap, ***push_heap*** and ***sort_heap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

map

`Container`

Summary

An associative container with access to non-key values using unique keys. A ***map*** supports bidirectional iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

begin()	key_comp()	
clear()	lower_bound()	operator[]()
count()	max_size()	rbegin()
empty()	operator!=()	rend()
end()	operator>()	size()
equal_range()	operator>=()	swap()
erase()	operator<()	upper_bound()
find()	operator<=()	value_comp()
get_allocator()	operator=()	
insert()	operator==()	

Synopsis

```
#include <map>
template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<pair<const Key, T>> >
class map;
```

Description

map <Key, T, Compare, Allocator> gives fast access to stored values of type `T` that are indexed by unique keys of type ***Key***. The default operation for key comparison is the `<` operator.

map has bidirectional iterators that point to an instance of `pair<const Key x, T y>` where `x` is the key and `y` is the stored value associated with that key. The definition of ***map*** includes a `typedef` to this pair called `value_type`.

©Copyright 1999 Rogue Wave Software, Inc.

The types used for both the template parameters `Key` and `T` must include the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

The type used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<pair<const Key, T>> >
 class map {

public:

// types

   typedef Key key_type;
   typedef typename Allocator::pointer pointer;
   typedef typename Allocator::const_pointer const_pointer;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;




   typedef typename
           Allocator::reference        reference;
   typedef typename
           Allocator::const_reference  const_reference;

   class iterator;
   class const_iterator;

   typedef typename
           Allocator::size_type        size_type;
   typedef typename
           Allocator::difference_type  difference_type;

   typedef typename std::reverse_iterator<iterator>
                        reverse_iterator;
   typedef typename std::reverse_iterator<const_iterator>
                        const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type, bool>
   {
     friend class map<Key, T, Compare, Allocator>;

     protected :
       Compare comp;
       value_compare(Compare c): comp(c) {}
     public :
       bool operator() (const value_type&,
                        const value_type&) const;
   };

// Construct/Copy/Destroy

   explicit map (const Compare& = Compare(),
                 const Allocator& = Allocator ());
   template <class InputIterator>
    map (InputIterator, InputIterator,
         const Compare& = Compare(),
         const Allocator& = Allocator ());
   map (const map<Key, T, Compare, Allocator>&);
   ~map();
   map<Key, T, Compare, Allocator>&
    operator= (const map<Key, T, Compare, Allocator>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   allocator_type get_allocator () const;

// Iterators

   iterator begin();
   const_iterator begin() const;
   iterator end();
   const_iterator end() const;
   reverse_iterator rbegin();
   const_reverse_iterator rbegin() const;
   reverse_iterator rend();
   const_reverse_iterator rend() const;

// Capacity

   bool empty() const;
   size_type size() const;
   size_type max_size() const;

// Element Access

   mapped_type& operator[] (const key_type&);

// Modifiers

   pair<iterator, bool> insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);

   void erase (iterator);
   size_type erase (const key_type&);
   void erase (iterator, iterator);
   void swap (map<Key, T, Compare, Allocator>&);
   void clear();

// Observers

   key_compare key_comp() const;
   value_compare value_comp() const;

// Map operations

   iterator find (const key_value&);
   const_iterator find (const key_value&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const key_type&);
   pair<const_iterator, const_iterator>
     equal_range (const key_type&) const;
};

// Non-member Map Operators

template <class Key, class T, class Compare, class Allocator>
 bool operator== (const map<Key, T, Compare, Allocator>&,
                  const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator!= (const map<Key, T, Compare, Allocator>&,
                  const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator< (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator> (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator<= (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class Key, class T, class Compare, class Allocator>
 bool operator>= (const map<Key, T, Compare, Allocator>&,
                  const map<Key, T, Compare, Allocator>&);


// Specialized Algorithms

template <class Key, class T, class Compare, class Allocator>
 void swap (map<*Key,T,Compare,Allocator>&,
            map<Key,T,Compare,Allocator>&);
```

Constructors

```
explicit map(const Compare& comp = Compare(),
             const Allocator& alloc = Allocator());
```

Constructs an empty map that uses the relation `comp` to order keys, if it is supplied. The map uses the allocator `alloc` for all storage management.

```
template <class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& alloc = Allocator());
```

Constructs a map containing values in the range `[first, last)`. Creation of the new map is only guaranteed to succeed if the iterators `first` and `last` return values of type `pair<class Key, class Value>` and all values of `Key` in the `range[first, last)` are unique. The map uses the relation `comp` to order keys, and the allocator `alloc` for all storage management.

```
map(const map<Key,T,Compare,Allocator>& x);
```

Creates a new map by copying all pairs of `key` and `value` from `x`.

Destructors

```
~map();
```

Releases any allocated memory for this map.

Allocators

```
allocator_type get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

Returns an `iterator` pointing to the first element stored in the map. "First" is defined by the map's comparison operator, `Compare`.

```
const_iterator
begin() const;
```

Returns a `const_iterator` pointing to the first element stored in the map.

```
iterator
end();
```

Returns an `iterator` pointing to the last element stored in the map (in other words, the off-the-end value).

```
const_iterator
end() const;
```

Returns a `const_iterator` pointing to the last element stored in the map.

©Copyright 1999 Rogue Wave Software, Inc.

```
reverse_iterator
rbegin();
```

Returns a `reverse_iterator` pointing to the first element stored in the map. "First" is defined by the map's comparison operator, `Compare`.

```
const_reverse_iterator
rbegin() const;
```

Returns a `const_reverse_iterator` pointing to the first element stored in the map.

```
reverse_iterator
rend();
```

Returns a `reverse_iterator` pointing to the last element stored in the map (in other words, the off-the-end value).

```
const_reverse_iterator
rend() const;
```

Returns a `const_reverse_iterator` pointing to the last element stored in the map.

Member Operators

```
map<Key, T, Compare, Allocator>&
operator=(const map<Key, T, Compare, Allocator>& x);
```

Replaces the contents of `*this` with a copy of the map `x`.

```
mapped_type&
operator[](const key_type& x);
```

If an element with the key `x` exists in the map, then a reference to its associated value is returned. Otherwise the pair `x,T()` is inserted into the map and a reference to the default object `T()` is returned.

Member Functions

```
void
clear();
```

Erases all elements from the self.

```
size_type
count(const key_type& x) const;
```

Returns a `1` if a value with the key `x` exists in the map. Otherwise returns a `0`.

```
bool
empty() const;
```

Returns `true` if the map is empty, `false` otherwise.

```
pair<iterator, iterator>
equal_range (const key_type& x);
```

Returns the pair `(lower_bound(x), upper_bound(x))`.

```
pair<const_iterator,const_iterator>
equal_range (const key_type& x) const;
```

Returns the pair `(lower_bound(x), upper_bound(x))`.

```
void
erase(iterator position);
```

Deletes the map element pointed to by the iterator `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
void
erase(iterator first, iterator last);
```

©Copyright 1999 Rogue Wave Software, Inc.

If the iterators `first` and `last` point to the same map and last is reachable from first, all elements in the range (`first, last`) are deleted from the map. Returns an iterator pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
size_type
erase(const key_type& x);
```

Deletes the element with the key value `x` from the map, if one exists. Returns `1` if `x` existed in the map, `0` otherwise.

```
iterator
find(const key_type& x);
```

Searches the map for a pair with the key value `x` and returns an `iterator` to that pair if it is found. If such a pair is not found the value `end()` is returned.

```
const_iterator find(const key_type& x) const;
```

Same as `find` above but returns a `const_iterator`.

```
pair<iterator, bool>
insert(const value_type& x);
iterator
insert(iterator position, const value_type& x);
```

If a `value_type` with the same key as `x` is not present in the map, then `x` is inserted into the map. Otherwise, the pair is not inserted. A position may be supplied as a hint regarding where to do the insertion. If the insertion is done right after `position`, then it takes amortized constant time. Otherwise it takes `O(log N)` time.

```
template <class InputIterator>
void
insert(InputIterator first, InputIterator last);
```

Copies of each element in the range `[first, last)` that possess a unique key (one not already in the map) are inserted into the map. The iterators `first` and `last` must return values of `type pair<T1,T2>`. This operation takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current map.

```
iterator
lower_bound(const key_type& x);
```

Returns a reference to the first entry with a key greater than or equal to `x`.

```
const_iterator
lower_bound(const key_type& x) const;
```

Same as `lower_bound` above but returns a `const_iterator`.

```
size_type
max_size() const;
```

Returns the maximum possible size of the map. This size is only constrained by the number of unique keys that can be represented by the type `Key`.

```
size_type
size() const;
```

Returns the number of elements in the map.

```
void
swap(map<Key, T, Compare, Allocator>& x);
```

Swaps the contents of the map `x` with the current map, `*this`.

```
iterator
upper_bound(const key_type& x);
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a reference to the first entry with a key less than or equal to x.

```
const_iterator
upper_bound(const key_type& x) const;
```

Same as upper_bound above but returns a const_iterator.

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing pair<const Key, T> values using the comparison operation, Compare, of the current map. This function is identical to key_comp for sets.

Non-member Operators

```
template <class Key, class T, class Compare, class Allocator>
bool operator==(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns true if all elements in x are element-wise equal to all elements in y, using (T::operator==). Otherwise it returns false.

```
template <class Key, class T, class Compare, class Allocator>
bool operator!=(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns !(x==y).

```
template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
               const map<Key, T, Compare, Allocator>& y);
```

Returns true if x is lexicographically less than y. Otherwise, it returns false.

```
template <class Key, class T, class Compare, class Allocator>
bool operator>(const map<Key, T, Compare, Allocator>& x,
               const map<Key, T, Compare, Allocator>& y);
```

Returns y < x.

```
template <class Key, class T, class Compare, class Allocator>
bool operator<=(const map<Key, T, Compare, Allocator>& x,
               const map<Key, T, Compare, Allocator>& y);
```

Returns !(y < x).

```
template <class Key, class T, class Compare, class Allocator>
bool operator>=(const map<Key, T, Compare, Allocator>& x,
               const map<Key, T, Compare, Allocator>& y);
```

Returns !(x < y).

Specialized Algorithms

```
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key, T, Compare, Allocator>& a,
          map<Key, T, Compare, Allocator>& b);
```

Swaps the contents of a and b.

Example

```
//
// map.cpp
//
 #include <string>
 #include <map>
 #include <iostream>
 using namespace std;

 typedef map<string, int, less<string> > months_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// Print out a pair
template <class First, class Second>
ostream& operator<<(ostream& out,
                    const pair<First,Second> & p)
{
  cout << p.first << " has " << p.second << " days";
  return out;
}

// Print out a map
ostream& operator<<(ostream& out, const months_type & l)
{
  copy(l.begin(),l.end(), ostream_iterator
              <months_type::value_type,char>(cout,"\n"));
  return out;
}


int main(void)
{
  // create a map of months and the number of days
  // in the month
  months_type months;

  typedef months_type::value_type value_type;

  // Put the months in the multimap
  months.insert(value_type(string("January"),   31));
  months.insert(value_type(string("February"),   28));
  months.insert(value_type(string("February"),   29));
  months.insert(value_type(string("March"),     31));
  months.insert(value_type(string("April"),     30));
  months.insert(value_type(string("May"),       31));
  months.insert(value_type(string("June"),      30));
  months.insert(value_type(string("July"),      31));
  months.insert(value_type(string("August"),    31));
  months.insert(value_type(string("September"), 30));
  months.insert(value_type(string("October"),   31));
  months.insert(value_type(string("November"),  30));
  months.insert(value_type(string("December"),  31));

  // print out the months
  // Second February is not present
  cout << months << endl;

  // Find the Number of days in June
  months_type::iterator p = months.find(string("June"));

  // print out the number of days in June
  if (p != months.end())
    cout << endl << *p << endl;

  return 0;
}
```

Program Output

```
April has 30 days
August has 31 days
December has 31 days
February has 28 days
January has 31 days
July has 31 days
June has 30 days
March has 31 days
May has 31 days
November has 30 days
October has 31 days
September has 30 days
```

Warnings

Member function templates are used in all containers included in the Standard Template Library. An example of this feature is the constructor for `map<Key,T,Compare,Allocator>` that takes two templatized iterators:

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class InputIterator>
 map (InputIterator, InputIterator,
      const Compare& = Compare(),
      const Allocator& = Allocator());
```

map also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a ***map*** in the following two ways:

```
map<int, int, less<int> >::value_type intarray[10];
map<int, int, less<int> > first_map(intarray,
                                    intarray + 10);
map<int, int, less<int> > second_map(first_map.begin(),
                                     first_map.end());
```

But not this way:

```
map<long, long, less<long> > long_map(first_map.begin(),
                                      first_map.end());
```

Since the `long_map` and `first_map` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these, you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you have to write:

```
map<int, int, less<int>, allocator<int> >
```

instead of:

```
map<int, int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Containers***, ***Iterators***, ***multimap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

mask_array

`Valarray helpers`

Summary

A numeric array class that gives a masked view of a valarray.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator%=()
operator&=() operator-=()
operator>>=() operator/=()
operator<<=() operator=()
operator*=() operator^=()
operator+=()

Synopsis

```
#include <valarray>
template <class T>
class mask_array;
```

Description

mask_array<T> gives a masked view into a ***valarray***. ***mask_arrays*** are only produced by applying the mask subscript operator to a ***valarray***. This subscript operator takes a `valarray<bool>` argument and produces a ***mask_array***. Only the elements in the ***valarray*** whose corresponding elements in the `valarray<bool>` argument were `true` are selected by the ***mask_array***. The elements in a ***mask_array*** are references to selected elements in the ***valarray*** (so changing an element in the ***mask_array*** really changes the corresponding element in the ***valarray***). A ***mask_array*** does not itself hold any distinct elements. The template cannot be instantiated directly since all its constructors are private. However, you can copy a ***slice_array*** to a ***valarray*** using either the ***valarray*** copy constructor or the assignment operator. Reference semantics are lost at that point.

Interface

```
template <class T> class mask_array {
public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // types
  typedef T value_type;

  // destructor
  ~mask_array();

  // public assignment
  void operator= (const valarray<T>& array) const;
  // computed assignment
  void operator*= (const valarray<T>& array) const;
  void operator/= (const valarray<T>& array) const;
  void operator%= (const valarray<T>& array) const;
  void operator+= (const valarray<T>& array) const;
  void operator-= (const valarray<T>& array) const;
  void operator^= (const valarray<T>& array) const;
  void operator&= (const valarray<T>& array) const;
  void operator|= (const valarray<T>& array) const;
  void operator<<= (const valarray<T>& array) const;
  void operator>>= (const valarray<T>& array) const;

  // other
  void operator= (const T&) const;

private:
  // constructors
  mask_array();
  mask_array(const mask_array<T>&);
  // operator =
  mask_array<T>& operator= (const mask_array<T>& array);
};
```

Constructors

```
mask_array();
mask_array(const mask_array&);
```

All mask_array constructors are private and cannot be called directly. This prevents copy construction of mask_arrays.

Assignment Operators

```
void operator=(const valarray<T>& x) const;
```

Assigns values from `x` to the selected elements of the valarray that self refers to. Remember that a mask_array never holds any elements itself, it simply refers to selected elements in the valarray used to generate it.

```
mask_array<T>&
operator=(const mask-_array<T>& x);
```

Private assignment operator. Cannot be called directly, thus preventing assignment between mask_arrays.

Computed Assignment Operators

```
void operator*=(const valarray<T>& val) const;
void operator/=(const valarray<T>& val) const;
void operator%=(const valarray<T>& val) const;
void operator+=(const valarray<T>& val) const;
void operator-=(const valarray<T>& val) const;
void operator^=(const valarray<T>& val) const;
void operator&=(const valarray<T>& val) const;
void operator|=(const valarray<T>& val) const;
void operator<<=(const valarray<T>& val) const;
void operator>>=(const valarray<T>& val) const;
```

Applies the indicated operation using elements from `val` to the selected elements of the valarray that self refers to. Remember that a mask_array never holds any elements itself; it simply refers to selected elements in the valarray used to generate it.

Member Functions

©Copyright 1999 Rogue Wave Software, Inc.

```
void operator= (const T& x) const;
```

Assigns x to the selected elements of the valarray that self refers to.

Example

```
//
// mask_array.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
using namespace std;

int main(void)
{
  int ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
  bool mbuf[10] = {1,0,1,1,1,0,0,1,1,0};

  // create a valarray of ints
  valarray<int>         vi(ibuf,10);

  // create a valarray of bools for a mask
  valarray<bool> mask(mbuf,10);

  // print out the valarray<int>
  cout << vi << endl;

  // Get a mask array and assign that mask to another array
  mask_array<int> msk = vi[mask];
  valarray<int> vi3 = msk;

  // print out the masked_array
  cout << vi3 << endl;

  // Double the masked values
  msk += vi3;

  // print out vi1 again
  cout << vi << endl;

  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,9]
[0,2,3,4,7,8]
[0,1,4,6,8,5,6,14,16,9]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, *slice_array*, *slice*, *gslice*, *gslice_array*, *indirect_array*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

max

`Algorithm`

Summary

Finds and returns the maximum of a pair of values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class T>
 const T& max(const T&, const T&);

template <class T, class Compare>
 const T& max(const T&, const T&, Compare);
```

Description

The ***max*** algorithm determines and returns the maximum of a pair of values. The optional argument `Compare` defines a comparison function that can be used in place of the default `operator<`.

max returns the first argument when the arguments are equal.

Example

```
//
// max.cpp
//
 #include <algorithm>
 #include <iostream>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   double d1 = 10.0, d2 = 20.0;

   // Find minimum
   double val1 = min(d1, d2);
   // val1 = 10.0

   // the greater comparator returns the greater of the
   // two values.
   double val2 = min(d1, d2, greater<double>());
   // val2 = 20.0;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // Find maximum
   double val3 = max(d1, d2);
   // val3 = 20.0;

   // the less comparator returns the smaller of
   // the two values.
   // Note that, like every comparison in the STL, max is
   // defined in terms of the < operator, so using less here
   // is the same as using the max algorithm with a default
   // comparator.
   double val4 = max(d1, d2, less<double>());
   // val4 = 20

   cout << val1 << " " << val2 << " "
        << val3 << " " << val4 << endl;

   return 0;
 }
```

Program Output

```
10 20 20 20
```

See Also

max_element, ***min***, ***min_element***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

max_element

`Algorithm`

Summary

Finds the maximum value in a range.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator>
 ForwardIterator
 max_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
 ForwardIterator
 max_element(ForwardIterator first, ForwardIterator last,
             Compare comp);
```

Description

The ***max_element*** algorithm returns an iterator that denotes the maximum element in a sequence. If the sequence contains more than one copy of the element, the iterator points to its first occurrence. The optional argument `comp` defines a comparison function that can be used in place of the default `operator<`.

Algorithm ***max_element*** returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the same range the following corresponding conditions hold:

```
!(*i < *j)
```

or

```
comp(*i, *j) == false.
```

Complexity

Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done for ***max_element***.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// max_elem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int d1[5] = {1,3,5,32,64};


   // set up vector
   vector<int>      v1(d1,d1 + 5);

   // find the largest element in the vector
   iterator it1 = max_element(v1.begin(), v1.end());
   // it1 = v1.begin() + 4

   // find the largest element in the range from
   // the beginning of the vector to the 2nd to last
   iterator it2 = max_element(v1.begin(), v1.end()-1,
                      less<int>());
   // it2 = v1.begin() + 3

   // find the smallest element
   iterator it3 = min_element(v1.begin(), v1.end());
   // it3 = v1.begin()

   // find the smallest value in the range from
   // the beginning of the vector plus 1 to the end
   iterator it4 = min_element(v1.begin()+1, v1.end(),
                      less<int>());
   // it4 = v1.begin() + 1

   cout << *it1 << " " << *it2 << " "
        << *it3 << " " << *it4 << endl;

   return 0;
 }
```

Program Output

```
64 32 1 3
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

max, ***min***, ***min_element***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

mem_fun, mem_fun1, mem_fun_ref, mem_fun_ref1

`Function Adaptors`

Summary

Function objects that adapt a pointer to a member function, to take the place of a global function.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class S, class T> class mem_fun_t;
template <class S, class T, class A> class mem_fun1_t;
template <class S, class T> class mem_fun_ref_t;
template <class S, class T, class A> class mem_fun1_ref_t;

template <class S, class T> class const_mem_fun_t;
template <class S, class T, class A> class const_mem_fun1_t;
template <class S, class T> class const_mem_fun_ref_t;
template <class S, class T, class A>
          class const_mem_fun1_ref_t;

template<class S, class T> mem_fun_t<S,T>
   mem_fun(S, (T::*f)());
template<class S, class T, class A> mem_fun1_t<S,T,A>
   mem_fun1(S, (T::*f)(A));
template<class S, class T> mem_fun_ref_t<S,T>
   mem_fun_ref(S, (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
   mem_fun1_ref(S, (T::*f)(A));

template<class S, class T> const_mem_fun_t<S,T>
   mem_fun(S, (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_t<S,T,A>
   mem_fun1(S, (T::*f)(A) const);
template<class S, class T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S, (T::*f)() const);
template<class S, class T, class A>
   const_mem_fun1_ref_t<S,T,A>
   mem_fun1_ref(S, (T::*f)(A) const);
```

Description

The ***mem_fun*** group of templates encapsulates a pointer to a member function. Each category of template (***mem_fun***, ***mem_fun1***, ***mem_fun_ref***, or ***mem_fun1_ref***) includes both a class template and a function template, where the class is

©Copyright 1999 Rogue Wave Software, Inc.

distinguished by the addition of `_t` on the end of the name to identify it as a type. A set of class templates for const member functions exists, each with `const_` prepended to the name.

The class's constructor takes a pointer to a member function, and uses an `operator()` to forward the call to that member function. In this way the resulting object serves as a global function object for that member function.

The accompanying function template simplifies the use of this facility by constructing an instance of the class on the fly.

The library includes zero and one argument adaptors for containers of pointers and containers of references (`_ref`). This technique can be easily extended to include adaptors for two argument functions, and so on.

Interface

```
template <class S, class T> class mem_fun_t
          : public unary_function<T*, S> {
    public:
      explicit mem_fun_t(S (T::*p)());
      S operator()(T* p) const;
};

template <class S, class T, class A> class mem_fun1_t
          : public binary_function<T*, A, S> {
    public:
      explicit mem_fun1_t(S (T::*p)(A));
      S operator()(T* p, A x) const;
};

template<class S, class T> mem_fun_t<S,T>
   mem_fun(S, (T::*f)());

template<class S, class T, class A> mem_fun1_t<S,T,A>
    mem_fun1(S, (T::*f)(A));

template <class S, class T> class mem_fun_ref_t
          : public unary_function<T, S> {
    public:
      explicit mem_fun_ref_t(S (T::*p)());
      S operator()(T* p) const;
};

template <class S, class T, class A> class mem_fun1_ref_t
          : public binary_function<T, A, S> {
    public:
      explicit mem_fun1_ref_t(S (T::*p)(A));
      S operator()(T* p, A x) const;
};
template<class S, class T> mem_fun_ref_t<S,T>
     mem_fun_ref(S, (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
     mem_fun1_ref(S, (T::*f)(A));

// For const member functions

template <class S, class T> class const_mem_fun_t
          : public unary_function<T*, S> {
    public:
      explicit const_mem_fun_t(S (T::*p)() const);
      S operator()(T* p) const;
};

template <class S, class T, class A> class const_mem_fun1_t
          : public binary_function<T*, A, S> {
    public:
      explicit const_mem_fun1_t(S (T::*p)(A) const);
      S operator()(T* p, A x) const;
};

template<class S, class T> const_mem_fun_t<S,T>
   mem_fun(S, (T::*f)() const);

template<class S, class T, class A> const_mem_fun1_t<S,T,A>
    mem_fun1(S, (T::*f)(A) const);

template <class S, class T> class const_mem_fun_ref_t
```

©Copyright 1999 Rogue Wave Software, Inc.

```
          : public unary_function<T, S> {
    public:
      explicit const_mem_fun_ref_t(S (T::*p)() const);
      S operator()(T* p) const;
};

template <class S, class T, class A>
          class const_mem_fun1_ref_t
          : public binary_function<T, A, S> {
    public:
      explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
      S operator()(T* p, A x) const;
};
template<class S, class T> const_mem_fun_ref_t<S,T>
     mem_fun_ref(S, (T::*f)() const);
template<class S, class T, class A>
     const_mem_fun1_ref_t<S,T,A>
     mem_fun1_ref(S, (T::*f)(A) const);
```

Example

```
//
// mem_fun example
//

#include <functional>
#include <list>
using namespace std;

int main(void)
{
  int a1[] = {2,1,5,6,4};
  int a2[] = {11,4,67,3,14};
  list<int> s1(a1,a1+5);
  list<int> s2(a2,a2+5);

  // Build a list of lists
  list<list<int>* > l;
  l.insert(l.begin(),s1);
  l.insert(l.begin(),s2);

  // Sort each list in the list
  for_each(l.begin(),l.end(),mem_fun(&list<int>::sort));
}
```

See Also

binary_function, *Function Objects*, *pointer_to_unary_function*, *ptr_fun*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

merge

`Algorithm`

Summary

Merges two sorted sequences into a third sequence.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
  merge(InputIterator first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator last2,
        OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
  merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator last2,
        OutputIterator result, Compare comp);
```

Description

The ***merge*** algorithm merges two sorted sequences, specified by `[first1, last1)` and `[first2, last2)`, into the sequence specified by `[result, result + (last1 - first1) + (last2 - first2))`. The first version of the ***merge*** algorithm uses the less than operator (`<`) to compare elements in the two sequences. The second version uses the comparison function included in the function call. If a comparison function is included, ***merge*** assumes that both sequences were sorted using that comparison function.

The merge is stable. This means that if the two original sequences contain equivalent elements, the elements from the first sequence always precede the matching elements from the second in the resulting sequence. The size of the result of a ***merge*** is equal to the sum of the sizes of the two argument sequences. ***merge*** returns an iterator that points to the end of the resulting sequence (in other words, `result + (last1 - first1) + (last2 -first2)`). The result of ***merge*** is undefined if the resulting range overlaps with either of the original ranges.

merge assumes that there are at least `(last1 - first1) + (last2 - first2)` elements following `result`, unless `result` has been adapted by an insert iterator.

©Copyright 1999 Rogue Wave Software, Inc.

Complexity

At most `(last - first1) + (last2 - first2) - 1` comparisons are performed.

Example

```
//
// merge.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main()
 {
   int d1[4] = {1,2,3,4};
   int d2[8] = {11,13,15,17,12,14,16,18};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
   // Set up four destination vectors
   vector<int> v3(d2,d2 + 8),v4(d2,d2 + 8),
               v5(d2,d2 + 8),v6(d2,d2 + 8);
   // Set up one empty vector
   vector<int> v7;

   // Merge v1 with v2
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         v3.begin());
   // Now use comparator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v4.begin(),
         less<int>());

   // In place merge v5
   vector<int>::iterator mid = v5.begin();
   advance(mid,4);
   inplace_merge(v5.begin(),mid,v5.end());
   // Now use a comparator on v6
   mid = v6.begin();
   advance(mid,4);
   inplace_merge(v6.begin(),mid,v6.end(),less<int>());

   // Merge v1 and v2 to empty vector using insert iterator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         back_inserter(v7));

   // Copy all cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;
   copy(v3.begin(),v3.end(),out);
   cout << endl;
   copy(v4.begin(),v4.end(),out);
   cout << endl;
   copy(v5.begin(),v5.end(),out);
   cout << endl;
   copy(v6.begin(),v6.end(),out);
   cout << endl;
   copy(v7.begin(),v7.end(),out);
   cout << endl;

   // Merge v1 and v2 to cout
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

©Copyright 1999 Rogue Wave Software, Inc.

```
1 2 3 4
1 2 3 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
11 12 13 14 15 16 17 18
11 12 13 14 15 16 17 18
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

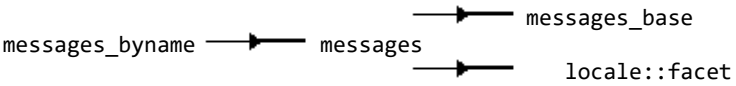
Containers, ***inplace_merge***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

messages, messages_byname



Summary

Messaging facets.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
string_type

Member Functions

close()
do_close()
do_get()
do_open()
get()
open()

Synopsis

```
#include<locale>
class messages_base;
template <class charT> class messages;
```

Description

messages gives access to a localized messaging facility. The ***messages*** facet is used with the "C" locale, while the ***messages_byname*** facet is used with named locales.

The ***messages_base*** class includes a catalog type for use by the derived ***messages*** and ***messages_byname*** classes.

Note that the default messages facet uses `catopen, catclose`, etc., to implement the message database. If your platform does not support these, then you need to imbue your own messages facet by implementing whatever database is available.

©Copyright 1999 Rogue Wave Software, Inc.

Interface

```
class messages_base {
public:
  typedef int catalog;
};

template <class charT>
class messages : public locale::facet, public messages_base {
public:
  typedef charT char_type;
  typedef basic_string<charT> string_type;
  explicit messages(size_t = 0);
  catalog open(const basic_string<char>&, const locale&)
              const;
  string_type get(catalog, int, int,
                  const string_type&) const;
  void    close(catalog) const;
  static locale::id id;
protected:
  ~messages();  // virtual
  virtual catalog do_open(const basic_string<char>&,
                          const locale&) const;
  virtual string_type do_get(catalog, int, int,
                             const string_type&) const;
  virtual void    do_close(catalog) const;
};

class messages_byname : public messages<charT> {
public:
  explicit messages_byname(const char*, size_t = 0);
protected:
  ~messages_byname();  // virtual
  virtual catalog do_open(const basic_string<char>&,
                          const locale&) const;
virtual string_type do_get(catalog, int, int,
                           const string_type&) const;
  virtual void    do_close(catalog) const;
};
```

Types

`char_type`

Type of character the facet is instantiated on.

`string_type`

Type of character string returned by member functions.

Constructors

`explicit` **`messages`**`(size_t refs = 0)`

Constructs a ***messages*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted because the locale does not do so.

`explicit` **`messages_byname`**`(const char* name,`
`  size_t refs = 0);`

Constructs a ***messages_byname*** facet. Uses the named locale specified by the name argument. The `refs` argument serves the same purpose as it does for the ***messages*** constructor.

Destructors

`~messages``();  // virtual and protected`

Destroys the facet.

Facet ID

©Copyright 1999 Rogue Wave Software, Inc.

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***messages*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `open` function simply calls its protected cousin `do_open`.

```
void

close(catalog c) const;
string_type
get(catalog c, int set, int msgid,
    const string_type& dfault) const;
catalog
open(const basic_string<char>& fn, const locale&) const;
```

Each of these public member functions `xxx` simply call the corresponding protected `do_xxx` function.

Protected Member Functions

```
virtual void
do_close(catalog cat) const;
```

Closes the catalog. The `cat` argument must be obtained by a call to `open()`.

```
virtual string_type
do_get(catalog cat, int set, int msgid,
       const string_type& dfault) const;
```

Retrieves a specific message. Returns the message identified by `cat`, `set`, `msgid`, and `dfault`. `cat` must be obtained by a previous call to `open()`. This function must not be called with a `cat` that has had close called on it after the last call to `open()`. That is, the catalog must be open and not yet closed.

```
virtual catalog
do_open(const basic_string<char>& name, const locale&) const;
```

Opens a message catalog. Returns a catalog identifier that can be passed to the `get()` function in order to access specific messages. Returns `-1` if the catalog name specified in the `name` argument is invalid. The `loc` argument is used for codeset conversion if necessary.

Example

```
//
// messages.cpp
//
#include <string>
#include <iostream>

int main ()
{
  using namespace std;

  locale loc;

  // Get a reference to the messages<char> facet
  const messages<char>& mess =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
    use_facet<messages<char> >(loc);
#else
    use_facet(loc,(messages<char>*)0);
#endif

  // Open a catalog and try to grab
  // both some valid messages, and an invalid message
  string def("Message Not Found");
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  messages<char>::catalog cat =
          mess.open("./rwstdmessages.cat",loc);
  if (cat != -1)
  {
    string msg0 = mess.get(cat,1,1,def);
    string msg1 = mess.get(cat,1,2,def);
    string msg2 = mess.get(cat,1,6,def); // invalid msg #
    string msg3 = mess.get(cat,2,1,def);

    mess.close(cat);
    cout << msg0 << endl << msg1 << endl
         << msg2 << endl << msg3 << endl;
  }
  else
    cout << "Unable to open message catalog" << endl;

  return 0;
}
```

See Also

locale, *facets*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

min

`Algorithm`

Summary

Finds and returns the minimum of a pair of values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class T>
 const T& min(const T&, const T&);

template <class T, class Compare>
 const T& min(const T& a, const T&, Compare);
```

Description

The ***min*** algorithm determines and returns the minimum of a pair of values. In the second version of the algorithm, the optional argument `Compare` defines a comparison function that can be used in place of the default `operator<`.

min returns the first argument when the two arguments are equal.

Example

```
//
// max.cpp
//
 #include <algorithm>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   double d1 = 10.0, d2 = 20.0;

   // Find minimum
   double val1 = min(d1, d2);
   // val1 = 10.0

   // the greater comparator returns the greater of the
   // two values.
   double val2 = min(d1, d2, greater<double>());
   // val2 = 20.0;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // Find maximum
   double val3 = max(d1, d2);
   // val3 = 20.0;

   // the less comparator returns the smaller of the
   // two values.
   // Note that, like every comparison in the STL, max is
   // defined in terms of the < operator, so using less here
   // is the same as using the max algorithm with a default
   // comparator.
   double val4 = max(d1, d2, less<double>());
   // val4 = 20

   cout << val1 << " " << val2 << " "
        << val3 << " " << val4 << endl;

   return 0;
 }
```

Program Output

```
10 20 20 20
```

See Also

max, ***max_element***, ***min_element***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

min_element

`Algorithm`

Summary

Finds the minimum value in a range.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator>
 ForwardIterator
 min_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
 InputIterator
 min_element(ForwardIterator first, ForwardIterator last,
             Compare comp);
```

Description

The ***min_element*** algorithm returns an iterator that denotes the minimum element in a sequence. If the sequence contains more than one copy of the minimum element, the iterator points to the first occurrence of the element. In the second version of the function, the optional argument `comp` defines a comparison function that can be used in place of the default `operator<`.

Algorithm ***min_element*** returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the same range, the following corresponding conditions hold:

```
!(*j < *i)
```

or

```
comp(*j, *i) == false.
```

Complexity

min_element performs exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// max_elem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int d1[5] = {1,3,5,32,64};

   // set up vector
   vector<int>      v1(d1,d1 + 5);

   // find the largest element in the vector
   iterator it1 = max_element(v1.begin(), v1.end());
   // it1 = v1.begin() + 4

   // find the largest element in the range from
   // the beginning of the vector to the 2nd to last
   iterator it2 = max_element(v1.begin(), v1.end()-1,
                      less<int>());
   // it2 = v1.begin() + 3

   // find the smallest element
   iterator it3 = min_element(v1.begin(), v1.end());
   // it3 = v1.begin()

   // find the smallest value in the range from
   // the beginning of the vector plus 1 to the end
   iterator it4 = min_element(v1.begin()+1, v1.end(),
                      less<int>());
   // it4 = v1.begin() + 1

   cout << *it1 << " " << *it2 << " "
        << *it3 << " " << *it4 << endl;

   return 0;
 }
```

Program Output

```
64 32 1 3
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int,allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

max, ***max_element***, ***min***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

minus

`Function Object`

Summary

Returns the result of subtracting its second argument from its first.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
template <class T>
struct minus : public binary_function<T, T, T>;
```

Description

minus is a binary function object. Its `operator()` returns the result of `x` minus `y`. You can pass a ***minus*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***minus*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), minus<int>());
```

After this call to ***transform***, `vecResult(n)` contains `vec1(n)` minus `vec2(n)`.

Interface

```
template <class T>
struct minus : binary_function<T, T, T> {
  T operator() (const T&, const T&) const;
};
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

mismatch

`Algorithm`

Summary

Compares elements from two sequences and returns the first two elements that don't match each other.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2>
  pair<InputIterator1,InputIterator2>
  mismatch(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
  pair<InputIterator1, Inputiterator2>
  mismatch(InputIterator first1, InputIterator1 last1,
           InputIterator2 first2,
           BinaryPredicate binary_pred);
```

Description

The ***mismatch*** algorithm compares members of two sequences and returns two iterators (`i` and `j`) that point to the first location in each sequence where the sequences differ from each other. Notice that the algorithm denotes both a starting position and an ending position for the first sequence, but denotes only a starting position for the second sequence. ***mismatch*** assumes that the second sequence has at least as many members as the first sequence. If the two sequences are identical, ***mismatch*** returns a pair of iterators that point to the end of the first sequence and the corresponding location at which the comparison stopped in the second sequence.

The first version of ***mismatch*** checks members of a sequence for equality, while the second version lets you specify a comparison function. The comparison function must be a binary predicate.

The iterators `i` and `j` returned by ***mismatch*** are defined as follows:

```
j  == first2  +  (i  -  first1)
```

and `i` is the first iterator in the range `[first1, last1)` for which the appropriate one of the following conditions hold:

```
!(*i  ==  *(first2  +  (i  -  first1)))
```

or

©Copyright 1999 Rogue Wave Software, Inc.

```
binary_pred(*i, *(first2 + (i - first1))) == false
```

If all of the members in the two sequences match, ***mismatch*** returns a pair of `last1` and `first2 + (last1 - first1)`.

Complexity

At most `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// mismatch.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> vi1(d1,d1 + 4), vi2(d2,d2 + 4);

   // p1 will contain two iterators that point to the
   // first pair of elements that are different between
   // the two vectors
   pair<iterator, iterator> p1 = mismatch(vi1.begin(),
                           vi1.end(),vi2.begin());

   // find the first two elements such that an element in
   // the first vector is greater than the element in
   // the second vector.
   pair<iterator, iterator> p2 = mismatch(vi1.begin(),
                                vi1.end(), vi2.begin(),
                                     less_equal<int>());

   // Output results
   cout << *p1.first << ", " << *p1.second << endl;
   cout << *p2.first << ", " << *p2.second << endl;

   return 0;
 }
```

Program Output

```
2, 3
3, 2
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

modulus

`Function Object`

Summary

Returns the remainder obtained by dividing the first argument by the second argument.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
template <class T>
struct modulus : public binary_function<T, T, T> ;
```

Description

modulus is a binary function object. Its `operator()` returns the remainder resulting from `x` divided by `y`. You can pass a ***modulus*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***modulus*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), modulus<int>());
```

After this call to ***transform***, `vecResult(n)` contains the remainder of `vec1(n)` divided by `vec2(n)`.

Interface

```
template <class T>
struct modulus : binary_function<T, T, T> {
  T operator() (const T&, const T&) const;
};
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

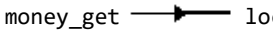
binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

money_get



Summary

Monetary formatting facet for input.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
do_get(
id
iter_type
string_type

Member Functions

do_get()
get()

Synopsis

```
#include <locale>
template <class charT,
          class InputIterator = istreambuf_iterator<charT> >
class money_get;
```

Description

The ***money_get*** facet interprets formatted monetary string values.

Interface

```
template <class charT,
          class InputIterator = istreambuf_iterator<charT> >
class money_get : public locale::facet {
public:
  typedef charT               char_type;
  typedef InputIterator       iter_type;
  typedef basic_string<charT> string_type;
  explicit money_get(size_t = 0);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  iter_type get(iter_type, iter_type, bool, ios_base&,
                ios_base::iostate&, long double&) const;
  iter_type get(iter_type, iter_type, bool, ios_base&,
                ios_base::iostate&, string_type&) const;
  static locale::id id;
protected:
  ~money_get();  // virtual
  virtual iter_type do_get(iter_type, iter_type,
                           bool, ios_base&,
                           ios_base::iostate&,
                           long double&) const;
  virtual iter_type do_get(iter_type, iter_type,
                           bool, ios_base&,
                           ios_base::iostate&,
                           string_type&) const;
};
```

Types

`char_type`

Type of character upon which the facet is instantiated.

`iter_type`

Type of iterator used to scan the character buffer.

`string_type`

Type of character string passed to member functions.

Constructors

```
explicit money_get(size_t refs = 0)
```

Construct a ***money_get*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

Destructors

```
~money_get();  // virtual and protected
```

Destroys the facet.

Static Members

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***money_get*** facet include an interface to protected members. Each public member `get` has a corresponding virtual protected member `do_get`.

```
iter_type
get(iter_type s, iter_type end, bool intl, ios_base& f,
    ios_base::iostate& err, long double& units) const;
iter_type
get(iter_type s, iter_type end, bool intl, ios_base& f,
    ios_base::iostate& err, string_type& digits) const;
```

Each of these two overloads of the public member function `get` calls the corresponding protected `do_get` function.

Protected Member Functions

©Copyright 1999 Rogue Wave Software, Inc.

```
virtual iter_type
do_get(iter_type s, iter_type end,
       bool intl, ios_base& f,
       ios_base::iostate& err,
       long double& units) const;
virtual iter_type
do_get(iter_type s, iter_type end,
       bool intl, ios_base& f,
       ios_base::iostate& err,
       string_type& digits) const;
```

Reads in a localized character representation of a monetary value and generates a generic representation, either as a sequence of digits or as a `long double` value. In either case `do_get` uses the smallest possible unit of currency.

Both overloads of `do_get` read characters from the range `[s,end)` until one of three things occurs:

- A monetary value is assembled
- An error occurs
- No more characters are available.

The functions use `f.flags()` and the `moneypunct<charT, true>` or `moneypunct<charT, false>` facet (depending on the `intl` argument) from `f.getloc()` for formatting information to use in interpreting the sequence of characters. `do_get`, then places a pure sequence of digits representing the monetary value in the smallest possible unit of currency into the string argument `digits`, or it calculates a `long double` value based on those digits and returns that value in `units`.

The following specifics apply to formatting:

- Digit group separators are optional. If no grouping is specified, then any thousands separator characters are treated as delimiters.
- If `space` or `none` are part of the format pattern in `moneypunct`, then optional whitespace is consumed, except at the end. See the `moneypunct` reference section for a description of money-specific formatting flags.
- If `iosbase::showbase` is set in `f.flags()`, then the currency symbol is optional, and if it appears after all other elements, then it is not consumed. Otherwise the currency symbol is required, and is consumed wherever it occurs.
- `digits` are preceded by a `-' or `units` are negated, if the monetary value is negative.
- See the `moneypunct` reference section for a description of money specific `formatting` flags.

The `err` argument is set to `iosbase::failbit` if an error occurs during parsing.

Returns an iterator pointing one past the last character that is part of a valid monetary sequence.

Example

```
//
// moneyget.cpp
//

#include <string>
#include <sstream>
using namespace std;

int main ()
{
  using namespace std;
  typedef istreambuf_iterator<char,char_traits<char> >
          iter_type;

  locale loc;
  string buffer("$100.02");
  string dest;
  long double ldest;
  ios_base::iostate state;
  iter_type end;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // Get a money_get facet
  const money_get<char,iter_type>& mgf =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
    use_facet<money_get<char,iter_type> >(loc);
#else
    use_facet(loc,(money_get<char,iter_type>*)0);
#endif

  {
    // Build an istringstream from the buffer and construct
    // a beginning iterator on it.
    istringstream ins(buffer);
    iter_type begin(ins);

    // Get a string representation of the monetary value
    mgf.get(begin,end,false,ins,state,dest);
  }
  {
    // Build another istringstream from the buffer, etc.
    // so we have an iterator pointing to the beginning
    istringstream ins(buffer);
    iter_type begin(ins);

    // Get a long double representation
    // of the monetary value
    mgf.get(begin,end,false,ins,state,ldest);
  }
  cout << buffer << " --> " << dest
       << " --> " << ldest << endl;

  return 0;
}
```

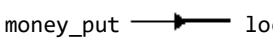
See Also

locale, *facets*, *money_put*, *moneypunct*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

money_put



Summary

Monetary formatting facet for output.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
iter_type
string_type

Member Functions

do_put()
put()

Synopsis

```
#include <locale>
template <class charT,
          class OutputIterator = ostreambuf_iterator<charT> >
class money_put;
```

Description

The ***money_put*** facet takes a `long double` value, or generic sequence of digits and writes out a formatted representation of the monetary value.

Interface

```
template <class charT,
          class OutputIterator = ostreambuf_iterator<charT> >
class money_put : public locale::facet {
public:
  typedef charT               char_type;
  typedef OutputIterator      iter_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef basic_string<charT> string_type;
  explicit money_put(size_t = 0);
  iter_type put(iter_type, bool, ios_base&, char_type,
                long double) const;
  iter_type put(iter_type, bool, ios_base&, char_type,
                const string_type&) const;
  static locale::id id;
protected:
  ~money_put();  // virtual
  virtual iter_type  do_put(iter_type, bool, ios_base&,
                            char_type, long double) const;
  virtual iter_type do_put(iter_type, bool, ios_base&,
                           char_type, const string_type&)
                           const;
};
```

Types

`char_type`

Type of the character upon which the facet is instantiated.

`iter_type`

Type of iterator used to scan the character buffer.

`string_type`

Type of character string passed to member functions.

Constructors

```
explicit money_put(size_t refs = 0)
```

Construct a ***money_put*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so.

Destructors

```
~money_put();  // virtual and protected
```

Destroys the facet.

Static Members

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***money_put*** facet include an interface to protected members. Each public member `put` has a corresponding virtual protected member `do_put`.

```
iter_type
put(iter_type s, bool intl, ios_base& f, char_type fill,
    long double units) const;
iter_type
put(iter_type s, bool intl, ios_base& f, char_type fill,
    const string_type& digits) const;
```

Each of these two overloads of the public member function `put` simply calls the corresponding protected `do_put` function.

Protected Member Functions

©Copyright 1999 Rogue Wave Software, Inc.

```
virtual iter_type
do_put(iter_type s, bool intl, ios_base& f, char_type fill,
       long double units) const;
```

Writes out a character string representation of the monetary value contained in `units`. Since `units` represents the monetary value in the smallest possible unit of currency, any fractional portions of the value are ignored. `f.flags()` and the `moneypunct<charT, intl>` facet from `f.getloc()` give the formatting information.

The `fill` argument is used for any padding.

Returns an iterator pointing one past the last character written.

```
virtual iter_type
do_put(iter_type s, bool intl, ios_base& f, char_type fill,
       const string_type& digits) const;
```

Writes out a character string representation of the monetary value contained in `digits`. `digits` represents the monetary value as a sequence of digits in the smallest possible unit of currency. `do_put` only looks at an optional `-` character and any immediately contiguous digits. `f.flags()` and the `moneypunct<charT, intl>` facet from `f.getloc()` give the formatting information.

The `fill` argument is used for any padding.

Returns an iterator pointing one past the last character written.

Example

```
//
// moneyput.cpp
//

#include <string>
#include <iostream>

int main ()
{
  using namespace std;

  typedef ostreambuf_iterator<char,char_traits<char> >
          iter_type;

  locale loc;
  string buffer("10002");
  long double ldval = 10002;

  // Construct a ostreambuf_iterator on cout
  iter_type begin(cout);

  // Get a money put facet
  const money_put<char,iter_type>& mp =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<money_put<char,iter_type> >(loc);
#else
  use_facet(loc,(money_put<char,iter_type>*)0);
#endif

  // Put out the string representation of the monetary value
  cout << buffer << " --> ";
  mp.put(begin,false,cout,' ',buffer);

  // Put out the long double representation
  // of the monetary value
  cout << endl << ldval << " --> ";
  mp.put(begin,false,cout,' ',ldval);

  cout <<  endl;

  return 0;
}
```

See Also

©Copyright 1999 Rogue Wave Software, Inc.

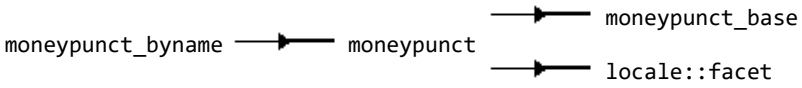
locale, ***facets***, ***money_get***, ***moneypunct***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

moneypunct, moneypunct_byname



Summary

Monetary punctuation facets.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
Intl
string_type

Member Functions

curr_symbol()	do_negative_sign()	grouping()
decimal_point()	do_neg_format()	negative_sign()
do_curr_symbol()	do_positive_sign()	neg_format()
do_decimal_point()	do_pos_format()	positive_sign()
do_frac_digits()	do_thousands_sep()	pos_format()
do_grouping()	frac_digits()	thousands_sep()

Synopsis

```
#include <locale>
class money_base;
template <class charT, bool International = false>
class moneypunct;
```

Description

The ***moneypunct*** facets include formatting specifications and punctuation character for monetary values. The ***moneypunct*** facet is used with the "C" locale, while the ***moneypunct_byname*** facet is used with named locales.

The facet is used by ***money_put*** for outputting formatted representations of monetary values and by ***money_get*** for reading these strings back in.

©Copyright 1999 Rogue Wave Software, Inc.

money_base includes a structure, `pattern`, that specifies the order of syntactic elements in a monetary value and enumeration values representing those elements. The `pattern` struct includes a simple array of characters, `field`. Each index in `field` is taken up by an enumeration value indicating the location of a syntactic element. The enumeration values are described below:

Format Flag	Meaning
none	No grouping separator
space	Use space for grouping separator
symbol	Currency symbol
sign	Sign of monetary value
value	The monetary value itself

The `do_pos_format` and `do_neg_format` member functions of ***moneypunct*** both return the `pattern` type. See the description of these functions for further elaboration.

Interface

```
class money_base {
public:
  enum part { none, space, symbol, sign, value };
  struct pattern { char field[4]; };
};

template <class charT, bool International = false>
class moneypunct : public locale::facet, public money_base {
public:
  typedef charT char_type;
  typedef basic_string<charT> string_type;
  explicit moneypunct(size_t = 0);
  charT        decimal_point() const;
  charT        thousands_sep() const;
  string       grouping()      const;
  string_type  curr_symbol()   const;
  string_type  positive_sign() const;
  string_type  negative_sign() const;
  int          frac_digits()   const;
  pattern      pos_format()    const;
  pattern      neg_format()    const;
  static locale::id id;
  static const bool intl = International;
protected:
  ~moneypunct();  // virtual
  virtual charT        do_decimal_point() const;
  virtual charT        do_thousands_sep() const;
  virtual string       do_grouping()      const;
  virtual string_type  do_curr_symbol()   const;
  virtual string_type  do_positive_sign() const;
  virtual string_type  do_negative_sign() const;
  virtual int          do_frac_digits()   const;
  virtual pattern      do_pos_format()    const;
  virtual pattern      do_neg_format()    const;
};

template <class charT, bool Intl = false>
class moneypunct_byname : public moneypunct<charT, Intl> {
public:
  explicit moneypunct_byname(const char*, size_t = 0);
protected:
  ~moneypunct_byname();  // virtual
  virtual charT        do_decimal_point() const;
  virtual charT        do_thousands_sep() const;
  virtual string       do_grouping()      const;
  virtual string_type  do_curr_symbol()   const;
  virtual string_type  do_positive_sign() const;
  virtual string_type  do_negative_sign() const;
  virtual int          do_frac_digits()   const;
  virtual pattern      do_pos_format()    const;
  virtual pattern      do_neg_format()    const;
};
```

Types

©Copyright 1999 Rogue Wave Software, Inc.

char_type

Type of character the facet is instantiated on.

string_type

Type of character string returned by member functions.

Constructors

```
explicit moneypunct(size_t refs = 0)
```

Constructs a ***moneypunct*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

```
explicit moneypunct_byname(const char* name,
                           size_t refs = 0);
```

Constructs a ***moneypunct_byname*** facet. Uses the named locale specified by the `name` argument. The `refs` argument serves the same purpose as it does for the ***moneypunct*** constructor.

Destructors

```
~moneypunct();  // virtual and protected
```

Destroys the facet.

Static Members

```
static locale::id id;
```

Unique identifier for this type of facet.

```
static const bool intl = Intl;
```

`true` for international representation, `false` otherwise.

Public Member Functions

The public members of the ***moneypunct*** and ***moneypunct_byname*** facets include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `decimal_point` function simply calls its protected cousin `do_decimal_point`.

```
string_type  curr_symbol()   const;
charT        decimal_point() const;
int          frac_digits()   const;
string       grouping()      const;
pattern      neg_format()    const;
string_type  negative_sign() const;
pattern      pos_format()    const;
string_type  positive_sign() const;
charT        thousands_sep() const;
```

Each public member function `xxx` simply calls the corresponding protected `do_xxx` function.

Protected Member Functions

```
virtual string_type
do_curr_symbol()   const;
```

Returns a string to use as the currency symbol.

```
virtual charT
do_decimal_point() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns the radix separator to use if fractional digits are allowed (see do_frac_digits).

```
virtual int
do_frac_digits()   const;
```

Returns the number of digits in the fractional part of the monetary representation.

```
virtual string
do_grouping()      const;
```

Returns a string in which each character is used as an integer value to represent the number of digits in a particular grouping, starting with the rightmost group. A group is simply the digits between adjacent thousands' separators. Each group at a position larger than the size of the string gets the same value as the last element in the string. If a value is less than or equal to zero, or equal to CHAR_MAX, then the size of that group is unlimited. ***moneypunct*** returns an empty string, indicating no grouping.

```
virtual string_type
do_negative_sign() const;
```

A string to use as the negative sign. The first character of this string is placed in the position indicated by the format pattern (see do_neg_format); the rest of the characters, if any, are placed after all other parts of the monetary value.

```
virtual pattern
do_neg_format()    const;
virtual pattern
do_pos_format()    const;
```

Returns a pattern object specifying the location of the various syntactic elements in a monetary representation. The enumeration values symbol, sign, and value appear exactly once in this pattern, with the remaining location taken by either none or space. none never occupies the first position in the pattern and space never occupies the first or the last position. Beyond these restrictions, elements may appear in any order. ***moneypunct*** returns {symbol, sign, none, value}.

```
virtual string_type
do_positive_sign() const;
```

A string to use as the positive sign. The first character of this string is placed in the position indicated by the format pattern (see do_pos_format); the rest of the characters, if any, are placed after all other parts of the monetary value.

```
virtual charT
do_thousands_sep() const;
```

Returns the grouping separator if grouping is allowed (see do_grouping).

Example

```
//
// moneypun.cpp
//

#include <string>

#include <iostream>

int main ()
{
  using namespace std;

  locale loc;

  // Get a moneypunct facet
  const moneypunct<char,false>& mp =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<moneypunct<char,false> >(loc);
#else
  use_facet(loc,(moneypunct<char,false>*)0);
#endif

  cout << "Decimal point        = "
       << mp.decimal_point() << endl;
  cout << "Thousands separator  = "
```

©Copyright 1999 Rogue Wave Software, Inc.

```
       << mp.thousands_sep() << endl;
  cout << "Currency symbol      = "
       << mp.curr_symbol() << endl;
  cout << "Negative Sign        = "
       << mp.negative_sign() << endl;
  cout << "Digits after decimal = "
       << mp.frac_digits() << endl;

  return 0;
}
```

See Also

locale, *facets*, *money_put*, *money_get*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

multimap

`Container`

Summary

An associative container that gives access to non-key values using keys. ***multimap*** keys are not required to be unique. A ***multimap*** supports bidirectional iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

			operator=()
begin()	erase()	max_size()	operator==()
clear()	find()	operator!=()	rbegin()
count()	get_allocator()	operator>()	rend()
empty()	insert()	operator>=()	size()
end()	key_comp()	operator<()	swap()
equal_range()	lower_bound()	operator<=()	upper_bound()
			value_comp()

Synopsis

```
#include <map>
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T>> >
class multimap;
```

Description

multimap <Key ,T, Compare, Allocator> gives fast access to stored values of type `T` that are indexed by keys of type `Key`. The default operation for key comparison is the `<` operator. Unlike ***map***, ***multimap*** allows insertion of duplicate keys.

multimap uses bidirectional iterators that point to an instance of `pair<const Key x, T y>` where `x` is the key and `y` is the stored value associated with that key. The definition of ***multimap*** includes a `typedef` to this pair called `value_type`.

©Copyright 1999 Rogue Wave Software, Inc.

The types used for both the template parameters `Key` and `T` must include the following (where `T` is the `type`, `t` is a value of `T` and `u` is a `const value` of `T`):

Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

The type used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T>> >
 class multimap {

public:

// types

   typedef Key key_type;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;




   typedef typename
           Allocator::reference        reference;
   typedef typename
           Allocator::const_reference  const_reference;

   class iterator;
   class const_iterator;

   typedef typename
           Allocator::size_type        size_type;
   typedef typename
           Allocator::difference_type  difference_type;

   typedef typename std::reverse_iterator<iterator>
                        reverse_iterator;
   typedef typename std::reverse_iterator<const_iterator>
                        const_reverse_iterator;

 class value_compare
    : public binary_function<value_type, value_type, bool>

    {
     friend class multimap<Key, T, Compare, Allocator>;

     protected :
       Compare comp;
       value_compare (Compare C) : comp(c) {}
     public :
       bool operator() (const value_type&,
                        const value_type&) const;
    };

// Construct/Copy/Destroy

   explicit multimap (const Compare& = Compare(),
                      const Allocator& =
                      Allocator());
   template <class InputIterator>
    multimap (InputIterator, InputIterator,
              const Compare& = Compare(),
              const Allocator& = Allocator());
   multimap (const multimap<Key, T, Compare, Allocator>&);
   ~multimap ();
   multimap<Key, T, Compare, Allocator>& operator=
       (const multimap<Key, T, Compare, Allocator>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   allocator_type get_allocator () const;

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modifiers

   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);

   void erase (iterator);
   size_type erase (const key_type&);
   void erase (iterator, iterator);
   void swap (multimap<Key, T, Compare, Allocator>&);
   void clear ();

// Observers

   key_compare key_comp () const;
   value_compare value_comp () const;

// Multimap operations

   iterator find (const key_type&);
   const_iterator find (const key_type&) const;
   size_type count (const key_type&) const;

   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const key_type&);
   pair<const_iterator, const_iterator>
     equal_range (const key_type&) const;
};
// Non-member Operators

template <class Key, class T, class Compare,
          class Allocator>
 bool operator== (const multimap<Key, T, Compare,
                  Allocator>&,
                  const multimap<Key, T, Compare,
                  Allocator>&);

template <class Key, class T, class Compare,
          class Allocator>
 bool operator!= (const multimap<Key, T, Compare,
                  Allocator>&,
                  const multimap<Key, T, Compare,
                  Allocator>&);

template <class Key, class T, class Compare,
          class Allocator>
 bool operator< (const multimap<Key, T, Compare,
                 Allocator>&,
                 const multimap<Key, T, Compare,
                 Allocator>&);

template <class Key, class T, class Compare,
          class Allocator>
 bool operator> (const multimap<Key, T, Compare,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                Allocator>&,
                const multimap<Key, T, Compare,
                Allocator>&);

template <class Key, class T, class Compare,
          class Allocator>
 bool operator<= (const multimap<Key, T, Compare,
                  Allocator>&,
                  const multimap<Key, T, Compare,
                  Allocator>&);

template <class Key, class T, class Compare,
          class Allocator>
 bool operator>= (const multimap<Key, T, Compare,
                  Allocator>&,
                  const multimap<Key, T, Compare,
                  Allocator>&);

// Specialized Algorithms

template <class Key, class T, class Compare,
          class Allocator>
 void swap (multimap<Key, T, Compare, Allocator>&,
            multimap<Key, T, Compare, Allocator>&;
```

Constructors

```
explicit multimap(const Compare& comp = Compare(),
                  const Allocator& alloc = Allocator());
```

Constructs an empty multimap that uses the optional relation `comp` to order keys and the allocator `alloc` for all storage management.

```
template <class InputIterator>
multimap(InputIterator first,
          InputIterator last,
          const Compare& comp = Compare()
          const Allocator& alloc = Allocator());
```

Constructs a multimap containing values in the range `[first, last)`. Creation of the new multimap is only guaranteed to succeed if the iterators `first` and `last` return values of type `pair<class Key, class T>.`

```
multimap(const multimap<Key, T, Compare, Allocator>& x);
```

Creates a new multimap by copying all pairs of `key` and `value` from `x`.

Destructors

```
~multimap();
```

Releases any allocated memory for this multimap.

Assignment Operators

```
multimap<Key, T, Compare, Allocator>&
operator=(const multimap<Key, T, Compare, Allocator>& x);
```

Replaces the contents of `*this` with a copy of the multimap `x`.

Allocators

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a bidirectional `iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
const_iterator
begin() const;
```

Returns a `const_iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
iterator
end();
```

Returns a bidirectional `iterator` pointing to the last element stored in the multimap (in other words, the off-the-end value).

```
const_iterator
end() const;
```

Returns a `const_iterator` pointing to the last element stored in the multimap.

```
reverse_iterator
rbegin();
```

Returns a `reverse_iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
const_reverse_iterator
rbegin() const;
```

Returns a `const_reverse_iterator` pointing to the first element stored in the multimap.

```
reverse_iterator
rend();
```

Returns a `reverse_iterator` pointing to the last element stored in the multimap (in other words, the off-the-end value).

```
const_reverse_iterator
rend() const;
```

Returns a `const_reverse_iterator` pointing to the last element stored in the multimap.

Member Functions

```
void
clear();
```

Erases all elements from the self.

```
size_type
count(const key_type& x) const;
```

Returns the number of elements in the multimap with the key value `x`.

```
bool
empty() const;
```

Returns `true` if the multimap is empty, `false` otherwise.

```
pair<iterator,iterator>
equal_range(const key_type& x);
pair<const_iterator,const_iterator>
equal_range(const key_type& x) const;
```

Returns the pair `(lower_bound(x), upper_bound(x))`.

```
void
erase(iterator first, iterator last);
```

If the iterators `first` and `last` point to the same multimap and last is reachable from first, all elements in the range (`first, last`) are deleted from the multimap. Returns an `iterator` pointing to the element following the last

©Copyright 1999 Rogue Wave Software, Inc.

deleted element or `end()`, if there were no elements after the deleted range.

```
void
erase(iterator position);
```

Deletes the multimap element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()`, if the deleted item was the last one in this list.

```
size_type
erase(const key_type& x);
```

Deletes the elements with the key value `x` from the map, if any exist. Returns the number of deleted elements, or `0` otherwise.

```
iterator
find(const key_type& x);
```

Searches the multimap for a pair with the key value `x` and returns an `iterator` to that pair if it is found. If such a pair is not found the value `end()` is returned.

```
const_iterator
find(const key_type& x) const;
```

Same as find above but returns a `const_iterator`.

```
iterator
insert(const value_type& x);
iterator
insert(iterator position, const value_type& x);
```

`x` is inserted into the multimap. A position may be supplied as a hint regarding where to do the insertion. If the insertion is done right after `position`, then it takes amortized constant time. Otherwise it takes `O(log N)` time.

```
template <class InputIterator>
void
insert(InputIterator first, InputIterator last);
```

Copies of each element in the range `[first, last)` are inserted into the multimap. The iterators `first` and `last` must return values of type `pair<T1,T2>`. This operation takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multimap.

```
iterator
lower_bound(const key_type& x);
```

Returns an `iterator` to the first multimap element whose key is greater than or equal to `x`. If no such element exists, then `end()` is returned.

```
const_iterator
lower_bound(const key_type& x) const;
```

Same as `lower_bound` above but returns a `const_iterator.`

```
size_type
max_size() const;
```

Returns the maximum possible size of the multimap.

```
size_type
size() const;
```

Returns the number of elements in the multimap.

```
void
swap(multimap<Key, T, Compare, Allocator>& x);
```

Swaps the contents of the multimap `x` with the current multimap, `*this`.

©Copyright 1999 Rogue Wave Software, Inc.

```
iterator
upper_bound(const key_type& x);
```

Returns an `iterator` to the first element whose key is less than or equal to `x`. If no such element exists, then `end()` is returned.

```
const_iterator
upper_bound(const key_type& x) const;
```

Same as `upper_bound` above but returns a `const_iterator`.

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing `value_types` (`key,value` pairs) using the comparison operation, `Compare`, of the current multimap.

Non-member Operators

```
bool
operator==(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==)`. Otherwise it returns `false`.

```
bool
operator!=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
bool
operator<(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

```
bool
operator>(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `y < x`.

```
bool
operator<=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
bool
operator>=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template<class Key, class T, class Compare, class Allocator>
void swap(multimap<Key, T, Compare, Allocator>& a,
           multimap<Key, T, Compare, Allocator>& b);
```

Swaps the contents of `a` and `b`.

Example

```
//
// multimap.cpp
//
 #include <string>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <map>
#include <iostream>
using namespace std;

typedef multimap<int, string, less<int> > months_type;

// Print out a pair
template <class First, class Second>
ostream& operator<<(ostream& out,
                    const pair<First,Second>& p)
{
  cout << p.second << " has " << p.first << " days";
  return out;
}

// Print out a multimap
ostream& operator<<(ostream& out, months_type l)
{
  copy(l.begin(),l.end(), ostream_iterator
            <months_type::value_type,char>(cout,"\n"));
  return out;
}

int main(void)
{
  // create a multimap of months and the number of
  // days in the month
  months_type months;

  typedef months_type::value_type value_type;

  // Put the months in the multimap
  months.insert(value_type(31, string("January")));
  months.insert(value_type(28, string("February")));
  months.insert(value_type(31, string("March")));
  months.insert(value_type(30, string("April")));
  months.insert(value_type(31, string("May")));
  months.insert(value_type(30, string("June")));
  months.insert(value_type(31, string("July")));
  months.insert(value_type(31, string("August")));
  months.insert(value_type(30, string("September")));
  months.insert(value_type(31, string("October")));
  months.insert(value_type(30, string("November")));
  months.insert(value_type(31, string("December")));


  // print out the months
  cout << "All months of the year" << endl << months
       << endl;

  // Find the Months with 30 days
  pair<months_type::iterator,months_type::iterator> p =
         months.equal_range(30);

  // print out the 30 day months
  cout << endl << "Months with 30 days" << endl;
  copy(p.first,p.second,
       ostream_iterator<months_type::value_type,char>
      (cout,"\n"));

  return 0;
}
```

Program Output

```
All months of the year
February has 28 days
April has 30 days
June has 30 days
September has 30 days
November has 30 days
January has 31 days
March has 31 days
May has 31 days
July has 31 days
August has 31 days
October has 31 days
```

©Copyright 1999 Rogue Wave Software, Inc.

```
December has 31 days

Months with 30 days
April has 30 days
June has 30 days
September has 30 days
November has 30 days
```

Warnings

Member function templates are used in all containers included in the Standard Template Library. An example of this feature is the constructor for ***multimap<Key,T,Compare,Allocator>*** that takes two templatized iterators:

```
template <class InputIterator>
 multimap (InputIterator, InputIterator,
           const Compare& = Compare(),
           const Allocator& = Allocator());
```

multimap also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a multimap in the following two ways:

```
multimap<int,int>::value_type intarray[10];
multimap<int,int> first_map(intarry, intarray + 10);
multimap<int,int> second_multimap(first_multimap.begin(),
                  first_multimap.end());
```

but not this way:

```
multimap<long,long>
 long_multimap(first_multimap.begin(),first_multimap.end());
```

since the `long_multimap` and `first_multimap` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you have to write:

```
multimap<int, int, less<int>, allocator<int> >
```

instead of:

```
multimap<int, int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Containers***, ***Iterators***, ***map***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

multiplies

`Function Object`

Summary

A binary function object that returns the result of multiplying its first and second arguments.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include<functional>
template <class T>
struct multiplies : binary_function<T, T, T> {
  T operator() (const T&, const T&) const;
};
```

Description

multiplies is a binary function object. Its `operator()` returns the result of multiplying `x` and `y`. You can pass a ***multiplies*** object to any algorithm that uses a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***multiplies*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vec2.end(),
          vecResult.begin(), multiplies<int>());
```

After this call to ***transform***, `vecResult(n)` contains `vec1(n)` times `vec2(n)`.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you have to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

multiset

`Container Class`

Summary

An associative container that allows fast access to stored key values. Storage of duplicate keys is allowed. A ***multiset*** supports bidirectional iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

			operator=()
begin()	erase()	max_size()	operator==()
clear()	find()	operator!=()	rbegin()
count()	get_allocator()	operator>()	rend()
empty()	insert()	operator>=()	size()
end()	key_comp()	operator<()	swap()
equal_range()	lower_bound()	operator<=()	upper_bound()
			value_comp()

Synopsis

```
#include <set>
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
class multiset;
```

Description

multiset <Key, Compare, Allocator> allows fast access to stored key values. The default operation for key comparison is the < operator. Insertion of duplicate keys is allowed with a multiset.

multiset uses bidirectional iterators that point to a stored key.

©Copyright 1999 Rogue Wave Software, Inc.

Any type used for the template parameter `Key` must include the following (where `T` is the `type`, `t` is a value of `T` and `u` is a `const value` of `T`):

Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
 class multiset {

public:

// typedefs

   typedef Key key_type;
   typedef Key value_type;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef Allocator allocator_type;




   typedef typename
           Allocator::reference        reference;
   typedef typename
           Allocator::const_reference  const_reference;

   class iterator;
   class const_iterator;

   typedef typename
           Allocator::size_type        size_type;
   typedef typename
           Allocator::difference_type  difference_type;

   typedef typename std::reverse_iterator<iterator>
                        reverse_iterator;
   typedef typename std::reverse_iterator<const_iterator>
                        const_reverse_iterator;

// Construct/Copy/Destroy

   explicit multiset (const Compare& = Compare(),
                      const Allocator& = Allocator());
   template <class InputIterator>
    multiset (InputIterator, InputIterator,
              const Compare& = Compare(),
              const Allocator& = Allocator());
   multiset (const multiset<Key, Compare, Allocator>&);
   ~multiset ();
   multiset<Key, Compare, Allocator>&
            operator= (const multiset<Key,
                       Compare, Allocator>&);

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modifiers

   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);

   void erase (iterator);
   size_type erase (const key_type&);
   void erase (iterator, iterator);
   void swap (multiset<Key, Compare, Allocator>&);
   void clear ();

// Observers

   key_compare key_comp () const;
   value_compare value_comp () const;

// Multiset operations

   iterator find (const key_type&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range
       (const key_type&) const;
   };

// Non-member Operators

template <class Key, class Compare, class Allocator>
 bool operator==
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator!=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator<
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator>
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator<=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator>=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

// Specialized Algorithms

template <class Key, class Compare, class Allocator>
 void swap ( multiset<Key, Compare, Allocator>&,
             multiset<Key, Compare, Allocator>&);
```

Constructors

```
explicit multiset(const Compare& comp = Compare(),
                  const Allocator& alloc = Allocator());
```

©Copyright 1999 Rogue Wave Software, Inc.

Constructs an empty multiset that uses the optional relation `comp` to order keys, if it is supplied, and the allocator `alloc` for all storage management.

```
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
         const Compare& = Compare(),
         const Allocator& = Allocator());
```

Constructs a multiset containing values in the range `[first, last)`.

```
multiset(const multiset<Key, Compare, Allocator>& x);
```

Creates a new multiset by copying all key values from `x`.

Destructors

```
~multiset();
```

Releases any allocated memory for this multiset.

Assignment Operators

```
multiset<Key, Compare, Allocator>&
operator=(const multiset<Key, Compare, Allocator>& x);
```

Replaces the contents of `*this` with a copy of the contents of `x`.

Allocators

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

Returns an `iterator` pointing to the first element stored in the multiset. "First" is defined by the multiset's comparison operator, `Compare`.

```
const_iterator
begin();
```

Returns a `const_iterator` pointing to the first element stored in the multiset.

```
iterator
end();
```

Returns an `iterator` pointing to the last element stored in the multiset (in other words, the off-the-end value).

```
const_iterator
end();
```

Returns a `const_iterator` pointing to the last element stored in the multiset (in other words, the off-the-end value).

```
reverse_iterator
rbegin();
```

Returns a `reverse_iterator` pointing to the first element stored in the multiset. "First" is defined by the multiset's comparison operator, `Compare`.

```
const_reverse_iterator
rbegin();
```

Returns a `const_reverse_iterator` pointing to the first element stored in the multiset.

©Copyright 1999 Rogue Wave Software, Inc.

```
reverse_iterator
rend();
```

Returns a `reverse_iterator` pointing to the last element stored in the multiset (in other words, the off-the-end value).

```
const_reverse_iterator
rend();
```

Returns a `const_reverse_iterator` pointing to the last element stored in the multiset (in other words, the off-the-end value).

Member Functions

```
void
clear();
```

Erases all elements from the self.

```
size_type
count(const key_type& x) const;
```

Returns the number of elements in the multiset with the key value `x`.

```
bool
empty() const;
```

Returns `true` if the multiset is empty, `false` otherwise.

```
pair<iterator,iterator>
equal_range(const key_type& x)const;
```

Returns the pair `(lower_bound(x), upper_bound(x))`.

```
size_type
erase(const key_type& x);
```

Deletes all elements with the key value `x` from the multiset, if any exist. Returns the number of deleted elements.

```
void
erase(iterator position);
```

Deletes the multiset element pointed to by the iterator `position`. Returns an iterator pointing to the element following the deleted element, or `end()`, if the deleted item was the last one in this list.

```
void
erase(iterator first, iterator last);
```

If the iterators `first` and `last` point to the same multiset and last is reachable from first, all elements in the range (`first, last`) are deleted from the multiset. Returns an iterator pointing to the element following the last deleted element or `end()`, if there were no elements after the deleted range.

```
iterator
find(const key_type& x) const;
```

Searches the multiset for a key value `x` and returns an iterator to that key if it is found. If such a value is not found, the iterator `end()` is returned.

```
iterator
insert(const value_type& x);
iterator
insert(iterator position, const value_type& x);
```

`x` is inserted into the multiset. A position may be supplied as a hint regarding where to do the insertion. If the insertion is done right after position, then it takes amortized constant time. Otherwise, it takes `O(log N)` time.

```
template <class InputIterator>
void
insert(InputIterator first, InputIterator last);
```

©Copyright 1999 Rogue Wave Software, Inc.

Copies of each element in the range `[first, last)` are inserted into the multiset. This `insert` takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multiset.

```
iterator
lower_bound(const key_type& x) const;
```

Returns an iterator to the first element whose key is greater than or equal to `x`. If no such element exists, `end()` is returned.

```
size_type
max_size() const;
```

Returns the maximum possible size of the multiset `size_type`.

```
size_type
size() const;
```

Returns the number of elements in the multiset.

```
void
swap(multiset<Key, Compare, Allocator>& x);
```

Swaps the contents of the multiset `x` with the current multiset, `*this`.

```
iterator
upper_bound(const key_type& x) const;
```

Returns an iterator to the first element whose key is smaller than or equal to `x`. If no such element exists, then `end()` is returned.

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multiset.

Non-member Operators

```
template <class Key, class Compare, class Allocator>
operator==(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==)`. Otherwise it returns `false`.

```
template <class Key, class Compare, class Allocator>
operator!=(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class Key, class Compare, class Allocator>
operator<(const multiset<Key, Compare, Allocator>& x,
          const multiset<Key, Compare, Allocator>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

```
template <class Key, class Compare, class Allocator>
operator>(const multiset<Key, Compare, Allocator>& x,
          const multiset<Key, Compare, Allocator>& y);
```

Returns `y < x`.

```
template <class Key, class Compare, class Allocator>
operator<=(const multiset<Key, Compare, Allocator>& x,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
          const multiset<Key, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class Key, class Compare, class Allocator>
operator>=(const multiset<Key, Compare, Allocator>& x,
          const multiset<Key, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>& a,
          multiset<Key,Compare,Allocator>&b);
```

Swaps the contents of `a` and `b`.

Example

```
//
// multiset.cpp
//
#include <set>
#include <iostream>
using namespace std;

 typedef multiset<int, less<int>, allocator> set_type;

 ostream& operator<<(ostream& out, const set_type& s)
 {
   copy(s.begin(),s.end(),
     ostream_iterator<set_type::value_type,char>(cout," "));
   return out;
 }


 int main(void)
 {
   // create a multiset of ints
   set_type  si;
   int  i;

   for (int j = 0; j < 2; j++)
   {
     for(i = 0; i < 10; ++i) {
       // insert values with a hint
       si.insert(si.begin(), i);
     }
   }

   // print out the multiset
   cout << si << endl;

   // Make another int multiset and an empty multiset
   set_type si2, siResult;
   for (i = 0; i < 10; i++)
      si2.insert(i+5);
   cout << si2 << endl;

   // Try a couple of set algorithms
   set_union(si.begin(),si.end(),si2.begin(),si2.end(),
          inserter(siResult,siResult.begin()));
   cout << "Union:" << endl << siResult << endl;

   siResult.erase(siResult.begin(),siResult.end());
   set_intersection(si.begin(),si.end(),
          si2.begin(),si2.end(),
          inserter(siResult,siResult.begin()));
   cout << "Intersection:" << endl << siResult << endl;

   return 0;
 }
```

©Copyright 1999 Rogue Wave Software, Inc.

Program Output

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 6 7 8 9 10 11 12 13 14
Union:
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 11 12 13 14
Intersection:
5 6 7 8 9
```

Warnings

Member function templates are used in all containers included in the Standard Template Library. An example of this feature is the constructor for ***multiset<Key, Compare, Allocator>***, which takes two templatized iterators:

```
template <class InputIterator>
multiset (InputIterator, InputIterator,
          const Compare& = Compare(),
          const Allocator& = Allocator());
```

multiset also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on). You can also use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a ***multiset*** in the following two ways:

```
int intarray[10];
multiset<int> first_multiset(intarray, intarray +10);
multiset<int> second_multiset(first_multiset.begin(),
              first_multiset.end());
```

but not this way:

```
multiset<long>
 long_multiset(first_multiset.begin(),first_multiset.end());
```

since the `long_multiset` and `first_multiset` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you have to write:

```
multiset<int, less<int>, allocator<int> >
```

instead of:

```
multiset<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Containers***, ***Iterators***, ***set***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

negate

`Function Object`

Summary

Unary function object that returns the negation of its argument.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct negate : public unary_function<T, T>;
```

Description

negate is a unary function object. Its `operator()` returns the negation of its argument: `true` if its argument is `false`, or `false` if its argument is `true`. You can pass a ***negate*** object to any algorithm that requires a unary function. For example, the ***transform*** algorithm applies a unary operation to the values in a collection and stores the result. ***negate*** could be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(), vecResult.begin(),
          negate<int>());
```

After this call to ***transform***, `vecResult(n)` contains the negation of the element in `vec1(n)`.

Interface

```
template <class T>
struct negate : unary_function<T, T> {
  T operator() (const T&) const;
};
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects, ***unary_function***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Negators

`Function Object`

Summary

Function adaptors and function objects used to reverse the sense of predicate function objects.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Predicate>
class unary_negate;

template <class Predicate>
unary_negate<Predicate> not1(const Predicate&);

template <class Predicate>
class binary_negate;

template <class Predicate>
binary_negate<Predicate> not2(const Predicate&);
```

Description

Negators ***not1*** and ***not2*** are functions that take predicate function objects as arguments and return predicate function objects with the opposite sense. Negators work only with function objects defined as subclasses of the classes ***unary_function*** and ***binary_function***. ***not1*** accepts and returns unary predicate function objects. ***not2*** accepts and returns binary predicate function objects.

unary_negate and ***binary_negate*** are function object classes that include return types for the negators, ***not1*** and ***not2***.

Interface

```
template <class Predicate>
class unary_negate
  : public unary_function
           <typename Predicate::argument_type, bool> {

public:
  explicit unary_negate (const Predicate&);
  bool operator() (const argument_type&) const;
};

template<class Predicate>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
unary_negate <Predicate> not1 (const Predicate&);

template<class Predicate>
class binary_negate
  : public binary_function
           <typename Predicate::first_argument_type,
            typename Predicate::second_argument_type, bool>
{
public:
  explicit binary_negate (const Predicate&);
  bool operator() (const first_argument_type&,
                   const second_argument_type&) const;
};

template <class Predicate>
binary_negate<Predicate> not2 (const Predicate&);
```

Example

```
//
// negator.cpp
//
 #include<functional>
 #include<algorithm>
 #include <iostream>
 using namespace std;

 //Create a new predicate from unary_function
 template<class Arg>
 class is_odd : public unary_function<Arg, bool>
 {
   public:
   bool operator()(const Arg& arg1) const
   {
     return (arg1 % 2 ? true : false);
   }
 };

 int main()
 {
   less<int> less_func;

   // Use not2 on less
   cout << (less_func(1,4) ? "TRUE" : "FALSE") << endl;
   cout << (less_func(4,1) ? "TRUE" : "FALSE") << endl;
   cout << (not2(less<int>())(1,4) ? "TRUE" : "FALSE")
        << endl;
   cout << (not2(less<int>())(4,1) ? "TRUE" : "FALSE")
        << endl;

   //Create an instance of our predicate
   is_odd<int> odd;

   // Use not1 on our user defined predicate
   cout << (odd(1) ? "TRUE" : "FALSE") << endl;
   cout << (odd(4) ? "TRUE" : "FALSE") << endl;
   cout << (not1(odd)(1) ? "TRUE" : "FALSE") << endl;
   cout << (not1(odd)(4) ? "TRUE" : "FALSE") << endl;

   return 0;
 }
```

Program Output

```
TRUE
FALSE
FALSE
TRUE
TRUE
FALSE
FALSE
TRUE
```

©Copyright 1999 Rogue Wave Software, Inc.

See Also

Algorithms, *binary_function*, *Function Objects*, *unary_function*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

next_permutation

`Algorithm`

Summary

Generates successive permutations of a sequence based on an ordering function.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator>
bool next_permutation (BidirectionalIterator first,
                       BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
 bool next_permutation (BidirectionalIterator first,
                        BidirectionalIterator last,
                        Compare comp);
```

Description

The permutation-generating algorithms (***next_permutation*** and ***prev_permutation***) assume that the set of all permutations of the elements in a sequence is lexicographically sorted with respect to `operator<` or `comp`. For example, if a sequence includes the integers 1 2 3, that sequence has six permutations. In order from first to last, they are: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, and 3 2 1.

The ***next_permutation*** algorithm takes a sequence defined by the range `[first, last)` and transforms it into its next permutation, if possible. If such a permutation does exist, the algorithm completes the transformation and returns `true`. If the permutation does not exist, ***next_permutation*** returns `false`, and transforms the permutation into its "first" permutation. ***next_permutation*** does the transformation according to the lexicographical ordering defined by either `operator<` (the default used in the first version of the algorithm) or `comp` (which is user-supplied in the second version of the algorithm).

For example, if the sequence defined by `[first, last)` contains the integers 3 2 1 (in that order), there is *not* a "next permutation." Therefore, the algorithm transforms the sequence into its first permutation (1 2 3) and returns `false`.

Complexity

At most `(last - first)/2` swaps are performed.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// permute.cpp
//
 #include <numeric>    //for accumulate
 #include <vector>        //for vector
 #include <functional> //for less
 #include <iostream>
 using namespace std;

 int main()
 {
   //Initialize a vector using an array of ints
   int  a1[] = {0,0,0,0,1,0,0,0,0,0};
   char a2[] = "abcdefghji";
   //Create the initial set and copies for permuting
   vector<int>  m1(a1, a1+10);
   vector<int>  prev_m1((size_t)10), next_m1((size_t)10);
   vector<char> m2(a2, a2+10);
   vector<char> prev_m2((size_t)10), next_m2((size_t)10);
   copy(m1.begin(), m1.end(), prev_m1.begin());
   copy(m1.begin(), m1.end(), next_m1.begin());
   copy(m2.begin(), m2.end(), prev_m2.begin());
   copy(m2.begin(), m2.end(), next_m2.begin());
   //Create permutations
   prev_permutation(prev_m1.begin(),
                    prev_m1.end(),less<int>());
   next_permutation(next_m1.begin(),
                    next_m1.end(),less<int>());
   prev_permutation(prev_m2.begin(),
                    prev_m2.end(),less<int>());
   next_permutation(next_m2.begin(),
                    next_m2.end(),less<int>());
   //Output results
   cout << "Example 1: " << endl << "     ";
   cout << "Original values:      ";
   copy(m1.begin(),m1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << "     ";
   cout << "Previous permutation: ";
   copy(prev_m1.begin(),prev_m1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl<< "     ";
   cout << "Next Permutation:     ";
   copy(next_m1.begin(),next_m1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   cout << "Example 2: " << endl << "     ";
   cout << "Original values: ";
   copy(m2.begin(),m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << "     ";
   cout << "Previous Permutation: ";
   copy(prev_m2.begin(),prev_m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << "     ";
   cout << "Next Permutation:     ";
   copy(next_m2.begin(),next_m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << endl;

   return 0;
 }
```

Program Output

```
Example 1:
     Original values:      0 0 0 0 1 0 0 0 0 0
     Previous permutation: 0 0 0 0 0 1 0 0 0 0
     Next Permutation:     0 0 0 1 0 0 0 0 0 0
Example 2:
     Original values: a b c d e f g h j i
     Previous Permutation: a b c d e f g h i j
     Next Permutation:     a b c d e f g i h j
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, the you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

prev_permutation

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

not1

`Function Adaptor`

Summary

A function adaptor used to reverse the sense of a unary predicate function object.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template<class Predicate>
unary_negate <Predicate> not1 (const Predicate&);
```

Description

not1 is a function adaptor, known as a negator, that takes a unary predicate function object as its argument and returns a unary predicate function object that is the complement of the original. ***unary_negate*** is a function object class that includes a return type for the ***not1*** negator.

Note that ***not1*** works only with function objects that are defined as subclasses of the class ***unary_function***.

See Also

Negators, ***not2***, ***unary_function***, ***unary_negate***, ***pointer_to_unary_function***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

not2

`Function Adaptor`

Summary

A function adaptor used to reverse the sense of a binary predicate function object.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Description

not2 is a function adaptor, known as a negator, that takes a binary predicate function object as its argument and returns a binary predicate function object that is the complement of the original. ***binary_negate*** is a function object class that includes a return type for the ***not2*** negator.

Note that ***not2*** works only with function objects that are defined as subclasses of the class ***binary_function***.

See Also

binary_function, ***binary_negate***, ***Negators***, ***not1***, ***pointer_to_binary_function***, ***unary_negate***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

not_equal_to

`Function Object`

Summary

A binary function object that returns `true` if its first argument is not equal to its second.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class T>
struct not_equal_to : public binary_function<T, T, bool>;
```

Description

not_equal_to is a binary function object. Its `operator()` returns `true` if `x` is not equal to `y`. You can pass a ***not_equal_to*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***not_equal_to*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), not_equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` contains a `1` if `vec1(n)` was not equal to `vec2(n)` or a `0` if `vec1(n)` was equal to `vec2(n)`.

Interface

```
template <class T>
struct not_equal_to : binary_function<T, T, bool> {
  bool operator() (const T&, const T&) const;
};
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function*,*Function Objects

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

nth_element

`Algorithm`

Summary

Rearranges a collection so that all elements lower in sorted order than the `nth` element come before it and all elements higher in sorter order than the `nth` element come after it.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
 void nth_element (RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
 void nth_element (RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last,
                   Compare comp);
```

Description

The ***nth_element*** algorithm rearranges a collection according to either the default comparison operator (`>`) or a comparison operator given by the user. After the algorithm is applied, three things are `true`:

- The element that would be in the `nth` position if the collection were completely sorted is in the `nth` position
- All elements prior to the `nth` position would also precede that position in an ordered collection
- All elements following the `nth` position would also follow that position in an ordered collection

That is, for any iterator `i` in the range `[first, nth)` and any iterator `j` in the range `[nth, last)`, it holds that `!(*i > *j)` or `comp(*i, *j) == false`.

Note that the elements that precede or follow the `nth` position are not necessarily sorted relative to each other. The ***nth_element*** algorithm does *not* sort the entire collection.

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

The algorithm is linear, on average, where `N` is the size of the range `[first, last)`.

Example

```
//
// nthelem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 template<class RandomAccessIterator>
 void quik_sort(RandomAccessIterator start,
                RandomAccessIterator end)
 {
   size_t dist = 0;
   distance(start, end, dist);

   //Stop condition for recursion
   if(dist > 2)
   {
     //Use nth_element to do all the work for quik_sort
     nth_element(start, start+(dist/2), end);

     //Recursive calls to each remaining unsorted portion
     quik_sort(start, start+(dist/2-1));
     quik_sort(start+(dist/2+1), end);
   }

   if(dist == 2 && *end < *start)
     swap(start, end);
 }

 int main()
 {
   //Initialize a vector using an array of ints
   int arr[10] = {37, 12, 2, -5, 14, 1, 0, -1, 14, 32};
   vector<int> v(arr, arr+10);

   //Print the initial vector
   cout << "The unsorted values are: " << endl << "     ";
   vector<int>::iterator i;
   for(i = v.begin(); i != v.end(); i++)
     cout << *i << ", ";
   cout << endl << endl;

   //Use the new sort algorithm
   quik_sort(v.begin(), v.end());

   //Output the sorted vector
   cout << "The sorted values are: " << endl << "     ";
   for(i = v.begin(); i != v.end(); i++)
     cout << *i << ", ";
   cout << endl << endl;

   return 0;
 }
```

Program Output

```
The unsorted values are:
     37, 12, 2, -5, 14, 1, 0, -1, 14, 32,
The sorted values are:
     -5, -1, 0, 1, 2, 12, 14, 14, 32, 37,
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

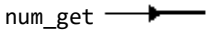
See Also

Algorithms

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

num_get



Summary

A numeric formatting facet for input.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
iter_type

Member Functions

do_get()
get()

Synopsis

```
#include <locale>
template <class charT, class InputIterator > class num_get;
```

Description

The ***num_get*** facet allows for formatted input of numbers. ***basic_istream*** and all other input-oriented streams use this facet to implement formatted numeric input.

Interface

```
template <class charT, class InputIterator =
  istreambuf_iterator<charT> >
class num_get : public locale::facet {
public:
  typedef charT            char_type;
  typedef InputIterator    iter_type;
  explicit num_get(size_t refs = 0);
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, bool&)          const;
  iter_type get(iter_type, iter_type, ios_base& ,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                ios_base::iostate&, long&)          const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, unsigned short&) const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, unsigned int&)  const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, unsigned long&) const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, float&)         const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, double&)        const;
  iter_type get(iter_type, iter_type, ios_base&,
                ios_base::iostate&, long double&)   const;
  static locale::id id;

protected:
  ~num_get();  // virtual
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&, bool&) const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&, long&) const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&,
                           unsigned short&) const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&,
                           unsigned int&) const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&,
                           unsigned long&) const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&, float&)
                           const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&, double&)
                           const;
  virtual iter_type do_get(iter_type, iter_type, ios_base&,
                           ios_base::iostate&,
                           long double&) const;
};
```

Types

`char_type`

Type of character the facet is instantiated on.

`iter_type`

Type of iterator used to scan the character buffer.

Constructors

`explicit num_get(size_t refs = 0)`

Construct a ***num_get*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

Destructors

`~num_get();  // virtual and protected`

Destroys the facet.

Facet ID

`static locale::id id;`

Unique identifier for this type of facet.

©Copyright 1999 Rogue Wave Software, Inc.

Public Member Functions

The public members of the ***num_get*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `get` function simply calls its protected cousin `do_get`.

```
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, bool& v)          const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, long& v)          const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, unsigned short& v) const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, unsigned int& v)  const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, unsigned long& v) const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, float& v) const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, double& v) const;
iter_type
get(iter_type in, iter_type end, ios_base& io,
    ios_base::iostate& err, long double& v) const;
```

Each of the eight overloads of the `get` function simply call the corresponding `do_get` function.

Protected Member Functions

```
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err, bool& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err, long& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err,
       unsigned short& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err,
       unsigned int& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err,
       unsigned long& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err, float& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& err, double& v) const;
virtual iter_type
do_get(iter_type in, iter_type end, ios_base& io,
       ios_base::iostate& long double& v) const;
```

The eight overloads of the `do_get` member function all take a sequence of characters `[int,end)`, and extract a numeric value. The numeric value is returned in `v`. The `io` argument is used to obtain formatting information and the `err` argument is used to set error conditions in a calling stream.

Example

```
//
// numget.cpp
//
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <sstream>

int main ()
{
  using namespace std;

 typedef istreambuf_iterator<char,char_traits<char> >
  iter_type;

  locale loc;
  ios_base::iostate state;
  bool bval = false;
  long lval = 0L;
  long double ldval = 0.0;
  iter_type end;

  // Get a num_get facet
  const num_get<char,iter_type>& tg =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<num_get<char,iter_type> >(loc);
#else
  use_facet(loc,(num_get<char,iter_type>*)0);
#endif

  {
    // Build an istringstream from the buffer and construct
    // beginning and ending iterators on it.
    istringstream ins("true");
    iter_type begin(ins);

    // Get a bool value
    tg.get(begin,end,ins,state,bval);
  }
  cout << bval << endl;
  {
    // Get a long value
    istringstream ins("2422235");
    iter_type begin(ins);
    tg.get(begin,end,ins,state,lval);
  }
  cout << lval << endl;
  {
    // Get a long double value
    istringstream ins("32324342.98908");
    iter_type begin(ins);
    tg.get(begin,end,ins,state,ldval);
  }
  cout << ldval << endl;
  return 0;
}
```

See Also

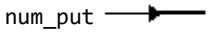
locale, *facets*, *num_put*, *numpunct*, *ctype*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

num_put



Summary

A numeric formatting facet for output.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
iter_type

Member Functions

do_put()
put()

Synopsis

```
#include <locale>
template <class charT, class OutputIterator> class num_put;
```

Description

The ***num_put<charT,OutputIterator>*** facet allows for formatted output of numbers. ***basic_ostream*** and all other output-oriented streams use this facet to implement formatted numeric output.

Interface

```
template <class charT, class OutputIterator =
  ostreambuf_iterator<charT> >
class num_put : public locale::facet {
public:
  typedef charT            char_type;
  typedef OutputIterator   iter_type;
  explicit num_put(size_t = 0);

  iter_type put(iter_type, ios_base&, char_type, bool)
            const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  iter_type put(iter_type, ios_base&, char_type, long)
            const;
  iter_type put(iter_type, ios_base&, char_type,
                unsigned long) const;
  iter_type put(iter_type, ios_base&, char_type,
                double) const;
  iter_type put(iter_type, ios_base&, char_type,
                long double) const;
  static locale::id id;

protected:
  ~num_put();  // virtual
  virtual iter_type do_put(iter_type, ios_base&, char_type,
                           bool) const;
  virtual iter_type do_put(iter_type, ios_base&, char_type,
                           long) const;
  virtual iter_type do_put(iter_type, ios_base&, char_type,
                           unsigned long) const;
  virtual iter_type do_put(iter_type, ios_base&, char_type,
                           double) const;
  virtual iter_type do_put(iter_type, ios_base&, char_type,
                           long double) const;
};
```

Types

char_type

Type of character upon which the facet is instantiated.

iter_type

Type of iterator used to scan the character buffer.

Constructors

```
explicit num_put(size_t refs = 0)
```

Constructs a ***num_put*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

Destructors

```
~num_put();  // virtual and protected
```

Destroys the facet.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***num_put*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `put` function simply calls its protected cousin `do_put`.

```
iter_type
put(iter_type s, ios_base& io, char_type fill, bool v)
    const;
iter_type
put(iter_type s, ios_base& io, char_type fill, long v)
    const;
iter_type
put(iter_type s, ios_base& io, char_type fill,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    unsigned long v) const;
iter_type
put(iter_type s, ios_base& io, char_type fill, double v)
    const;
iter_type
put(iter_type s, ios_base& io, char_type fill,
    long double v) const;
```

Each of the five overloads of the `put` function simply call the corresponding `do_put` function.

Protected Member Functions

```
virtual iter_type
do_put(iter_type s, ios_base& io,
       char_type fill, bool v) const;
virtual iter_type
do_put(iter_type s, ios_base& io,
       char_type fill, long v) const;
virtual iter_type
do_put(iter_type s, ios_base& io,
       char_type fill,unsigned long) const;
virtual iter_type
do_put(iter_type s, ios_base& io,
       char_type fill, double v) const;
virtual iter_type
do_put(iter_type s, ios_base& io,
       char_type fill,long double v) const;
```

The five overloads of the `do_put` member function all take a numeric value and output a formatted character string representing that value. The character string is output through the `s` argument to the function. The `io` argument is used to obtain formatting specifications, and the `fill` argument determines the character to use in padding.

Example

```
//
// numput.cpp
//

#include <iostream>

int main ()
{
  using namespace std;

  typedef ostreambuf_iterator<char,char_traits<char> >
   iter_type;

  locale loc;
  bool bval = true;
  long lval = 422432L;
  unsigned long ulval = 12328889UL;
  double dval = 10933.8934;
  long double ldval = 100028933.8934;

  // Construct a ostreambuf_iterator on cout
  iter_type begin(cout);

  // Get a num_put facet reference
  const num_put<char,iter_type>& np =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<num_put<char,iter_type> >(loc);
#else
  use_facet(loc,(num_put<char,iter_type>*)0);
#endif

  // Put out a bool
  cout << bval << " --> ";
  np.put(begin,cout,' ',bval);

  // Put out a long
  cout << endl << lval << " --> ";
  np.put(begin,cout,' ',lval);

  // Put out an unsigned long
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  cout << endl << ulval << " --> ";
  np.put(begin,cout,' ',ulval);

  // Put out a double
  cout << endl << dval << " --> ";
  np.put(begin,cout,' ',dval);

  // Put out a long double
  cout << endl << ldval << " --> ";
  np.put(begin,cout,' ',ldval);

  cout <<  endl;

  return 0;
}
```

See Also

locale, ***facets***, ***numget***, ***numpunct***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

numeric_limits

```
Numeric Limits Library
```


Summary

A class for representing information about scalar types.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

digits	is_bounded	is_specialized	
digits10	is_exact	max_exponent	round_style
has_denorm	is_iec559	max_exponent10	tinyness_before
has_infinity	is_integer	min_exponent	traps
has_quiet_NaN	is_modulo	min_exponent10	
has_signaling_NaN	is_signed	radix	

Member Functions

denorm_min()	
epsilon()	
infinity()	round_error()
max()	signaling_NaN()
min()	
quiet_NaN()	

Specializations

```
numeric_limits<float>
numeric_limits<double>
numeric_limits<long double>
numeric_limits<short>
numeric_limits<unsigned short>
numeric_limits<int>
numeric_limits<unsigned int>
numeric_limits<long>
numeric_limits<unsigned long>
numeric_limits<char>
numeric_limits<wchar_t>
numeric_limits<unsigned char>
numeric_limits<signed char>
numeric_limits<bool>
```

Synopsis

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <limits>
template <class T>
class numeric_limits ;
```

Description

numeric_limits is a class for representing information about scalar types. Specializations are included for each fundamental type, both floating point and integer, including `bool`.

This class encapsulates information that is contained in the `<climits>` and `<cfloat>` headers, and includes additional information that is not contained in any existing C or C++ header.

Not all of the information given by members is meaningful for all specializations of ***numeric_limits***. Any value that is not meaningful for a particular type is set to `0` or `false`.

Interface

```
template <class T>
 class numeric_limits {

 public:

 // General -- meaningful for all specializations.

    static const bool is_specialized ;
    static T min () throw();
    static T max () throw();
    static const int radix ;
    static const int digits ;
    static const int digits10 ;
    static const bool is_signed ;
    static const bool is_integer ;
    static const bool is_exact ;
    static const bool traps ;
    static const bool is_modulo ;
    static const bool is_bounded ;

 // Floating point specific.

    static T epsilon () throw();
    static T round_error () throw();
    static const int min_exponent10 ;
    static const int max_exponent10 ;
    static const int min_exponent ;
    static const int max_exponent ;
    static const bool has_infinity ;
    static const bool has_quiet_NaN ;
    static const bool has_signaling_NaN ;
    static const bool is_iec559 ;
    static const float_denorm_style has_denorm ;
    static const bool has_denorm_loss;
    static const bool tinyness_before ;
    static const float_round_style round_style ;
    static T denorm_min () throw();
    static T infinity () throw();
    static T quiet_NaN () throw();
    static T signaling_NaN () throw();
 };

 enum float_round_style {
   round_indeterminate       = -1,
   round_toward_zero         =  0,
   round_to_nearest          =  1,
   round_toward_infinity     =  2,
   round_toward_neg_infinity =  3
 };
 enum float_denorm_style {
   denorm_indeterminate      = -1,
   denorm_absent             =  0,
   denorm_present            =  1
 };
```

©Copyright 1999 Rogue Wave Software, Inc.

Member Fields and Functions

```
static T
denorm_min () throw();
```

Returns the minimum denormalized value. Meaningful for all floating point types. For types that do not allow denormalized values, this method must return the minimum normalized value.

```
static const int
digits ;
```

The number of radix digits that can be represented without change. For built-in integer types, `digits` is usually the number of non-sign bits in the representation. For floating point types, `digits` is the number of radix digits in the mantissa. This member is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static const int
digits10 ;
```

The number of base 10 digits that can be represented without change. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static T
epsilon () throw();
```

Returns the machine epsilon (the difference between `1` and the least value greater than `1` that is representable). This function is meaningful for floating point types only.

```
static const float_denorm_style
has_denorm ;
```

Returns `denorm_present` if the type allows denormalized values. Returns `denorm_absent` if the type does not allow denormalized values. Returns `denorm_indeterminate` if it is indeterminate at compile time whether the type allows denormalized values. It is meaningful for floating point types only.

```
static const bool
has_infinity ;
```

This field is `true` if the type has a representation for positive infinity. It is meaningful for floating point types only. This field must be `true` for any type claiming conformance to IEC 559.

```
static const bool
has_quiet_NaN ;
```

This field is `true` if the type has a representation for a quiet (non-signaling) "Not a Number". It is meaningful for floating point types only and must be `true` for any type claiming conformance to IEC 559.

```
static const bool
has_signaling_NaN ;
```

This field is `true` if the type has a representation for a signaling "Not a Number". It is meaningful for floating point types only, and must be `true` for any type claiming conformance to IEC 559.

```
static T
infinity () throw();
```

Returns the representation of positive infinity, if available. This member function is meaningful for only those specializations that declare `has_infinity` to be `true`. Required for any type claiming conformance to IEC 559.

```
static const bool
is_bounded ;
```

This field is `true` if the set of values representable by the type is finite. All built-in C types are bounded; this member would be `false` for arbitrary precision types.

```
static const bool
is_exact ;
```

©Copyright 1999 Rogue Wave Software, Inc.

This static member field is `true` if the type uses an exact representation. All integer types are exact, but not vice versa. For example, rational and fixed-exponent representations are exact but not integer. This member is meaningful for all specializations.

```
static const bool
is_iec559 ;
```

This member is `true` if and only if the type adheres to the IEC 559 standard. It is meaningful for floating point types only.

```
static const bool
is_integer ;
```

This member is `true` if the type is integer. This member is meaningful for all specializations.

```
static const bool
is_modulo ;
```

This field is `true` if the type is modulo. Generally, this is `false` for floating types, `true` for unsigned integers, and `true` for signed integers on most machines. A type is modulo if it is possible to add two positive numbers and have a result that wraps around to a third number, which is less.

```
static const bool
is_signed ;
```

This member is `true` if the type is signed. This member is meaningful for all specializations.

```
static const bool
is_specialized ;
```

Indicates whether ***numeric_limits*** has been specialized for type `T`. This flag must be `true` for all specializations of `numeric_limits`. For the default ***numeric_limits<T>*** template, this flag must be `false`.

```
static T
max () throw();
```

Returns the maximum finite value. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static const int
max_exponent ;
```

The maximum positive integer such that the radix raised to the power one less than that integer is in range. This field is meaningful for floating point types only.

```
static const int
max_exponent10 ;
```

The maximum positive integer such that 10 raised to that power is in range. This field is meaningful for floating point types only.

```
static T
min () throw();
```

Returns the minimum finite value. For floating point types with denormalization, `min()`must return the minimum normalized value. The minimum denormalized value is given by `denorm_min()`. This function is meaningful for all specializations that declare `is_bounded` to be `true`, or `is_bounded == false && is_signed == false`.

```
static const int
min_exponent ;
```

The minimum negative integer such that the radix raised to the power one less than that integer is in range. This field is meaningful for floating point types only.

```
static const int
min_exponent10 ;
```

The minimum negative integer such that 10 raised to that power is in range. This field is meaningful for floating point types only.

©Copyright 1999 Rogue Wave Software, Inc.

```
static T
quiet_NaN () throw();
```

Returns the representation of a quiet "Not a Number", if available. This function is meaningful only for those specializations that declare `has_quiet_NaN` to be `true`. This field is required for any type claiming conformance to IEC 559.

```
static const int
radix ;
```

For floating types, specifies the base or radix of the exponent representation (often 2). For integer types, this member must specify the base of the representation. This field is meaningful for all specializations.

```
static T
round_error () throw();
```

Returns the measure of the maximum rounding error. This function is meaningful for floating point types only.

```
static const float_round_style
round_style ;
```

The rounding style for the type. Specializations for integer types must return `round_toward_zero`. This is meaningful for all floating point types.

```
static T
signaling_NaN() throw();
```

Returns the representation of a signaling "Not a Number", if available. This function is meaningful for only those specializations that declare `has_signaling_NaN` to be `true`. This function must be meaningful for any type claiming conformance to IEC 559.

```
static const bool
tinyness_before ;
```

This member is `true` if `tinyness` is detected before rounding. It is meaningful for floating point types only.

```
static const bool
traps ;
```

This field is `true` if trapping is implemented for this type. The `traps` field is meaningful for all specializations.

Example

```
//
// limits.cpp
//
 #include <limits>
 #include <iostream>
 using namespace std;


 int main()
 {
    numeric_limits<float> float_info;
    if (float_info.is_specialized &&
        float_info.has_infinity)
    {
      // get value of infinity
      cout<< float_info.infinity() << endl;
    }
    return 0;
 }
```

Warnings

The specializations for `wide chars` and `bool` are only available if your compiler has implemented them as real types and not simulated them with typedefs.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

See Also

IEEE Standard for Binary Floating-Point Arithmetic, 345 East 47th Street, New York, NY 10017

Language Independent Arithmetic (LIA-1)

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

numpunct, numpunct_byname

Summary

A numeric punctuation facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
string_type

Member Functions

decimal_point()
do_decimal_point()
do_falsename()
do_grouping()
do_thousands_sep()
do_truename()
falsename()
grouping()
thousands_sep()
truename()

Synopsis

```
#include <locale>
template <class charT>  class numpunct;
template <class charT>  class numpunct_byname;
```

Description

The ***numpunct<charT>*** facet specifies numeric punctuation. ***numpunct*** is used with the "C" locale, while the ***numpunct_byname*** facet is used with named locales.

Both ***num_put*** and ***num_get*** make use of this facet.

Interface

```
template <class charT>
class numpunct : public locale::facet {
public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef charT              char_type;
  typedef basic_string<charT> string_type;
  explicit numpunct(size_t refs = 0);
  char_type    decimal_point()   const;
  char_type    thousands_sep()   const;
  string       grouping()        const;
  string_type  truename()        const;
  string_type  falsename()       const;
  static locale::id id;
protected:
  ~numpunct();  // virtual
  virtual char_type    do_decimal_point() const;
  virtual char_type    do_thousands_sep() const;
  virtual string       do_grouping()      const;
  virtual string_type  do_truename()     const;  // for bool
  virtual string_type  do_falsename()    const;  // for bool
};

template <class charT>
class numpunct_byname : public numpunct<charT> {
public:
  explicit numpunct_byname(const char*, size_t refs = 0);
protected:
  ~numpunct_byname();  // virtual
  virtual char_type    do_decimal_point() const;
  virtual char_type    do_thousands_sep() const;
  virtual string       do_grouping()      const;
  virtual string_type  do_truename()     const;  // for bool
  virtual string_type  do_falsename()    const;  // for bool
};
```

Types

`char_type`

Type of character upon which the facet is instantiated.

`string_type`

Type of character string returned by member functions.

Constructors

```
explicit numpunct(size_t refs = 0)
```

Constructs a ***numpunct*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

```
explicit numpunct_byname(const char* name,
         size_t refs = 0);
```

Constructs a ***numpunct_byname*** facet. Uses the named locale specified by the name argument. The `refs` argument serves the same purpose as it does for the ***numpunct*** constructor.

Destructors

```
~numpunct();  // virtual and protected
~numpunct_byname();  // virtual and protected
```

Destroys the facet.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

©Copyright 1999 Rogue Wave Software, Inc.

Public Member Functions

The public members of the ***numpunct*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `grouping` function simply calls its protected cousin `do_grouping`.

```
char_type    decimal_point()   const;
string_type  falsename()       const;
string       grouping()        const;
char_type    thousands_sep()   const;
string_type  truename()        const;
```

Each of these public member functions `xxx` simply call the corresponding protected `do_xxx` function.

Protected Member Functions

```
virtual char_type
do_decimal_point() const;
```

Returns the decimal radix separator. ***numpunct*** returns `.'.

```
virtual string_type
do_falsename()     const;  // for bool
virtual string_type
do_truename()      const;  // for bool
```

Returns a string containing `true` or `false`.

```
virtual string
do_grouping()      const;
```

Returns a string in which each character is used as an integer value to represent the number of digits in a particular grouping, starting with the rightmost group. A group is simply the digits between adjacent thousands separators. Each group at a position larger than the size of the string gets the same value as the last element in the string. If a value is less than or equal to zero, or equal to `CHAR_MAX`, then the size of that group is unlimited. ***numpunct*** returns an empty string, indicating no grouping.

```
virtual char_type
do_thousands_sep() const;
```

Returns the decimal digit group separator. ***numpunct*** returns `,'.

Example

```
//
// numpunct.cpp
//

#include <iostream>

int main ()
{
  using namespace std;
  locale loc;

  // Get a numpunct facet
  const numpunct<char>& np =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<numpunct<char> >(loc);
#else
  use_facet(loc,(numpunct<char>*)0);
#endif

  cout << "Decimal point       = "
       << np.decimal_point() << endl;
  cout << "Thousands separator = "
       << np.thousands_sep() << endl;
  cout << "True name           = "
       << np.truename() << endl;
  cout << "False name          = "
       << np.falsename() << endl;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  return 0;
}
```

See Also

locale, *facets*, *num_put*, *num_get*, *ctype*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

Operators

`Utility Operators`

Summary

Operators for the C++ Standard Template Library.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <utility>
namespace rel_ops {
template <class T>
 bool operator!= (const T&, const T&);

template <class T>
 bool operator> (const T&, const T&);

template <class T>
 bool operator<= (const T&, const T&);

template <class T>
 bool operator>= (const T&, const T&);
}
```

Description

To avoid redundant definitions of `operator!=` out of `operator==` and of `operators >`, `<=`, and `>=` out of `operator<`, the library includes these definitions:

`operator!=(x,y)` returns `!(x==y)`

`operator>(x,y)` returns `y<x`

`operator<=(x,y)` returns `!(y<x)`

`operator>=(x,y)` returns `!(x<y)`

To avoid clashes with other global operators, these definitions are contained in the namespace `rel_ops`. To use them, either scope explicitly or include a "using" declaration (for example, `using namespace rel_ops`).

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ostream_iterator

`Iterator`

Summary

Stream iterators allow for use of iterators with ostreams and istreams. They allow generic algorithms to be used directly on streams.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
ostream_type
traits_type
value_type

Member Functions

operator*()
operator++()
operator=()

Synopsis

```
#include <ostream>
template <class T, class charT,
          class traits = char_traits<charT> >
class ostream_iterator
 : public iterator<output_iterator_tag,void,void>;
```

Description

Stream iterators use the standard iterator interface for input and output streams.

The class ***ostream_iterator*** writes elements to an output stream. If you use the constructor that has a second `char *` argument, then that string is written after every element (the string must be null-terminated). Since an ostream iterator is an output iterator, it is not possible to get an element out of the iterator. You can only assign to it.

Interface

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T, class charT,
          class traits = char_traits<charT> >
 class ostream_iterator
   : public iterator<output_iterator_tag,void,void>
{
 public:



  typedef T value_type;
  typedef charT char_type;
  typedef traits traits_type;
  typedef basic_ostream<charT,traits> ostream_type;

    ostream_iterator(ostream&);
    ostream_iterator (ostream&, const char*);
    ostream_iterator (const
            ostream_iterator<T,charT,char_traits<charT> >&);
    ~ostream_itertor ();
    ostream_iterator<T,charT,char_traits<charT> >&
         operator=(const T&);
    ostream_iterator<T,charT,char_traits<charT> >&
         operator* () const;
    ostream_iterator<T,charT,char_traits<charT> >&
         operator++ ();
    ostream_iterator<T,charT,char_traits<charT> >
         operator++ (int);
 };
```

Types

`value_type;`

Type of value to stream in.

`char_type;`

Type of character the stream is built on.

`traits_type;`

Traits used to build the stream.

`ostream_type;`

Type of stream this iterator is constructed on.

Constructors

`ostream_iterator` `(ostream& s);`

Constructs an ***ostream_iterator*** on the given stream.

`ostream_iterator` `(ostream& s, const char* delimiter);`

Constructs an ***ostream_iterator*** on the given stream. The null terminated string `delimiter` is written to the stream after every element.

`ostream_iterator` `(const ostream_iterator<T>& x);`

Copy constructor.

Destructors

`~ostream_iterator` `();`

Destroys an object of class `ostream_iterator`.

Operators

©Copyright 1999 Rogue Wave Software, Inc.

```
const T&
operator= (const T& value);
```

Shift the value `T` onto the output stream.

```
const T& ostream_iterator<T>&
operator* ();
ostream_iterator<T>&
operator++();
ostream_iterator<T>
operator++ (int);
```

These operators do nothing. They simply allow the iterator to be used in common constructs.

Example

```
#include <iterator>
#include <numeric>
#include <deque>
#include <iostream>
using namespace std;

int main ()
{
  //
  // Initialize a vector using an array.
  //
  int arr[4] = { 3,4,7,8 };
  int total=0;
  deque<int> d(arr+0, arr+4);
  //
  // stream the whole vector and a sum to cout
  //
  copy(d.begin(),d.end()-1,
       ostream_iterator<int,char>(cout," + "));
  cout << *(d.end()-1) << " = " <<
       accumulate(d.begin(),d.end(),total) << endl;
  return 0;
}
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

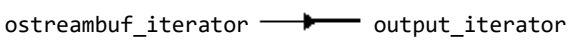
istream_iterator, *Iterators*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ostreambuf_iterator



Summary

Writes successive characters onto the stream buffer object from which it was constructed.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
ostream_type
streambuf_type
traits_type

Member Functions

failed()
operator*()
operator++()
operator=()

Synopsis

```
#include <streambuf>
template<class charT, class traits = char_traits<charT> >
class ostreambuf_iterator
: public output_iterator
```

Description

The template class ***ostreambuf_iterator*** writes successive characters onto the stream buffer object from which it was constructed. `operator=` is used to write the characters. In case of failure, the member function `failed()` returns `true`.

Interface

```
template<class charT, class traits = char_traits<charT> >
class ostreambuf_iterator
: public output_iterator {

 public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef charT                          char_type;
  typedef traits                         traits_type;
  typedef basic_streambuf<charT, traits> streambuf_type;
  typedef basic_ostream<charT, traits>   ostream_type;

  ostreambuf_iterator(ostream_type& s) throw();
  ostreambuf_iterator(streambuf_type *s) throw();
  ostreambuf_iterator& operator=(charT c);

  ostreambuf_iterator& operator*();
  ostreambuf_iterator& operator++();
  ostreambuf_iterator operator++(int);

  bool failed( ) const throw();

};
```

Types

char_type

The type `char_type` is a synonym for the template parameter `charT`.

ostream_type

The type `ostream_type` is an instantiation of class `basic_ostream` on types `charT` and `traits`:

```
typedef basic_ostream<charT, traits> ostream_type;
```

streambuf_type

The type `streambuf_type` is an instantiation of class `basic_streambuf` on types `charT` and `traits`:

```
typedef basic_streambuf<charT, traits> streambuf_type;
```

traits_type

The type `traits_type` is a synonym for the template parameter `traits`.

Constructors

ostreambuf_iterator`(ostream_type& s) throw();`

Constructs an `ostreambuf_iterator` that uses the `basic_streambuf` object pointed to by `s.rdbuf()`to output characters. If `s.rdbuf()` is a null pointer, calls to the member function `failed()` return `true`.

ostreambuf_iterator`(streambuf_type *s) throw();`

Constructs an `ostreambuf_iterator` that uses the `basic_streambuf` object pointed to by `s` to output characters. If `s` is a null pointer, calls the member function `failed()` return `true`.

Member Operators

`ostreambuf_iterator&`
operator=`(charT c);`

Inserts the `character c` into the output sequence of the attached stream buffer. If the operation fails, calls to the member function `failed()` return `true`.

`ostreambuf_iterator&`
operator++`();`

Returns `*this`.

`ostreambuf_iterator`
operator++`(int);`

Returns `*this`.

©Copyright 1999 Rogue Wave Software, Inc.

```
ostreambuf_iterator
operator*();
```

Returns `*this`.

Public Member Functions

```
bool
failed() const
  throw();
```

Returns `true` if the iterator failed while inserting a character. Otherwise returns `false`.

Example

```
//
// stdlib/examples/manual/ostreambuf_iterator.cpp
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // create a filebuf object
  filebuf  buf;

  // open the file iter_out and link it
  // to the filebuf object
  buf.open("iter_out", ios_base::in | ios_base::out );

  // create an ostreambuf_iterator and link it to
  // the filebuf object
  ostreambuf_iterator<char> out_iter(&buf);

  // output into the file using the ostreambuf_iterator
  for(char i=64; i<128; i++ )
   out_iter = i;

  // seek to the beginning of the file
  buf.pubseekpos(0);

  // create an istreambuf_iterator and link it to
  // the filebuf object
  istreambuf_iterator<char> in_iter(&buf);

  // construct an end of stream iterator
  istreambuf_iterator<char> end_of_stream_iterator;

  cout << endl;

  // output the content of the file
  while( !in_iter.equal(end_of_stream_iterator) )

  // use both operator++ and operator*
  cout << *in_iter++;

  cout << endl;

}
```

See Also

basic_streambuf(3C++), ***basic_ostream***(3C++), ***istreambuf_iterator***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 24.5.4

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

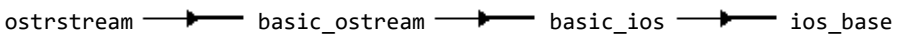
©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ostrstream



Summary

Writes to an array in memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
off_type
pos_type

traits

Member Functions

freeze()
pcount()
rdbuf()
str()

Synopsis

```
#include <strstream>
class ostrstream
: public basic_ostream<char>
```

Description

The class ***ostrstream*** writes to an array in memory. It uses a private ***strstreambuf*** object to control the associated array object. It inherits from ***basic_ostream<char>*** and therefore can use all the formatted and unformatted output functions.

This is a deprecated feature and might not be available in future versions.

Interface

```
class ostrstream
: public basic_ostream<char> {

 public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  typedef char_traits<char>             traits;

  typedef char                          char_type;
  typedef typename traits::int_type     int_type;
  typedef typename traits::pos_type     pos_type;
  typedef typename traits::off_type     off_type;

  ostrstream();
  ostrstream(char *s, int n,
             ios_base::openmode = ios_base::out);

  virtual ~ostrstream();

  strstreambuf *rdbuf() const;
  void freeze(int freezefl = 1);
  char *str();
  int pcount() const;

};
```

Types

char_type

The type char_type is a synonym of type char.

int_type

The type int_type is a synonym of type traits::in_type.

off_type

The type off_type is a synonym of type traits::off_type.

pos_type

The type pos_type is a synonym of type traits::pos_type.

traits

The type traits is a synonym of type char_traits<char>.

Constructors

ostrstream();

Constructs an object of class ostrstream, initializing the base class basic_ostream<char> with the associated strstreambuf object. The strstreambuf object is initialized by calling its default constructor strstreambuf().

```
ostrstream(char* s, int n, ios_base::openmode
           mode = ios_base::out);
```

Constructs an object of class ostrstream, initializing the base class basic_ostream<char> with the associated strstreambuf object. The strstreambuf object is initialized by calling one of two constructors:

- If mode & app == 0, it calls strstreambuf(s,n,s)
- Otherwise it calls strstreambuf(s,n,s + ::strlen(s))

Destructors

virtual **~ostrstream**();

Destroys an object of class ostrstream.

Member Functions

```
void
freeze(bool freezefl = 1);
```

©Copyright 1999 Rogue Wave Software, Inc.

If the mode is dynamic, alters the freeze status of the dynamic array object as follows:

- If `freezefl` is `false`, the function sets the freeze status to frozen.
- Otherwise, it clears the freeze status.

```
int
pcount() const;
```

Returns the size of the output sequence.

```
strstreambuf*
rdbuf() const;
```

Returns a pointer to the private `strstreambuf` object associated with the stream.

```
char*
str();
```

Returns a pointer to the underlying array object, which may be null.

Example

See ***strstream***, ***istrstream*** and ***strstreambuf*** examples.

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***strstreambuf***(3C++), ***istrstream***(3C++), ***strstream***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Annex D Compatibility features Section D.6.3

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Output Iterators

`Iterator`

Summary

A write-only, forward moving iterator.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

NOTE:For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Output iterators are write-only, forward moving iterators that satisfy the requirements listed below. Note that unlike other iterators used with the standard library, output iterators cannot be constant.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a and b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Output Iterators

The following expressions must be valid for output iterators:

`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`*a = t`	result is not used
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++ = t`	result is not used

The only valid use for the `operator *` is on the left-hand side of the assignment statement.

Algorithms using output iterators should be single pass algorithms. That is, they should not pass through the same iterator twice.

©Copyright 1999 Rogue Wave Software, Inc.

See Also

Iterators, *Input Iterators*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

pair

`Utility Class`

Summary

A template for heterogeneous pairs of values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

first_type
second_type

Member Functions

make_pair()
operator!=()
operator>()
operator>=()
operator<()
operator<=()
operator==()

Synopsis

```
#include <utility>
template <class T1, class T2>
struct pair ;
```

Description

The ***pair*** class is a template for encapsulating pairs of values that may be of different types.

Interface

```
template <class T1, class T2>
 struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;
        pair();
        pair (const T1&, const T2&);
        template <class V, class U>
        pair (const pair <V, U>& p);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        ~pair();
};

template <class T1, class T2>
 bool operator== (const pair<T1, T2>&,
                  const pair T1, T2>&);

template <class T1, class T2>
 bool operator!= (const pair<T1, T2>&,
                  const pair T1, T2>&);

template <class T1, class T2>
 bool operator< (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator> (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator<= (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator>= (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 pair<T1,T2> make_pair (const T1&, const T2&);
```

Types

first_type

Type of the first element in a pair.

second_type

Type of the second element in a pair.

Constructors

pair ();

Default constructor. Initializes `first` and `second` using their default constructors.

pair (const T1& x, const T2& y);

Creates a pair of types `T1` and `T2`, making the necessary conversions in `x` and `y`.

```
template <class V, class U>
pair (const pair <V, U>& p);
```

Copies `first` and `second` from the corresponding elements of `p`.

Destructors

~pair ();

Non-member Operators

```
template <class T1, class T2>
 bool operator== (const pair<T1, T2>& x,
                  const pair T1, T2>& y);
```

Returns `true` if (`x.first == y.first && x.second == y.second`) is `true`. Otherwise it returns `false`.

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T1, class T2>
 bool operator!= (const pair<T1, T2>& x,
                  const pair T1, T2>& y);
```

Returns `!(x==y)`.

```
template <class T1, class T2>
bool operator< (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `true` if `(x.first < y.first || (!(y.first < x.first) && x.second < y.second))` is `true`. Otherwise it returns `false`.

```
template <class T1, class T2>
bool operator> (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `y < x`.

```
template <class T1, class T2>
bool operator<= (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `!(y < x)`.

```
template <class T1, class T2>
bool operator>= (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `!(x < y)`.

Non-member Functions

```
template <class T1, class T2>
pair<T1,T2>
make_pair(x,y);
```

`make_pair(x,y)` creates a pair by deducing and returning the types of `x` and `y`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

partial_sort

`Algorithm`

Summary

Templatized algorithm for sorting collections of entities.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
 void partial_sort (RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
 void partial_sort (RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
```

Description

The ***partial_sort*** algorithm takes the range `[first,last)` and places the first `middle - first` values into sorted order. The result is that the range `[first, middle)` is sorted like it would be if the entire range `[first,last)` were sorted. The remaining elements in the range (those in `[middle, last)`) are not in any defined order. The first version of the algorithm uses less than (`operator<`) as the comparison operator for the sort. The second version uses the comparison function `comp`.

Complexity

partial_sort does approximately `(last - first) * log(middle-first)` comparisons.

Example

```
//
// partsort.cpp
//
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
int main()
 {
   int d1[20] = {17, 3,  5,  -4, 1, 12, -10, -1, 14, 7,
                 -6, 8, 15, -11, 2, -2,  18,  4, -3, 0};
   //
   // Set up a vector.
   //
   vector<int> v1(d1+0, d1+20);
   //
   // Output original vector.
   //
   cout << "For the vector: ";
   copy(v1.begin(), v1.end(),
        ostream_iterator<int,char>(cout," "));
   //
   // Partial sort the first seven elements.
   //
   partial_sort(v1.begin(), v1.begin()+7, v1.end());
   //
   // Output result.
   //
   cout << endl << endl << "A partial_sort of seven elements
                           gives: "
        << endl << "     ";
   copy(v1.begin(), v1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   //
   // A vector of ten elements.
   //
   vector<int> v2(10, 0);
   //
   // Sort the last ten elements in v1 into v2.
   //
   partial_sort_copy(v1.begin()+10, v1.end(), v2.begin(),
                     v2.end());
   //
   // Output result.
   //
   cout << endl << "A partial_sort_copy of the last ten
         elements gives: "
        << endl << "     ";
   copy(v2.begin(), v2.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
For the vector: 17 3 5 -4 1 12 -10 -1 14 7 -6 8 15 -11 2 -2 18 4 -3 0
A partial_sort of seven elements gives:
     -11 -10 -6 -4 -3 -2 -1 17 14 12 7 8 15 5 3 2 18 4 1 0
A partial_sort_copy of the last ten elements gives:
     0 1 2 3 4 5 7 8 15 18
```

Warnings

If your compiler does not support default template parameters, then you always need to include the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

sort, *stable_sort*, *partial_sort_copy*

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

partial_sort_copy

`Algorithm`

Summary

Templatized algorithm for sorting collections of entities.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class RandomAccessIterator>
 void partial_sort_copy (InputIterator first,
                         InputIterator last,
                         RandomAccessIterator result_first,
                         RandomAccessIterator result_last);
template <class InputIterator,
          class RandomAccessIterator,
          class Compare>
 void partial_sort_copy (InputIterator first,
                         InputIterator last,
                         RandomAccessIterator result_first,
                         RandomAccessIterator result_last,
                         Compare comp);
```

Description

The ***partial_sort_copy*** algorithm places the smaller of `last - first` and `result_last - result_first` sorted elements from the range `[first, last)` into the range beginning at `result_first` (in other words, the range: `[result_first, result_first+min(last - first, result_last - result_first))`. The effect is as if the range `[first,last)` were placed in a temporary buffer, sorted, and then as many elements as possible copied into the range `[result_first, result_last)`.

The first version of the algorithm uses less than (`operator<`) as the comparison operator for the sort. The second version uses the comparison function `comp`.

Complexity

partial_sort_copy does approximately `(last-first) * log(min(last-first, result_last-result_first))` comparisons.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// partsort.cpp
 #include <vector>
 #include <algorithm>
 #include <iostream>
 using namespace std;
 int main()
 {
   int d1[20] = {17, 3,  5,  -4, 1, 12, -10, -1, 14, 7,
                 -6, 8, 15, -11, 2, -2,  18,  4, -3, 0};
   //
   // Set up a vector.
   //
   vector<int> v1(d1+0, d1+20);
   //
   // Output original vector.
   //
   cout << "For the vector: ";
   copy(v1.begin(), v1.end(),
        ostream_iterator<int>(cout," "));
   //
   // Partial sort the first seven elements.
   //
   partial_sort(v1.begin(), v1.begin()+7, v1.end());
   //
   // Output result.
   //
   cout << endl << endl << "A partial_sort of 7
        elements gives: "
        << endl << "     ";
   copy(v1.begin(), v1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   //
   // A vector of ten elements.
   vector<int> v2(10, 0);
   //
   // Sort the last ten elements in v1 into v2.
   partial_sort_copy(v1.begin()+10, v1.end(), v2.begin(),
                     v2.end());
   //
   // Output result.
   cout << endl << "A partial_sort_copy of the last
         ten elements gives: " << endl << "     ";
   copy(v2.begin(), v2.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   return 0;
 }
```

Program Output

```
For the vector: 17 3 5 -4 1 12 -10 -1 14 7 -6 8 15 -11 2 -2 18 4 -3 0
A partial_sort of seven elements gives:
     -11 -10 -6 -4 -3 -2 -1 17 14 12 7 8 15 5 3 2 18 4 1 0
A partial_sort_copy of the last ten elements gives:
     0 1 2 3 4 5 7 8 15 18
```

Warnings

If your compiler does not support default template parameters, then you need to always include the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

©Copyright 1999 Rogue Wave Software, Inc.

sort* *stable_sort*, *partial_sort

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

partial_sum

`Generalized Numeric Operation`

Summary

Calculates successive partial sums of a range of values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <numeric>
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum (InputIterator first,
                            InputIterator last,
                            OutputIterator result);

template <class InputIterator,
          class OutputIterator,
          class BinaryOperation>
OutputIterator partial_sum (InputIterator first,
                            InputIterator last,
                            OutputIterator result,
                            BinaryOperation binary_op);
```

Description

The ***partial_sum*** algorithm creates a new sequence in which every element is formed by adding all the values of the previous elements, or, in the second form of the algorithm, by applying the operation `binary_op` successively on every previous element. That is, ***partial_sum*** assigns to every iterator `i` in the range `[result, result + (last - first))` a value equal to:

```
((...(*first + *(first + 1)) + ... ) +
      *(first + (i - result)))
```

or, in the second version of the algorithm:

```
binary_op(binary_op(..., binary_op (*first,
          *(first + 1)),...),*(first + (i - result)))
```

For instance, applying ***partial_sum*** to (1,2,3,4,) yields (1,3,6,10).

The ***partial_sum*** algorithm returns `result + (last - first)`.

If `result` is equal to `first`, the elements of the new sequence successively replace the elements in the original sequence, effectively turning ***partial_sum*** into an inplace transformation.

©Copyright 1999 Rogue Wave Software, Inc.

Complexity

Exactly `(last - first) - 1` applications of the default `+` operator or `binary_op` are performed.

Example

```
//
// partsum.cpp
//
 #include <numeric>   //for accumulate
 #include <vector>    //for vector
 #include <functional> //for times
 #include <iostream>
 using namespace std;

 int main()
 {
   //Initialize a vector using an array of ints
   int d1[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(d1, d1+10);

   //Create an empty vectors to store results
   vector<int> sums((size_t)10), prods((size_t)10);

   //Compute partial_sums and partial_products
   partial_sum(v.begin(), v.end(), sums.begin());
   partial_sum(v.begin(), v.end(), prods.begin(),
               times<int>());
   //Output the results
   cout << "For the series: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   cout << "The partial sums: " << endl << "     " ;
   copy(sums.begin(),sums.end(),
        ostream_iterator<int,char>(cout," "));
   cout <<" should each equal (N*N + N)/2" << endl << endl;

   cout << "The partial products: " << endl << "     ";
   copy(prods.begin(),prods.end(),
        ostream_iterator<int,char>(cout," "));
   cout << " should each equal N!" << endl;

   return 0;
 }
```

Program Output

```
For the series:
  1 2 3 4 5 6 7 8 9 10
The partial sums:
  1 3 6 10 15 21 28 36 45 55  should each equal (N*N + N)/2
The partial products:
  1 2 6 24 120 720 5040 40320 362880 3628800  should each equal N!
```

Warnings

If your compiler does not support default template parameters, then you always need to include the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

partition

`Algorithm`

Summary

Places all of the entities that satisfy the given predicate before all of the entities that do not.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator
partition (BidirectionalIterator first,
           BidirectionalIterator last,
           Predicate pred);
```

Description

For the range `[first, last)`, the ***partition*** algorithm places all elements that satisfy `pred` before all elements that do not satisfy `pred`. It returns an iterator that is one past the end of the group of elements that satisfy `pred`. In other words, ***partition*** returns `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and, for any iterator `k` in the range `[i, last)`, `pred(*j) == false`.

Note that ***partition*** does not necessarily maintain the relative order of the elements that match and elements that do not match the predicate. Use the algorithm ***stable_partition*** if relative order is important.

Complexity

The ***partition*** algorithm does at most `(last - first)/2` swaps, and applies the predicate exactly `last - first` times.

Example

```
//
// prtition.cpp
//
 #include <functional>
 #include <deque>
 #include <algorithm>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// Create a new predicate from unary_function.
//
template<class Arg>
class is_even : public unary_function<Arg, bool>
{
  public:
  bool operator()(const Arg& arg1) { return (arg1 % 2)
                == 0; }
};

int main ()
{
  //
  // Initialize a deque with an array of integers.
  //
  int init[10] = { 1,2,3,4,5,6,7,8,9,10 };
  deque<int> d1(init+0, init+10);
  deque<int> d2(init+0, init+10);
  //
  // Print out the original values.
  //
  cout << "Unpartitioned values: " << "\t\t";
  copy(d1.begin(), d1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl;
  //
  // A partition of the deque according to even/oddness.
  //
  partition(d2.begin(), d2.end(), is_even<int>());
  //
  // Output result of partition.
  //
  cout << "Partitioned values: " << "\t\t";
  copy(d2.begin(), d2.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl;
  //
  // A stable partition of the deque according
  // to even/oddness.
  //
  stable_partition(d1.begin(), d1.end(), is_even<int>());
  //
  // Output result of partition.
  //
  cout << "Stable partitioned values: " << "\t";
  copy(d1.begin(), d1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl;

  return 0;
}
```

Program Output

```
Unpartitioned values:           1 2 3 4 5 6 7 8 9 10
Partitioned values:             10 2 8 4 6 5 7 3 9 1
Stable partitioned values:      2 4 6 8 10 1 3 5 7 9
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

©Copyright 1999 Rogue Wave Software, Inc.

stable_partition

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

permutation

`Algorithm`

Summary

Generates successive permutations of a sequence based on an ordering function. See the entries for ***next_permutation*** and ***prev-_permutation***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

plus

`Function Object`

Summary

A binary function object that returns the result of adding its first and second arguments.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template<class T>
struct plus : public binary_function<T, T, T> ;
```

Description

plus is a binary function object. Its `operator()` returns the result of adding `x` and `y`. You can pass a ***plus*** object to any algorithm that uses a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***plus*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(), vec2.begin(),
          vecResult.begin(), plus<int>());
```

After this call to ***transform***, `vecResult(n)` contains `vec1(n)` plus `vec2(n)`.

Interface

```
template<class T>
struct plus : binary_function<T, T, T> {
  T operator() (const T&, const T&) const;
};
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

binary_function, ***Function Objects***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

pointer_to_binary_function

`Function Object`

Summary

A function object that adapts a pointer to a binary function, to take the place of a ***binary_function***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public
      binary_function<Arg1, Arg2, Result> ;
```

Description

The ***pointer_to_binary_function*** class encapsulates a pointer to a two-argument function. The class uses `operator()` so that the resulting object serves as a binary function object for that function.

The `ptr_fun` function is overloaded to create instances of a ***pointer_to_binary_function*** when included with the appropriate pointer to a function.

Interface

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public
      binary_function<Arg1, Arg2, Result> {
 public:
   explicit pointer_to_binary_function
            (Result (*f)(Arg1, Arg2));
   Result operator() (const Arg1&, const Arg2&) const;
};

template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
             ptr_fun (Result (*x)(Arg1, Arg2));
```

See Also

binary_function, ***Function Objects***, ***pointer_to_unary_function***, ***ptr_fun***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

pointer_to_unary_function

`Function Object`

Summary

A function object class that adapts a pointer to a function, to take the place of a ***unary_function***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result>;
```

Description

The ***pointer_to_unary_function*** class encapsulates a pointer to a single-argument function. The class uses `operator()` so that the resulting object serves as a function object for that function.

The `ptr_fun` function is overloaded to create instances of ***pointer_to_unary_function*** when included with the appropriate pointer to a function.

Interface

```
template <class Arg, class Result>
class pointer_to_unary_function : public
      unary_function<Arg, Result> {

 public:
   explicit pointer_to_unary_function (Result (*f)(Arg));
   Result operator() (const Arg&) const;
};

template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
  ptr_fun (Result (*f)(Arg));
```

See Also

Function Objects, ***pointer_to_binary_function***, ***ptr_fun***, ***unary_function***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

pop_heap

`Algorithms`

Summary

Moves the largest element off the heap.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
template <class RandomAccessIterator>
  void
  pop_heap(RandomAccessIterator first,
           RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  pop_heap(RandomAccessIterator first,
           RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** algorithm or a new element may be added by the ***push_heap*** algorithm, in `O(logN)` time.

These properties make heaps useful as priority queues.

The ***pop_heap*** algorithm uses the less than (`<`) operator as the default comparison. An alternate comparison operator can be specified.

The ***pop_heap*** algorithm can be used as part of an operation to remove the largest element from a heap. It assumes that the range `[first, last)` is a valid heap (in other words, that `first` is the largest element in the heap or the first element based on the alternate comparison operator). It then swaps the value in the location `first` with the value in the location `last - 1` and makes the range `[first, last -1)`back into a heap. You can then access the element in `last` using the vector or deque `back()` member function, or you can remove the element using the `pop_back` member function. Note that ***pop_heap*** does not actually remove the element from the data structure; you must use another function to do that.

©Copyright 1999 Rogue Wave Software, Inc.

Complexity

pop_heap performs at most `2 * log(last - first)` comparisons.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

   // Make heaps
   make_heap(v1.begin(),v1.end());
   make_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
   // Note that x, y and z represent the remaining
   // values in the container (other than 4).
   // The definition of the heap and heap operations
   // does not require any particular ordering
   // of these values.

   // Copy both vectors to cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now let's pop
   pop_heap(v1.begin(),v1.end());
   pop_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (3,x,y,4) and v2 = (3,x,y,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // And push
   push_heap(v1.begin(),v1.end());
   push_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z) and v2 = (4,x,y,z)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now sort those heaps
   sort_heap(v1.begin(),v1.end());
   sort_heap(v2.begin(),v2.end(),less<int>());
   // v1 = v2 = (1,2,3,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   return 0;
 }
```

©Copyright 1999 Rogue Wave Software, Inc.

Program Output

```
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

make_heap, ***push_heap***, ***sort_heap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Predicates

Summary

A function or a function object that returns a boolean (`true`/`false`) value or an integer value.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

prev_permutation

`Algorithm`

Summary

Generates successive permutations of a sequence based on an ordering function.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator>
bool prev_permutation (BidirectionalIterator first,
                       BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool prev_permutation (BidirectionalIterator first,
                       BidirectionalIterator last,
                       Compare comp);
```

Description

The permutation-generating algorithms (***next_permutation*** and ***prev_permutation***) assume that the set of all permutations of the elements in a sequence is lexicographically sorted with respect to `operator<` or `comp`. For example, if a sequence includes the integers 1 2 3, that sequence has six permutations. In order from first to last, they are: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, and 3 2 1.

The ***prev_permutation*** algorithm takes a sequence defined by the range `[first, last)` and transforms it into its previous permutation, if possible. If such a permutation does exist, the algorithm completes the transformation and returns `true`. If the permutation does not exist, ***prev_permutation*** returns `false`, and transforms the permutation into its "last" permutation. ***prev_permutation*** does the transformation according to the lexicographical ordering defined by either `operator<` (the default used in the first version of the algorithm) or `comp` (which is user-supplied in the second version of the algorithm).

For example, if the sequence defined by `[first, last)` contains the integers 1 2 3 (in that order), there is *not* a "previous permutation." Therefore, the algorithm transforms the sequence into its last permutation (3 2 1) and returns `false`.

Complexity

At most `(last - first)/2` swaps are performed.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// permute.cpp
//
 #include <numeric>    //for accumulate
 #include <vector>        //for vector
 #include <functional> //for less
 #include <iostream>
 using namespace std;

 int main()
 {
   //Initialize a vector using an array of ints
   int  a1[] = {0,0,0,0,1,0,0,0,0,0};
   char a2[] = "abcdefghji";

   //Create the initial set and copies for permuting
   vector<int>  m1(a1, a1+10);
   vector<int>  prev_m1((size_t)10), next_m1((size_t)10);
   vector<char> m2(a2, a2+10);
   vector<char> prev_m2((size_t)10), next_m2((size_t)10);

   copy(m1.begin(), m1.end(), prev_m1.begin());
   copy(m1.begin(), m1.end(), next_m1.begin());
   copy(m2.begin(), m2.end(), prev_m2.begin());
   copy(m2.begin(), m2.end(), next_m2.begin());

   //Create permutations
   prev_permutation(prev_m1.begin(),
                    prev_m1.end(),less<int>());
   next_permutation(next_m1.begin(),
                    next_m1.end(),less<int>());
   prev_permutation(prev_m2.begin(),
                    prev_m2.end(),less<int>());
   next_permutation(next_m2.begin(),
                    next_m2.end(),less<int>());
   //Output results
   cout << "Example 1: " << endl << "     ";
   cout << "Original values:      ";
   copy(m1.begin(),m1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << "     ";
   cout << "Previous permutation: ";
   copy(prev_m1.begin(),prev_m1.end(),
        ostream_iterator<int,char>(cout," "));

   cout << endl<< "     ";
   cout << "Next Permutation:     ";
   copy(next_m1.begin(),next_m1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   cout << "Example 2: " << endl << "     ";
   cout << "Original values: ";
   copy(m2.begin(),m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << "     ";
   cout << "Previous Permutation: ";
   copy(prev_m2.begin(),prev_m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << "     ";

   cout << "Next Permutation:     ";
   copy(next_m2.begin(),next_m2.end(),
        ostream_iterator<char,char>(cout," "));
   cout << endl << endl;

   return 0;
 }
```

Program Output

```
Example 1:
     Original values:      0 0 0 0 1 0 0 0 0 0
     Previous permutation: 0 0 0 0 0 1 0 0 0 0
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    Next Permutation:     0 0 0 1 0 0 0 0 0 0
Example 2:
    Original values: a b c d e f g h j i
    Previous Permutation: a b c d e f g h i j
    Next Permutation:     a b c d e f g i h j
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

next_permutation

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

priority_queue

`Container Adaptor`

Summary

A container adapter that behaves like a priority queue. Items popped from the queue are in order with respect to a "priority."

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

empty()
pop()
push()
size()
top()

Synopsis

```
#include <queue>
template <class T,
          class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue;
```

Description

priority_queue is a container adaptor that allows a container to act as a priority queue. This means that the item with the highest priority, as determined by either the default comparison operator (`operator <`) or the comparison `Compare`, is brought to the front of the queue whenever anything is pushed onto or popped off the queue.

priority_queue adapts any container that gives `front()`, `push_back()`, and `pop_back()`. In particular, ***deque*** and ***vector*** can be used.

Interface

```
template <class T, class Container = vector<T>,
          class Compare = less<typename
                          Container::value_type> >
 class priority_queue {
public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// typedefs
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef Container container_type;

//  Construct
   explicit priority_queue (const Compare& = Compare(),
                            const Container& = Container());
   template <class InputIterator>
     priority_queue (InputIterator first,
                     InputIterator last,
                     const Compare& = Compare(),
                     const Container& = Container());
   bool empty () const;
   size_type size () const;
   const value_type& top () const;
   void push (const value_type&);
   void pop();
};
```

Constructors

```
explicit priority_queue (const Compare& x = Compare(),
                         const Container& = Container());
```

Constructs a priority queue that uses `Container` for its underlying implementation, `x` as its standard for determining priority, and the allocator Allocator`()` for all storage management.

```
template <class InputIterator>
priority_queue (InputIterator first, InputIterator last,
                const Compare& x = Compare(),
                const allocator_type& alloc =
                allocator_type());
```

Constructs a new priority queue and places into it every entity in the range `[first, last)`. The priority_queue uses `x` for determining the priority, and the allocator `alloc` for all storage management.

Member Functions

```
bool
empty () const;
```

Returns `true` if the priority_queue is empty, `false` otherwise.

```
void
pop();
```

Removes the item with the highest priority from the queue.

```
void
push (const value_type& x);
```

Adds `x` to the queue.

```
size_type
size () const;
```

Returns the number of elements in the priority_queue.

```
const value_type&
top () const;
```

Returns a constant reference to the element in the queue with the highest priority.

Example

```
//
// p_queue.cpp
//
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <queue>
#include <deque>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

int main(void)
{
  // Make a priority queue of int using a vector container
  priority_queue<int, vector<int>, less<int> > pq;

  // Push a couple of values
  pq.push(1);
  pq.push(2);

  // Pop a couple of values and examine the ends
  cout << pq.top() << endl;
  pq.pop();
  cout << pq.top() << endl;
  pq.pop();

  // Make a priority queue of strings using
  // a deque container
  priority_queue<string, deque<string>, less<string> >
     pqs;

  // Push on a few strings then pop them back off
  int i;
  for (i = 0; i < 10; i++)
  {
    pqs.push(string(i+1,'a'));
    cout << pqs.top() << endl;
  }
  for (i = 0; i < 10; i++)
  {
    cout << pqs.top() << endl;
    pqs.pop();
  }

  // Make a priority queue of strings using a deque
  // container, and greater as the compare operation
  priority_queue<string,deque<string>, greater<string> >
     pgqs;

  // Push on a few strings then pop them back off
  for (i = 0; i < 10; i++)
  {
    pgqs.push(string(i+1,'a'));
    cout << pgqs.top() << endl;
  }

  for (i = 0; i < 10; i++)
  {
    cout << pgqs.top() << endl;
    pgqs.pop();
  }

  return 0;
}
```

Program Output

```
2
1
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
```

©Copyright 1999 Rogue Wave Software, Inc.

```
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
a
a
a
a
a
a
a
a
a
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Warnings

If your compiler does not support default template parameters, you must always include a `Container` template parameter and a `Compare` template parameter when declaring an instance of ***priority_queue***. For example, you would not be able to write:

```
priority_queue<int> var;
```

Instead, you would have to write:

```
priority_queue<int, vector<int>,
  less<typename vector<int>::value_type> > var;
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Containers, ***queue***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

ptr_fun

`Function Adaptor`

Summary

A function that is overloaded to adapt a pointer to a function, to take the place of a function.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
  ptr_fun (Result (*f)(Arg));

template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
  ptr_fun (Result (*x)(Arg1, Arg2));
```

Description

The ***pointer_to_unary_function*** and ***pointer_to_binary_function*** classes encapsulate pointers to functions and use `operator()` so that the resulting object serves as a function object for the function.

The `ptr_fun` function is overloaded to create instances of ***pointer_to_unary_function*** or ***pointer_to_binary_function*** when included with the appropriate pointer to a function.

Example

```
//
// pnt2fnct.cpp
//
 #include <functional>
 #include <deque>
 #include <vector>
 #include <algorithm>
 #include <iostream>
 using namespace std;

 //Create a function
 int factorial(int x)
 {
   int result = 1;
   for(int i = 2; i <= x; i++)
       result *= i;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  return result;
}

int main()
{
  //Initialize a deque with an array of ints
  int init[7] = {1,2,3,4,5,6,7};
  deque<int> d(init, init+7);

  //Create an empty vector to store the factorials
  vector<int> v((size_t)7);

  //Transform the numbers in the deque to their
  //factorials and store in the vector
  transform(d.begin(), d.end(), v.begin(),
            ptr_fun(factorial));

  //Print the results
  cout << "The following numbers: " << endl << "     ";
  copy(d.begin(),d.end(),
       ostream_iterator<int,char>(cout," "));

  cout << endl << endl;
  cout << "Have the factorials: " << endl << "     ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));

  return 0;
}
```

Program Output

```
The following numbers:
     1 2 3 4 5 6 7
Have the factorials:
     1 2 6 24 120 720 5040
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Function Objects, *pointer_to_binary_function*, *pointer_to_unary_function*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

push_heap

`Algorithms`

Summary

Places a new element into a heap.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
  void
  push_heap(RandomAccessIterator first,
            RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  push_heap(RandomAccessIterator first,
            RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** `algorithm, or a new element may be added by the` ***push_heap*** `algorithm, in O(logN) time.`

These properties make heaps useful as priority queues.

The ***push_heap*** algorithms uses the less than (`<`) operator as the default comparison. As with all of the heap manipulation algorithms, an alternate comparison function can be specified.

The ***push_heap*** algorithm is used to add a new element to the heap. First, a new element for the heap is added to the end of a range. (For example, you can use the ***vector*** or ***deque*** member function `push_back()`to add the element to the end of either of those containers.) The ***push_heap*** algorithm assumes that the range `[first, last - 1)` is a valid heap. Then it properly positions the element in the location `last - 1` into its proper position in the heap, resulting in a heap over the range `[first, last)`.

©Copyright 1999 Rogue Wave Software, Inc.

Note that the ***push_heap*** algorithm does not place an element into the heap's underlying container. You must user another function to add the element to the end of the container before applying ***push_heap***.

Complexity

For ***push_heap*** at most `log(last - first)` comparisons are performed.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main(void)
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

   // Make heaps
   make_heap(v1.begin(),v1.end());
   make_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
   // Note that x, y and z represent the remaining
   // values in the container (other than 4).
   // The definition of the heap and heap operations
   // does not require any particular ordering
   // of these values.

   // Copy both vectors to cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now let's pop
   pop_heap(v1.begin(),v1.end());
   pop_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (3,x,y,4) and v2 = (3,x,y,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // And push
   push_heap(v1.begin(),v1.end());
   push_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z) and v2 = (4,x,y,z)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now sort those heaps
   sort_heap(v1.begin(),v1.end());
   sort_heap(v2.begin(),v2.end(),less<int>());
   // v1 = v2 = (1,2,3,4)


// Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  cout << endl;

  return 0;
}
```

Program Output

```
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

make_heap, ***pop_heap***, ***sort_heap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

queue

`Container Adaptor`

Summary

A container adaptor that behaves like a queue (first in, first out).

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

back()	
empty()	operator==()
front()	pop()
operator!=()	push()
operator>()	size()
operator<()	

Synopsis

```
#include <queue>
template <class T, class Container = deque<T> >
  class queue ;
```

Description

The ***queue*** container adaptor lets a container act as a queue. In a queue, items are pushed into the back of the container and removed from the front. The first items pushed into the queue are the first items to be popped off of the queue (first in, first out, or "FIFO").

queue can adapt any container that supports the `front()`, `back()`, `push_back()`, and `pop_front()` operations. In particular, ***deque*** and ***list*** can be used.

Interface

```
template <class T, class Container = deque<T> >
 class queue {

public:

// typedefs
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef Container container_type;

// Construct/Copy/Destroy
   explicit queue (const Container& = Container());

// Accessors

   bool empty () const;
   size_type size () const;
   value_type& front ();
   const value_type& front () const;
   value_type& back ();
   const value_type& back () const;
   void push (const value_type&);
   void pop ();
};

// Non-member Operators

template <class T, class Container>
 bool operator== (const queue<T, Container>&,
                  const queue<T, Container>&);

template <class T, class Container>
 bool operator!= (const queue<T, Container>&,
                  const queue<T, Container>&);

template <class T, class Container>
 bool operator< (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator> (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator<= (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator>= (const queue<T, Container>&,
                 const queue<T, Container>&);
```

Constructors

```
explicit queue (const Container& = Container());
```

Creates a queue of zero elements. The queue uses the allocator `Allocator()` for all storage management.

Member Functions

```
value_type&
back ();
```

Returns a reference to the item at the back of the queue (the last item pushed into the queue).

```
const value_type&
back() const;
```

Returns a constant reference to the item at the back of the queue as a `const_value_type`.

```
bool
empty () const;
```

Returns `true` if the queue is empty, otherwise `false`.

```
value_type&
front ();
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a reference to the item at the front of the queue. This is the first item pushed onto the queue unless `pop()` has been called since then.

```
const value_type&
front () const;
```

Returns a constant reference to the item at the front of the queue as a `const_value_type`.

```
void
pop ();
```

Removes the item at the front of the queue.

```
void
push (const value_type& x);
```

Pushes `x` onto the back of the queue.

```
size_type
size () const;
```

Returns the number of elements on the queue.

Non-member Operators

```
template <class T, class Container>
  bool operator== (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T, class Container>
  bool operator!= (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Returns `!(x==y)`.

```
template <class T, class Container>
  bool operator< (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `true` if the queue defined by the elements contained in `x` is lexicographically less than the queue defined by the elements contained in `y`.

```
template <class T, class Container>
  bool operator> (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `y < x`.

```
template <class T, class Container>
  bool operator<= (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Returns `!(y < x)`.

```
template <class T, class Container>
  bool operator>= (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Returns `!(x < y)`.

Example

```
//
// queue.cpp
//
 #include <queue>
 #include <string>
 #include <deque>
 #include <list>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <iostream>
using namespace std;

int main(void)
{
  // Make a queue using a list container
  queue<int, list<int>> q;

  // Push a couple of values on then pop them off
  q.push(1);
  q.push(2);
  cout << q.front() << endl;
  q.pop();
  cout << q.front() << endl;
  q.pop();

  // Make a queue of strings using a deque container
  queue<string,deque<string>> qs;

  // Push on a few strings then pop them back off
  int i;
  for (i = 0; i < 10; i++)
  {
    qs.push(string(i+1,'a'));
    cout << qs.front() << endl;
  }
  for (i = 0; i < 10; i++)
  {
    cout << qs.front() << endl;
    qs.pop();
  }

  return 0;
}
```

Program Output

```
1
2
a
a
a
a
a
a
a
a
a
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Warnings

If your compiler does not support default template parameters, you must always include a `Container` template parameter. For example you would not be able to write:

```
queue<int> var;
```

rather, you would have to write,

```
queue<int, deque<int> > var;
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

See Also

allocator, *Containers*, *priority_queue*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Random Access Iterators

`Iterator`

Summary

An iterator that reads, writes, and allows random access to a container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Description

NOTE:For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Random access iterators can read, write, and allow random access to the containers they serve. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a and b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp and m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Random Access Iterators

The following expressions must be valid for random access iterators:

`X u`	u might have a singular value
`X()`	X() might be singular
`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`r = a`	assignment, `r == a`
`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`

©Copyright 1999 Rogue Wave Software, Inc.

`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return value convertible to const `X&`
`*r--`	returns `T&`
`r += n`	Semantics of `--r` or `++r` `n` times depending on the sign of `n`
`a + n, n + a`	returns type `X`
`r -= n`	returns `X&`, behaves as `r += -n`
`a - n`	returns type `X`
`b - a`	returns `distance`
`a[n]`	*(a+n), return value convertible to `T`
`a < b`	total ordering relation
`a > b`	total ordering relation opposite to `<`
`a <= b`	!(a < b)
`a >= b`	!(a > b)

Like forward iterators, random access iterators have the condition that `a == b` implies `*a == *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

All relational operators return a value convertible to `bool`.

See Also

Iterators, ***Forward Iterators***, ***Bidirectional Iterators***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

random_shuffle

`Algorithm`

Summary

Randomly shuffles elements of a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
 void random_shuffle (RandomAccessIterator first,
                      RandomAccessIterator last);

template <class RandomAccessIterator,
          class RandomNumberGenerator>
 void random_shuffle (RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator& rand);
```

Description

The ***random_shuffle*** algorithm shuffles the elements in the range `[first, last)` with uniform distribution. ***random_shuffle*** can take a particular random number generating function object `rand` (where `rand` takes a positive argument `n` of distance `type` of the `RandomAccessIterator`) and returns a randomly chosen value between `0` and `n - 1`.

Complexity

In the ***random_shuffle*** algorithm, `(last - first) -1` swaps are done.

Example

```
//
// rndshufl.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;
 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   vector<int> v(arr, arr+10);
   //Print out elements in original (sorted) order
   cout << "Elements before random_shuffle: " << endl
        << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //Mix them up with random_shuffle
   random_shuffle(v.begin(), v.end());
   //Print out the mixed up elements
   cout << "Elements after random_shuffle: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
Elements before random_shuffle:
     1 2 3 4 5 6 7 8 9 10
Elements after random_shuffle:
     7 9 10 3 2 5 4 8 1 6
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

raw_storage_iterator

`Memory Management`

Summary

Enables iterator-based algorithms to store results into uninitialized memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator++()
operator=()

Synopsis

```
#include <memory>
template <class OutputIterator, class T>
 class raw_storage_iterator : public
       iterator <output_iterator_tag, void,void,void,void> {

public:
   explicit raw_storage_iterator (OutputIterator);
   raw_storage_iterator<OutputIterator, T>& operator*();
   raw_storage_iterator<OutputIterator, T>&
     operator= (const T&);
   raw_storage_iterator<OutputIterator, T>& operator++();
   raw_storage_iterator<OutputIterator, T> operator++ (int);
};
```

Description

Class ***raw_storage_iterator*** enables iterator-based algorithms to store their results in uninitialized memory. The template parameter, `OutputIterator` is required to have its `operator*` return an object for which `operator&` is both defined and returns a pointer to `T`.

Constructors

```
raw_storage_iterator (OutputIterator x);
```

Initializes the iterator to point to the same value as `x`.

Member Operators

```
raw_storage_iterator <OutputIterator, T>&
operator=(const T& element);
```

Constructs an instance of `T`, initialized to the value `element`, at the location pointed to by the iterator.

©Copyright 1999 Rogue Wave Software, Inc.

```
raw_storage_iterator <OutputIterator, T>&
operator++();
```

Pre-increment: advances the iterator and returns a reference to the updated iterator.

```
raw_storage_iterator<OutputIterator, T>
operator++(int);
```

Post-increment: advances the iterator and returns the old value of the iterator.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

remove

`Algorithm`

Summary

Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
ForwardIterator
remove (ForwardIterator first,
        ForwardIterator last,
        const T& value);
```

Description

The ***remove*** algorithm eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following condition holds: `*i == value`. ***remove*** returns an iterator that points to the end of the resulting range. ***remove*** is stable, which means that the relative order of the elements that are not removed is the same as their relative order in the original range.

remove does not actually reduce the size of the sequence. It actually: 1) copies the values that are to be *retained* to the front of the sequence, and 2) returns an iterator that describes where the sequence of retained values ends. Elements that follow this iterator are simply the original sequence values, left unchanged. Here's a simple example:

Say we want to remove all values of "2" from the following sequence:

```
354621271
```

Applying the ***remove*** algorithm results in the following sequence:

```
3546171|XX
```

The vertical bar represents the position of the iterator returned by ***remove***. Note that the elements to the left of the vertical bar are the original sequence with the "2s" removed.

If you want to actually delete items from the container, use the following technique:

©Copyright 1999 Rogue Wave Software, Inc.

```
container.erase(remove(first,last,value),container.end());
```

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>

 using namespace std;

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg& x){ return 1; }
 };
 int main ()
 {
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove the 7
   vector<int>::iterator result =
           remove(v.begin(), v.end(), 7);
   // delete dangling elements from the vector
   v.erase(result, v.end());
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove everything beyond the fourth element
   result = remove_if(v.begin()+4,
                      v.begin()+8, all_true<int>());
   // delete dangling elements
   v.erase(result, v.end());
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   return 0;
 }
```

Program Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

©Copyright 1999 Rogue Wave Software, Inc.

remove_if, ***remove_copy***, ***remove_copy_if***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

remove_copy

`Algorithm`

Summary

Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class OutputIterator,
          class T>
 OutputIterator remove_copy (InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             const T& value);
```

Description

The ***remove_copy*** algorithm copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding condition does *not* hold: `*i == value`. ***remove_copy*** returns an iterator that points to the end of the resulting range. ***remove_copy*** is stable, which means that the relative order of the elements in the resulting range is the same as their relative order in the original range. The elements in the original sequence are not altered by ***remove_copy***.

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class Arg>
struct all_true : public unary_function<Arg, bool>
{
  bool operator() (const Arg&) { return 1; }
};

int main ()
{
  int arr[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(arr+0, arr+10);

  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Remove the 7.
  //
  vector<int>::iterator result = remove(v.begin(),
                        v.end(), 7);
  //
  // Delete dangling elements from the vector.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Remove everything beyond the fourth element.
  //
  result = remove_if(v.begin()+4, v.begin()+8,
           all_true<int>());
  //
  // Delete dangling elements.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Now remove all 3s on output.
  //
  remove_copy(v.begin(), v.end(),
              ostream_iterator<int,char>(cout," "), 3);
  cout << endl << endl;
  //
  // Now remove everything satisfying predicate on output.
  // Should yield a NULL vector.
  //
  remove_copy_if(v.begin(), v.end(),
                 ostream_iterator<int,char>(cout," "),
                 all_true<int>());

  return 0;
}
```

Program Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

remove, ***remove_if***, ***remove_copy_if***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

remove_copy_if

`Algorithm`

Summary

Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class OutputIterator,
          class Predicate>
 OutputIterator remove_copy_if (InputIterator first,
                                InputIterator last,
                                OutputIterator result,
                                Predicate pred);
```

Description

The ***remove_copy_if*** algorithm copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following condition does *not* hold: `pred(*i) == true`. ***remove_copy_if*** returns an iterator that points to the end of the resulting range. ***remove_copy_if*** is stable, which means that the relative order of the elements in the resulting range is the same as their relative order in the original range.

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class Arg>
struct all_true : public unary_function<Arg, bool>
{
  bool operator() (const Arg&) { return 1; }
};

int main ()
{
  int arr[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(arr+0, arr+10);


  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Remove the 7.
  //
  vector<int>::iterator result = remove(v.begin(),
                                  v.end(), 7);
  //
  // Delete dangling elements from the vector.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Remove everything beyond the fourth element.
  //
  result = remove_if(v.begin()+4, v.begin()+8,
           all_true<int>());
  //
  // Delete dangling elements.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Now remove all 3s on output.
  //
  remove_copy(v.begin(), v.end(),
              ostream_iterator<int>(cout," "), 3);
  cout << endl << endl;
  //
  // Now remove everything satisfying predicate on output.
  // Should yield a NULL vector.
  //
  remove_copy_if(v.begin(), v.end(),
                 ostream_iterator<int,char>(cout," "),
                 all_true<int>());

  return 0;
}
```

Program Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

remove, ***remove_if***, ***remove_copy***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

remove_if

`Algorithm`

Summary

Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if (ForwardIterator first,
                           ForwardIterator last,
                           Predicate pred);
```

Description

The ***remove_if*** algorithm eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding condition holds: `pred(*i) == true`. ***remove_if*** returns an iterator that points to the end of the resulting range. ***remove_if*** is stable, which means that the relative order of the elements that are not removed is the same as their relative order in the original range.

remove_if does not actually reduce the size of the sequence. It actually: 1) copies the values that are to be *retained* to the front of the sequence, and 2) returns an iterator that describes where the sequence of retained values ends. Elements that follow this iterator are simply the original sequence values, left unchanged. Here's a simple example:

Say we want to remove all even numbers from the following sequence:

```
123456789
```

Applying the ***remove_if*** algorithm results in the following sequence:

```
13579|XXXX
```

The vertical bar represents the position of the iterator returned by ***remove_if***. Note that the elements to the left of the vertical bar are the original sequence with the even numbers removed. The elements to the right of the bar are simply the untouched original members of the original sequence.

If you want to actually delete items from the container, use the following technique:

©Copyright 1999 Rogue Wave Software, Inc.

```
container.erase(remove(first,last,value),container.end());
```

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;
 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg& x){ return 1; }
 };
 int main ()
 {
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove the 7
   vector<int>::iterator result =
           remove(v.begin(), v.end(), 7);
   // delete dangling elements from the vector
   v.erase(result, v.end());
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove everything beyond the fourth element
   result = remove_if(v.begin()+4,
                      v.begin()+8, all_true<int>());
   // delete dangling elements
   v.erase(result, v.end());
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   return 0;
 }
```

Program Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

remove, *remove_copy*, *remove_copy_if*

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

replace

`Algorithm`

Summary

Substitutes elements in a collection with new values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
void replace (ForwardIterator first,
              ForwardIterator last,
              const T& old_value,
              const T& new_value);
```

Description

For the range `[first, last)`, the ***replace*** algorithm replaces elements referred to by iterator `i` with `new_value`, when the following condition holds: `*i == old_value`.

Complexity

Exactly `last - first` comparisons or applications of the corresponding predicate are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg&){ return 1; }
 };
```

©Copyright 1999 Rogue Wave Software, Inc.

```
int main()
{

  //Initialize a vector with an array of integers
  int arr[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(arr, arr+10);

  //Print out original vector
  cout << "The original list: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int,char>
      (cout," "));
  cout << endl << endl;

  //Replace the number 7 with 11
  replace(v.begin(), v.end(), 7, 11);


  // Print out vector with 7 replaced,
  // s.b. 1 2 3 4 5 6 11 8 9 10
  cout << "List after replace " << endl << "     ";
  copy(v.begin(),v.end(),o
       stream_iterator<int,char>(cout," "));
  cout << endl << endl;

  //Replace 1 2 3 with 13 13 13
  replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);

  // Print out the remaining vector,
  // s.b. 13 13 13 4 5 6 11 8 9 10
  cout << "List after replace_if " << endl << "     ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  return 0;
}
```

Program Output

```
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

replace_if, *replace_copy*, *replace_copy_if*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

replace_copy

`Algorithm`

Summary

Substitutes elements in a collection with new values, and moves the revised sequence into `result`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class OutputIterator,
          class T>
OutputIterator replace_copy (InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             const T& old_value,
                             const T& new_value);
```

Description

The ***replace_copy*** algorithm leaves the original sequence intact and places the revised sequence into `result`. For the range `[first, last)`, the algorithm compares elements referred to by iterator `i` with `old_value`. If `*i` does not equal `old_value`, then ***replace_copy*** copies `*i` to `result+(first-i)`. If `*i==old_value`, then ***replace_copy*** copies `new_value` to `result+(first-i)`. ***replace_copy*** returns `result+(last-first)`.

Complexity

Exactly `last - first` comparisons between values are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;

 template<class Arg>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 struct all_true: public unary_function<Arg, bool>
 {
   bool operator() (const Arg&) {return 1;}
};


 int main ()
 {
   //
   // Initialize a vector with an array of integers.
   //
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr+0, arr+10);
   //
   // Print out original vector.
   //
   cout << "The original list: "<< endl << "     ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace the number 7 with 11.
   //
   replace(v.begin(), v.end(), 7, 11);
   //
   // Print out vector with 7 replaced.
   //
   cout << "List after replace:" << endl << "     ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace 1 2 3 with 13 13 13.
   //
   replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);
   //
   // Print out the remaining vector.
   //
   cout << "List after replace_if:" << endl << "     ";
   copy(v.begin(), v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace those 13s with 17s on output.
   //
   cout << "List using replace_copy to cout:" << endl
        << "     ";
   replace_copy(v.begin(), v.end(),
                   ostream_iterator<int,char>(cout, " "),
                13, 17);
   cout << endl << endl;
   //
   // A simple example of replace_copy_if.
   //
   cout << "List w/ all elements output as 19s:" << endl
       << "    ";
   replace_copy_if(v.begin(), v.end(),
                   ostream_iterator<int,char>(cout, " "),
                   all_true<int>(), 19);
   cout << endl;

   return 0;
 }
```

Program Output

```
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

replace, ***replace_if***, ***replace_copy_if***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

replace_copy_if

`Algorithm`

Summary

Substitutes elements in a collection with new values, and moves the revised sequence into `result`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class OutputIterator,
          class Predicate,
          class T>
OutputIterator replace_copy_if (InputIterator first,
                                InputIterator last,
                                OutputIterator result,
                                Predicate pred,
                                const T& new_value);
```

Description

The ***replace_copy_if*** algorithm leaves the original sequence intact and places a revised sequence into `result`. For the range `[first,last)`, the algorithm compares each element `*i` with the conditions specified by `pred`. If `pred(*i)==false`, ***replace_copy_if*** copies `*i` to `result+(first-i)`. If `pred(*i)==true`, then ***replace_copy*** copies `new_value` to `result+(first-i)`. ***replace_copy_if*** returns `result+(last-first)`.

Complexity

Exactly `last - first` applications of the predicate are performed.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class Arg>
struct all_true : public unary_function<Arg, bool>
{
  bool operator() (const Arg&) { return 1; }
};

int main ()
{
  //
  // Initialize a vector with an array of integers.
  //
  int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
  vector<int> v(arr+0, arr+10);
  //
  // Print out original vector.
  //
  cout << "The original list: " << endl << "     ";
  copy(v.begin(), v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Replace the number 7 with 11.
  //
  replace(v.begin(), v.end(), 7, 11);
  //
  // Print out vector with 7 replaced.
  //
  cout << "List after replace:" << endl << "     ";
  copy(v.begin(), v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Replace 1 2 3 with 13 13 13.
  //
  replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);
  //
  // Print out the remaining vector.
  //
  cout << "List after replace_if:" << endl << "     ";
  copy(v.begin(), v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Replace those 13s with 17s on output.
  //
  cout << "List using replace_copy to cout:" << endl
       << "     ";
  replace_copy(v.begin(), v.end(),
               ostream_iterator<int,char>(cout, " "),
               13, 17);
  cout << endl << endl;
  //
  // A simple example of replace_copy_if.
  //
  cout << "List w/ all elements output as 19s:" << endl
      << "   ";
  replace_copy_if(v.begin(), v.end(),
                  ostream_iterator<int,char>(cout, " "),
                  all_true<int>(), 19);
  cout << endl;

  return 0;
}
```

Program Output

```
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

replace, ***replace_if***, ***replace_copy***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

replace_if

`Algorithm`

Summary

Substitutes elements in a collection with new values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator,
          class Predicate,
          class T>
void replace_if (ForwardIterator first,
                 ForwardIterator last,
                 Predicate pred
                 const T& new_value);
```

Description

The ***replace_if*** algorithm replaces element referred to by iterator `i` in the range `[first, last)` with `new_value` when the following condition holds: `pred(*i) == true`.

Complexity

Exactly `last - first` applications of the predicate are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream>
 using namespace std;

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg&){ return 1; }
```

©Copyright 1999 Rogue Wave Software, Inc.

```
};

int main()
{

  //Initialize a vector with an array of integers
  int arr[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(arr, arr+10);

  //Print out original vector
  cout << "The original list: " << endl << "     ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  //Replace the number 7 with 11
  replace(v.begin(), v.end(), 7, 11);

  // Print out vector with 7 replaced,
  // s.b. 1 2 3 4 5 6 11 8 9 10
  cout << "List after replace " << endl << "     ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  //Replace 1 2 3 with 13 13 13
  replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);

  // Print out the remaining vector,
  // s.b. 13 13 13 4 5 6 11 8 9 10
  cout << "List after replace_if " << endl << "     ";
  copy(v.begin(),v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  return 0;
}
```

Program Output

```
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

replace, *replace_copy*, *replace_copy_if*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

return_temporary_buffer

`Memory Handling Primitive`

Summary

A pointer-based primitive for handling memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

```
#include <memory>
template <class T>
void return_temporary_buffer (T* p, T*);
```

Description

The ***return_temporary_buffer*** templatized function returns a buffer, previously allocated through ***get_temporary_buffer***, to available memory. Parameter `p` points to the buffer.

See Also

allocator, ***get_temporary_buffer***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

reverse

`Algorithm`

Summary

Reverses the order of elements in a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator>
void reverse (BidirectionalIterator first,
              BidirectionalIterator last);
```

Description

The algorithm ***reverse*** reverses the elements in a sequence so that the last element becomes the new first element, and the first element becomes the new last. For each non-negative integer `i <= (last - first)/2`, ***reverse*** applies ***swap*** to all pairs of iterators `first + I, (last - I) - 1`.

Because the iterators are assumed to be bidirectional, ***reverse*** does not return anything.

Complexity

reverse performs exactly `(last - first)/2` swaps.

Example

```
//
// reverse.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;
 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out elements in original (sorted) order
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   cout << "Elements before reverse: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Reverse the ordering
   reverse(v.begin(), v.end());

   //Print out the reversed elements
   cout << "Elements after reverse: " << endl << "     ";

   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
Elements before reverse:
     1 2 3 4 5 6 7 8 9 10
Elements after reverse:
     10 9 8 7 6 5 4 3 2 1
A reverse_copy to cout:
     1 2 3 4 5 6 7 8 9 10
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

reverse_copy, *swap*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

__reverse_bi_iterator, reverse_iterator

`Iterator`

Summary

An iterator that traverses a collection backwards. ***__reverse_bi_iterator*** is included for those compilers that do not support partial specialization. The template signature for ***reverse_iterator*** matches that of ***__reverse_bi_iterator*** when partial specialization is not available (in other words, it has six template parameters rather than one).

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iterator>
template <class Iterator,
          class Category,
          class T,
          class Reference = T&,
          class Pointer = T*
          class Distance = ptrdiff_t>
class __reverse_bi_iterator ;

template <class Iterator>
class reverse_iterator ;
```

Description

The iterators ***reverse_iterator*** and ***__reverse_bi_iterator*** correspond to ***random_access_iterator*** and ***bidirectional_iterator***, except that they traverse collections in the opposite direction. The fundamental relationship between a reverse iterator and its corresponding iterator `i` is established by the identity:

```
&*(reverse_iterator(i)) == &*(i-1);
```

This mapping is dictated by the fact that, while there is always a pointer past the end of a container, there might not be a valid pointer before its beginning.

The following are true for ***__reverse_bi_iterators***:

- These iterators may be instantiated with the default constructor or by a single argument constructor that initializes the new `__reverse_bi_iterator` with a `bidirectional_iterator`.
- `operator*` returns a reference to the current value.

©Copyright 1999 Rogue Wave Software, Inc.

- `operator++` advances the iterator to the previous item (`--current`) and returns a reference to `*this`.
- `operator++(int)` advances the iterator to the previous item (`--current`) and returns the old value of `*this`.
- `operator--` advances the iterator to the following item (`++current`) and returns a reference to `*this`.
- `operator--(int)` advances the iterator to the following item (`++current`) and returns the old value of `*this.`
- `operator==` is a non-member operator that returns `true` if the iterators `x` and `y` point to the same item.

The following are true for ***reverse_iterator***s:

- These iterators may be instantiated with the default constructor or by a single argument constructor that initializes the new `reverse_iterator` with a `random_access_iterator`.
- `operator*` returns a reference to the current value.
- `operator++` advances the iterator to the previous item (`--current`) and returns a reference to `*this`.
- `operator++(int)` advances the iterator to the previous item (`--current`) and returns the old value of `*this`.
- `operator--` advances the iterator to the following item (`++current`) and returns a reference to `*this`.
- `operator--(int)` advances the iterator to the following item (`++current`) and returns the old value of `*this`.
- `operator==` is a non-member operator that returns `true` if the iterators `x` and `y` point to the same item.
- `operator!=` is a non-member operator that returns `!(x==y).`
- `operator<` is a non-member operator that returns `true` if the iterator `x` precedes the iterator `y`.
- `operator>` is a non-member operator that returns `y < x`.
- `operator<=` is a non-member operator that returns `!(y < x)`.
- `operator>=` is a non-member operator that returns `!(x < y)`.
- The remaining operators (`<, +, -, +=, -=`) are redefined to behave exactly as they would in a `random_access_iterator`, except with the sense of direction reversed.

Complexity

All iterator operations are required to take at most amortized constant time.

Interface

```
template <class Iterator,
          class Category,
          class T,
          class Reference = T&,
          class Pointer = T*,
          class Distance = ptrdiff_t>
class __reverse_bi_iterator
   : public iterator<bidirectional_iterator_tag, Category T,
                     Distance> {
    typedef
      __reverse_bi_iterator<Iterator,
      Category T, Reference, Pointer, Distance> self;
    friend bool operator== (const self&, const self&);
  public:
    __reverse_bi_iterator ();
    explicit __reverse_bi_iterator
      (Iterator);
    Iterator base ();
    Reference operator* ();
    self& operator++ ();
    self operator++ (int);
    self& operator-- ();
    self operator-- (int);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  };

// Non-member Operators

  template <class Iterator, class Category
            class T, class Reference,
            class Pointer, class Distance>
  bool operator== (
    const __reverse_bi_iterator
     <Iterator,Category T,Reference,Pointer,Distance>&,
     const __reverse_bi_iterator
     <Iterator,Category T,Reference,Pointer,Distance>&);

template <class Iterator,
            class T, class Reference, class Category,
            class Pointer, class Distance>
  bool operator!= (
    const __reverse_bi_iterator
     <Iterator,Category T,Reference,Pointer,Distance>&,
     const __reverse_bi_iterator
     <Iterator,Category T,Reference,Pointer,Distance>&);

template <class Iterator>
class reverse_iterator
  : public iterator
     <typename iterator_traits<iterator>::iterator_category,
     <typename iterator_traits<iterator>::value_type,
     <typename iterator_traits<iterator>::difference_type,
     <typename iterator_traits<iterator>::pointer,
     <typename iterator_traits<iterator>::reference>
{

    typedef reverse_iterator<Iterator> self;

    friend bool operator==    (const self&, const self&);
    friend bool operator<     (const self&, const self&);
    friend Distance operator- (const self&, const self&);
    friend self operator+     (Distance, const self&);

public:
    reverse_iterator ();
    explicit reverse_iterator (Iterator);
    Iterator base ();
    Reference operator* ();
    self& operator++ ();
    self operator++ (int);
    self& operator-- ();
    self operator-- (int);

    self  operator+ (Distance) const;
    self& operator+= (Distance);
    self operator- (Distance) const;
    self& operator-= (Distance);
    Reference operator[] (Distance);
};

// Non-member Operators

    template <class Iterator> bool operator==
        const reverse_iterator<Iterator>&,
        const reverse_iterator<Iterator>&);

template <class Iterator> bool operator!=
        const reverse_iterator<Iterator>&,
        const reverse_iterator<Iterator>&);

     template <class Iterator> bool operator< (
         const reverse_iterator<Iterator>&,
         const reverse_iterator<Iterator>&);

template <class Iterator> bool operator>
         const reverse_iterator<Iterator>&,
         const reverse_iterator<Iterator>&);

template <class Iterator> bool operator<=
         const reverse_iterator<Iterator>&,
         const reverse_iterator<Iterator>&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class Iterator> bool operator>=
         const reverse_iterator<Iterator>&,
         const reverse_iterator<Iterator>&);

     template <class Iterator> Distance operator-
         const reverse_iterator<Iterator>&,
         const reverse_iterator<Iterator>&);

     template <class Iterator>
      reverse_iterator<Iterator> operator+ Distance,
         const reverse_iterator<Iterator>&);
```

Example

```
//
// rev_itr.cpp
//
 #include <iterator>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main()
 {
   //Initialize a vector using an array
   int arr[4] = {3,4,7,8};
   vector<int> v(arr,arr+4);
   //Output the original vector
   cout << "Traversing vector with iterator: " << endl
        << "     ";
   for(vector<int>::iterator i = v.begin(); i != v.end();
       i++)
     cout << *i << " ";
   //Declare the reverse_iterator
   vector<int>::reverse_iterator rev(v.end());
   vector<int>::reverse_iterator rev_end(v.begin());
   //Output the vector backwards
   cout << endl << endl;
   cout << "Same vector, same loop, reverse_iterator: "
         << endl << "     ";
   for(; rev != rev_end; rev++)
     cout << *rev << " ";
   return 0;
 }
```

Program Output

```
Traversing vector with iterator:
     3 4 7 8
Same vector, same loop, reverse_iterator:
     8 7 4 3
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

Iterators

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

reverse_copy

`Algorithm`

Summary

Reverses the order of elements in a collection while copying them to a new collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy (BidirectionalIterator first,
                             BidirectionalIterator last,
                             OutputIterator result);
```

Description

The ***reverse_copy*** algorithm copies the range `[first, last)` to the range `[result, result + (last - first))` such that for any non- negative integer `i < (last - first)`, the following assignment takes place:

```
*(result + (last - first) -i) = *(first + i)
```

reverse_copy returns `result + (last - first)`. The ranges `[first, last)` and `[result, result + (last - first))` must not overlap.

Complexity

reverse_copy performs exactly `(last - first)` assignments.

Example

```
//
// reverse.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main ()
 {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  //
  // Initialize a vector with an array of integers.
  //
  int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
  vector<int> v(arr+0, arr+10);
  //
  // Print out elements in original (sorted) order.
  //
  cout << "Elements before reverse: " << endl << "     ";
  copy(v.begin(), v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Reverse the ordering.
  //
  reverse(v.begin(), v.end());
  //
  // Print out the reversed elements.
  //
  cout << "Elements after reverse: " << endl << "     ";
  copy(v.begin(), v.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  cout << "A reverse_copy to cout: " << endl << "     ";
  reverse_copy(v.begin(), v.end(),
               ostream_iterator<int,char>(cout, " "));
  cout << endl;

  return 0;
}
```

Program Output

```
Elements before reverse:
     1 2 3 4 5 6 7 8 9 10
Elements after reverse:
     10 9 8 7 6 5 4 3 2 1
A reverse_copy to cout:
     1 2 3 4 5 6 7 8 9 10
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

reverse

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

reverse_iterator

See the *__reverse_bi_iterator* section of this reference.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

rotate, rotate_copy

`Algorithm`

Summary

Swaps the segment that contains elements from `first` through `middle-1` with the segment that contains the elements from `middle` through `last`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator>
void rotate (ForwardIterator first,
             ForwardIterator middle,
             ForwardIterator last);

template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy (ForwardIterator first,
                            ForwardIterator middle,
                            ForwardIterator last,
                            OutputIterator result);
```

Description

The ***rotate*** algorithm takes three iterator arguments: `first`, which defines the start of a sequence; `last`, which defines the end of the sequence; and `middle`, which defines a point within the sequence. ***rotate*** "swaps" the segment that contains elements from `first` through `middle-1` with the segment that contains the elements from `middle` through `last`. After ***rotate*** has been applied, the element that was in position `middle`, is in position `first`, and the other elements in that segment are in the same order relative to each other. Similarly, the element that was in position `first` is now in position `last-middle +1`. An example illustrates how ***rotate*** works:

Say that we have the sequence:

```
2 4 6 8 1 3 5
```

If we call ***rotate*** with `middle = 5`, the two segments are

```
2 4 6 8 and 1 3 5
```

After we apply rotate, the new sequence is:

```
1 3 5 2 4 6 8
```

©Copyright 1999 Rogue Wave Software, Inc.

Note that the element that was in the fifth position is now in the first position, and the element that was in the first position is in position 4 (`last - first + 1`, or 8 - 5 +1 =4).

The formal description of this algorithms is: for each non-negative integer `i < (last - first)`, ***rotate*** places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`. `[first, middle)` and `[middle, last)` are valid ranges.

rotate_copy rotates the elements as described above, but instead of swapping elements within the same sequence, it copies the result of the rotation to a container specified by `result`. ***rotate_copy*** copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non- negative integer `i < (last - first)` the following assignment takes place:

```
*(result + (i + (last - middle)) % (last -first)) = *(first + i).
```

The ranges `[first, last)` and `[result, result, + (last - first))` may not overlap.

Complexity

For ***rotate***, at most `last - first` swaps are performed.

For ***rotate_copy***, `last - first` assignments are performed.

Example

```
//
// rotate
//
 #include <algorithm>
 #include <vector>
 #include <iostream>
 using namespace std;

 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out elements in original (sorted) order
   cout << "Elements before rotate: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Rotate the elements
   rotate(v.begin(), v.begin()+4, v.end());
   //Print out the rotated elements
   cout << "Elements after rotate: " << endl << "     ";
   copy(v.begin(),v.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

Program Output

```
Elements before rotate:
     1 2 3 4 5 6 7 8 9 10
Elements after rotate:
     5 6 7 8 9 10 1 2 3 4
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

search, search_n

`Algorithm`

Summary

Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
 ForwardIterator1 search (ForwardIterator1 first1,
                          ForwardIterator1 last1,
                          ForwardIterator2 first2,
                          ForwardIterator2 last2);

template <class ForwardIterator1,
          class ForwardIterator2,
          class BinaryPredicate>
 ForwardIterator1 search (ForwardIterator1 first1,
                          ForwardIterator1 last1,
                          ForwardIterator2 first2,
                          ForwardIterator2 last2,
                          BinaryPredicate binary_pred);

template <class ForwardIterator,
          class Size,
          class T>
 ForwardIterator search_n (ForwardIterator first,
                          ForwardIterator last,
                          Size count, const T& value);

template <class ForwardIterator,
          class Size,
          class T,
          class BinaryPredicate>
 ForwardIterator search_n (ForwardIterator first,
                          ForwardIterator last,
                          Size count, const T& value,
                          BinaryPredicate pred)
```

Description

The ***search*** and ***search_n*** algorithms search for a sub-sequence within a sequence. The ***search*** algorithm searches for a sub-sequence `[first2, last2)` within a sequence `[first1, last1)`, and returns the beginning location of the sub-

©Copyright 1999 Rogue Wave Software, Inc.

sequence. If it does not find the sub-sequence, ***search*** returns `last1`. The first version of ***search*** uses the equality (`==`) operator as a default, and the second version allows you to specify a binary predicate to perform the comparison.

The ***search_n*** algorithm searches for the sub-sequence composed of `count` occurrences of `value` within a sequence `[first, last)`, and returns `first` if this sub-sequence is found. If it does not find the sub-sequence, ***search_n*** returns `last`. The first version of ***search_n*** uses the equality (`==`) operator as a default, and the second version allows you to specify a binary predicate to perform the comparison.

Complexity

search performs at most `(last1 - first1)*(last2-first2)` applications of the corresponding predicate.

search_n performs at most `(last - first)* count` applications of the corresponding predicate.

Example

```
//
// search.cpp
//
 #include <algorithm>
 #include <list>
 #include <iostream>
 using namespace std;

 int main()
 {
   // Initialize a list sequence and
   // sub-sequence with characters
   char seq[40] = "Here's a string with a substring in it";
   char subseq[10] = "substring";
   list<char> sequence(seq, seq+39);
   list<char> subseqnc(subseq, subseq+9);

   //Print out the original sequence
   cout << endl << "The sub-sequence, " << subseq
        << ", was found at the ";
   cout << endl << "location identified by a '*'"
        << endl << "     ";

   // Create an iterator to identify the location of
   // sub-sequence within sequence
   list<char>::iterator place;

   //Do search
   place = search(sequence.begin(), sequence.end(),
                  subseqnc.begin(), subseqnc.end());

   //Identify result by marking first character with a '*'
   *place = '*';

   //Output sequence to display result
   for(list<char>::iterator i = sequence.begin();
           i != sequence.end(); i++)
     cout << *i;
   cout << endl;

   return 0;
 }
```

Program Output

```
The sub-sequence, substring, was found at the
location identified by a '*'
     Here's a string with a *substring in it
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
list<char, allocator<char> >
```

©Copyright 1999 Rogue Wave Software, Inc.

instead of:

```
list<char>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Sequences

Summary

A ***sequence*** is a container that organizes a set of objects of the same type into a linear arrangement. ***vector***, ***list***, ***deque***, and ***string*** fall into this category.

Sequences offer different complexity trade-offs. ***vector*** offers fast inserts and deletes from the end of the container. ***deque*** is useful when insertions and deletions take place at the beginning or end of the sequence. Use ***list*** when there are frequent insertions and deletions from the middle of the sequence.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

See Also

For more information about sequences and their requirements, see the ***Containers*** section of this reference guide, or see the section on the specific container.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

set

`Container`

Summary

An associative container that supports unique keys. A ***set*** supports bidirectional iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

			operator=()
begin()	erase()	max_size()	operator==()
clear()	find()	operator!=()	rbegin()
count()	get_allocator()	operator>()	rend()
empty()	insert()	operator>=()	size()
end()	key_comp()	operator<()	swap()
equal_range()	lower_bound()	operator<=()	upper_bound()
			value_comp()

Synopsis

```
#include <set>
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> >
class set ;
```

Description

set<Key, Compare, Allocator> is an associative container that supports unique keys and allows for fast retrieval of the keys. A ***set*** contains, at most, one of any key value. The keys are sorted using `Compare`.

Since a ***set*** maintains a total order on its elements, you cannot alter the key values directly. Instead, you must insert new elements with an `insert_iterator`.

©Copyright 1999 Rogue Wave Software, Inc.

Any type used for the template parameter `Key` must include the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> >
class set {
public:
// types
typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
class iterator;
class const_iterator;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename std::reverse_iterator<iterator>
                     reverse_iterator;
typedef typename std::reverse_iterator<const_iterator>
                     const_reverse_iterator;

// Construct/Copy/Destroy
explicit set (const Compare& = Compare(),
              const Allocator& = Allocator ());
template <class InputIterator>
set (InputIterator, InputIterator,
     const Compare& = Compare(),
     const Allocator& = Allocator ());
set (const set<Key, Compare, Allocator>&);
~set ();
set<Key, Compare, Allocator>& operator=
   (const set <Key, Compare, Allocator>&);
allocator_type get_allocator () const;

// Iterators
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator rend ();
const_reverse_iterator rend () const;

// Capacity
bool empty () const;
size_type size () const;
size_type max_size () const;

// Modifiers
pair<iterator, bool> insert (const value_type&);
iterator insert (iterator, const value_type&);
template <class InputIterator>
void insert (InputIterator, InputIterator);
void erase (iterator);
size_type erase (const key_type&);
void erase (iterator, iterator);
void swap (set<Key, Compare, Allocator>&);
void clear ();
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// Observers
key_compare key_comp () const;
value_compare value_comp () const;

// Set operations
size_type count (const key_type&) const;
pair<iterator, iterator> equal_range (const  key_type&) const;
iterator find (const key_type&) const;
iterator lower_bound (const key_type&) const;
iterator upper_bound (const key_type&) const
};

// Non-member Operators
template <class Key, class Compare, class Allocator>
bool operator== (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);
template <class Key, class Compare, class Allocator>
bool operator!= (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);
template <class Key, class Compare, class Allocator>
bool operator< (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);
template <class Key, class Compare, class Allocator>
bool operator> (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);
template <class Key, class Compare, class Allocator>
bool operator<= (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);
template <class Key, class Compare, class Allocator>
bool operator>= (const set<Key, Compare, Allocator>&,
const set<Key, Compare, Allocator>&);

// Specialized Algorithms
template <class Key, class Compare, class Allocator>
void swap (set <Key, Compare, Allocator>&,
set <Key, Compare, Allocator>&);
```

Constructors

```
explicit
set(const Compare& comp = Compare(),
const Allocator& alloc = Allocator());
```

Creates a set of zero elements. If the function object `comp` is supplied, it is used to compare elements of the set. Otherwise, the default function object in the template argument is used. The template argument defaults to `less (<)`. The allocator alloc is used for all storage management.

```
template <class InputIterator>
set(InputIterator first, InputIterator last,
const Compare& comp = Compare()
const Allocator& alloc = Allocator());
```

Creates a set of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. If the function object `comp` is supplied, it is used to compare elements of the set. Otherwise, the default function object in the template argument is used. The template argument defaults to `less (<)`. Uses the allocator Allocator() for all storage management.

```
set(const set<Key, Compare, Allocator>& x);
```

Copy constructor. Creates a copy of `x`.

Destructors

```
~set();
```

Releases any allocated memory for self.

Assignment Operators

©Copyright 1999 Rogue Wave Software, Inc.

```
set<Key, Compare, Allocator>&
operator=(const set<Key, Compare, Allocator>& x);
```

Returns a reference to self. Self shares an implementation with `x`.

Allocators

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

Returns an `iterator` that points to the first element in self.

```
const_iterator
begin() const;
```

Returns a `const_iterator` that points to the first element in self.

```
iterator
end();
```

Returns an `iterator` that points to the past-the-end value.

```
const_iterator
end() const;
```

Returns a `const_iterator` that points to the past-the-end value.

```
reverse_iterator
rbegin();
```

Returns a `reverse_iterator` that points to the past-the-end value.

```
const_reverse_iterator
rbegin() const;
```

Returns a `const_reverse_iterator` that points to the past-the-end value.

```
reverse_iterator
rend();
```

Returns a `reverse_iterator` that points to the first element.

```
const_reverse_iterator
rend() const;
```

Returns a `const_reverse_iterator` that points to the first element.

Member Functions

```
void
clear();
```

Erases all elements from the set.

```
size_type
count(const key_type& x) const;
```

Returns the number of elements equal to `x`. Since a set supports unique keys, `count` always returns `1` or `0`.

```
bool
empty() const;
```

Returns `true` if the size is zero.

©Copyright 1999 Rogue Wave Software, Inc.

```
pair<iterator, iterator>
equal_range(const key_type&  x) const;
```

Returns `pair(lower_bound(x),upper_bound(x))`. The `equal_range` function indicates the valid range for insertion of `x` into the set.

```
size_type
erase(const key_type& x);
```

Deletes all the elements matching `x`. Returns the number of elements erased. Since a set supports unique keys, `erase` always returns `1` or `0`.

```
void
erase(iterator position);
```

Deletes the map element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
void
erase(iterator first, iterator last);
```

Deletes the elements in the range (`first, last`). Returns an `iterator` pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
iterator
find(const key_value& x) const;
```

Returns an `iterator` that points to the element equal to `x`. If there is no such element, the iterator points to the past-the-end value.

```
pair<iterator, bool>
insert(const value_type& x);
```

Inserts `x` into self according to the comparison function object. The template's default comparison function object is `less (<)`. If the insertion succeeds, it returns a pair composed of the iterator position where the insertion took place and `true`. Otherwise, the pair contains the end value and `false`.

```
iterator
insert(iterator position, const value_type& x);
```

`x` is inserted into the set. A position may be supplied as a hint regarding where to do the insertion. If the insertion is done right after `position`, then it takes amortized constant time. Otherwise it takes `0 (log N)` time. The return value points to the inserted `x`.

```
template <class InputIterator>
void
insert(InputIterator first, InputIterator last);
```

Inserts copies of the elements in the range `[first, last]`.

```
key_compare
key_comp() const;
```

Returns the comparison function object for the set.

```
iterator
lower_bound(const key_type& x) const;
```

Returns an `iterator` that points to the first element that is greater than or equal to `x`. If there is no such element, the iterator points to the past-the-end value.

```
size_type
max_size() const;
```

Returns the size of the largest possible set.

```
size_type
size() const;
```

Returns the number of elements.

©Copyright 1999 Rogue Wave Software, Inc.

```
void
swap(set<Key, Compare, Allocator>& x);
```

Exchanges self with `x`.

```
iterator
upper_bound(const key_type& x) const
```

Returns an iterator that points to the first element that is greater than or equal to `x`. If there is no such element, the iterator points to the past-the-end value.

```
value_compare
value_comp() const;
```

Returns the set's comparison object. This is identical to the function `key_comp()`.

Non-member Operators

```
template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
const set<Key, Compare, Allocator>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class Key, class Compare, class Allocator>
bool operator!=(const set<Key, Compare, Allocator>& x,
const set<Key, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class Key, class Compare, class Allocator>
bool operator<(const set <Key, Compare, Allocator>& x,
const set <Key, Compare, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

```
template <class Key, class Compare, class Allocator>
bool operator>(const set <Key, Compare, Allocator>& x,
const set <Key, Compare, Allocator>& y);
```

Returns `y < x`.

```
template <class Key, class Compare, class Allocator>
bool operator<=(const set <Key, Compare, Allocator>& x,
const set <Key, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class Key, class Compare, class Allocator>
bool operator>=(const set <Key, Compare, Allocator>& x,
const set <Key, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class Key, class Compare, class Allocator>
void swap(set <Key, Compare, Allocator>& a,
set <Key, Compare, Allocator>& b);
```

Swaps the contents of `a` and `b`.

Example

```
//
// setex.cpp
//
#include <set>
#include <iostream>
 using namespace std;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
typedef set<double, less<double>, allocator<double> >
        set_type;
ostream& operator<<(ostream& out, const set_type& s)
 {
copy(s.begin(), s.end(),
     ostream_iterator<set_type::value_type,char>(cout," "));
return out;
 }

int main(void)
 {
// create a set of doubles
   set_type   sd;
   int         i;

for(i = 0; i < 10; ++i) {
// insert values
sd.insert(i);
   }

// print out the set
cout << sd << endl << endl;

// now let's erase half of the elements in the set
int half = sd.size() >> 1;
set_type::iterator sdi = sd.begin();
advance(sdi,half);
sd.erase(sd.begin(),sdi);
// print it out again
cout << sd << endl << endl;

// Make another set and an empty result set
set_type sd2, sdResult;
for (i = 1; i < 9; i++)
  sd2.insert(i+5);
cout << sd2 << endl;
// Try a couple of set algorithms
set_union(sd.begin(),sd.end(),sd2.begin(),sd2.end(),
          inserter(sdResult,sdResult.begin()));
cout << "Union:" << endl << sdResult << endl;
sdResult.erase(sdResult.begin(),sdResult.end());
set_intersection(sd.begin(),sd.end(),
                 sd2.begin(),sd2.end(),
                 inserter(sdResult,sdResult.begin()));
cout << "Intersection:" << endl << sdResult << endl;
return 0;
 }
```

Program Output

```
0 1 2 3 4 5 6 7 8 9

5 6 7 8 9

6 7 8 9 10 11 12 13
Union:
5 6 7 8 9 10 11 12 13
Intersection:
6 7 8 9
```

Warnings

Member function templates are used in all containers included in the Standard Template Library. An example of this feature is the constructor for set `<Key, Compare, Allocator>` that takes two templatized iterators:

```
template <class InputIterator>
set (InputIterator, InputIterator,
const Compare& = Compare(),
const Allocator& = Allocator());
```

set also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, substitute functions allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

©Copyright 1999 Rogue Wave Software, Inc.

For example, if your compiler does not support member function templates, you can construct a set in the following two ways:

```
int intarray[10];
set<int> first_set(intarray, intarray + 10);
set<int> second_set(first_set.begin(),
first_set.end());
```

but not this way:

```
set<long> long_set(first_set.begin(),
first_set.end());
```

since the `long_set` and `first_set` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you need to write:

```
set<int, less<int>, allocator<int> >
```

instead of:

```
set<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Bidirectional Iterators***, ***Containers***, ***lexicographical_compare***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

set_difference

`Algorithm`

Summary

A basic set operation for constructing a sorted difference.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
     set_difference (InputIterator1 first1,
                     InputIterator1 last1,
                     InputIterator2 first2,
                     InputIterator2 last2,
                     OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
     set_difference (InputIterator1 first1,
                     InputIterator1 last1,
                     InputIterator2 first2,
                     InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

Description

The ***set_difference*** algorithm constructs a sorted difference that includes copies of the elements that are present in the range `[first1, last1)` but are not present in the range `[first2, last2)`. It returns the end of the constructed range.

As an example, assume we have the following two sets:

```
1 2 3 4 5
```

and

```
3 4 5 6 7
```

The result of applying ***set_difference*** is the set:

```
1 2
```

©Copyright 1999 Rogue Wave Software, Inc.

The result of ***set_difference*** is undefined if the result range overlaps with either of the original ranges.

set_difference assumes that the ranges are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Use the ***set_symetric_difference*** algorithm to return a result that contains all elements that are not in common between the two sets.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// set_diff.cpp
//
#include <algorithm>
#include <set>
#include <iostream>
using namespace std;
int main()
 {
//Initialize some sets
int a1[10] = {1,2,3,4,5,6,7,8,9,10};
int a2[6]  = {2,4,6,8,10,12};
set<int, less<int> > all(a1+0, a1+10), even(a2+0, a2+6),
                     odd;

//Create an insert_iterator for odd
insert_iterator<set<int, less<int> > >
                odd_ins(odd, odd.begin());

//Demonstrate set_difference
cout << "The result of:" << endl << "{";
copy(all.begin(),all.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} - {";
copy(even.begin(),even.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} =" << endl << "{";
set_difference(all.begin(), all.end(),
               even.begin(), even.end(), odd_ins);
copy(odd.begin(),odd.end(),
     ostream_iterator<int,char>(cout," "));
cout << "}" << endl << endl;
return 0;
 }
```

Program Output

```
The result of:
{1 2 3 4 5 6 7 8 9 10 } - {2 4 6 8 10 12 } =
{1 3 5 7 9 }
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you need to write:

```
set<int, less<int> allocator<int> >
```

instead of:

```
set<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

©Copyright 1999 Rogue Wave Software, Inc.

includes, *set*, *set_union*,*set_intersection*, *set_symmetric_difference*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

set_intersection

`Algorithm`

Summary

A basic set operation for constructing a sorted intersection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
    set_intersection (InputIterator1 first1,
                      InputIterator1 last1,
                      InputIterator2 first2,
                      InputIterator last2,
                      OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
    set_intersection (InputIterator1 first1,
                      InputIterator1 last1,
                      InputIterator2 first2,
                      InputIterator2 last2,
                      OutputIterator result, Compare comp);
```

Description

The ***set_intersection*** algorithm constructs a sorted intersection of elements from the two ranges. It returns the end of the constructed range. When it finds an element present in both ranges, ***set_intersection*** always copies the element from the first range into `result`. This means that the result of ***set_intersection*** is guaranteed to be stable. The result of ***set_intersection*** is undefined if the result range overlaps with either of the original ranges.

set_intersection assumes that the ranges are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

©Copyright 1999 Rogue Wave Software, Inc.

Example

```
//
// set_intr.cpp
//
#include <algorithm>
#include <set>
#include <iostream>
using namespace std;
int main()
 {

//Initialize some sets
int a1[10] = {1,3,5,7,9,11};
int a3[4]  = {3,5,7,8};
set<int, less<int> > odd(a1+0, a1+6),
    result, small(a3+0,a3+4);
//Create an insert_iterator for result
insert_iterator<set<int, less<int> > >
    res_ins(result, result.begin());
//Demonstrate set_intersection
cout << "The result of:" << endl << "{";
copy(small.begin(),small.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} intersection {";
copy(odd.begin(),odd.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} =" << endl << "{";
set_intersection(small.begin(), small.end(),
                 odd.begin(), odd.end(), res_ins);
copy(result.begin(),result.end(),
     ostream_iterator<int,char>(cout," "));
cout << "}" << endl << endl;
return 0;
 }
```

Program Output

```
The result of:
{3 5 7 8 } intersection {1 3 5 7 9 11 } =
{3 5 7 }
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you need to write:

```
set<int, less<int> allocator<int> >
```

instead of:

```
set<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

includes, *set*, *set_union*, *set_difference*, *set_symmetric_difference*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

set_symmetric_difference

`Algorithm`

Summary

A basic set operation for constructing a sorted symmetric difference.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
    set_symmetric_difference (InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2,
                              InputIterator2 last2,
                              OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
    set_symmetric_difference (InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2,
                              InputIterator2 last2,
                              OutputIterator result,
                              Compare comp);
```

Description

set_symmetric_difference constructs a sorted symmetric difference of the elements from the two ranges. This means that the constructed range includes copies of the elements that are present in the range `[first1, last1)` (but not present in the range `[first2, last2)`) *and* copies of the elements that are present in the range `[first2, last2)` (but not in the range `[first1, last1)`). It returns the end of the constructed range.

For example, suppose we have two sets:

```
1 2 3 4 5
```

and

```
3 4 5 6 7
```

©Copyright 1999 Rogue Wave Software, Inc.

The ***set_symmetric_difference*** of these two sets is:

```
1 2 6 7
```

The result of ***set_symmetric_difference*** is undefined if the result range overlaps with either of the original ranges.

set_symmetric_difference assumes that the ranges are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Use the ***set_symmetric_difference*** algorithm to return a result that includes elements that are present in the first set and not in the second.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// set_s_di.cpp
//
#include<algorithm>
#include<set>
#include <istream.h>
using namespace std;

int main()
 {

//Initialize some sets
int a1[] = {1,3,5,7,9,11};
int a3[] = {3,5,7,8};
set<int, less<int> > odd(a1+0,a1+6), result,
    small(a3+0,a3+4);
//Create an insert_iterator for result
insert_iterator<set<int, less<int> > >
    res_ins(result, result.begin());
//Demonstrate set_symmetric_difference
cout << "The symmetric difference of:" << endl << "{";
copy(small.begin(),small.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} with {";
copy(odd.begin(),odd.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} =" << endl << "{";
set_symmetric_difference(small.begin(), small.end(),
                         odd.begin(), odd.end(), res_ins);
copy(result.begin(),result.end(),
     ostream_iterator<int,char>(cout," "));
cout << "}" << endl << endl;
return 0;
 }
```

Program Output

```
The symmetric difference of:
{3 5 7 8 } with {1 3 5 7 9 11 } =
{1 8 9 11 }
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you need to write:

```
set<int, less<int>, allocator<int> >
```

instead of:

```
set<int>
```

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

includes, ***set***, ***set_union***,***set_intersection***, ***set_difference***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

set_union

`Algorithm`

Summary

A basic set operation for constructing a sorted union.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
    set_union (InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
    set_union (InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result, Compare comp);
```

Description

The ***set_union*** algorithm constructs a sorted union of the elements from the two ranges. It returns the end of the constructed range. ***set_union*** is stable, which means that if an element is present in both ranges, the one from the first range is copied. The result of ***set_union*** is undefined if the result range overlaps with either of the original ranges. Note that ***set_union*** does not merge the two sorted sequences. If an element is present in both sequences, only the element from the first sequence is copied to `result`. (Use the ***merge*** algorithm to create an ordered merge of two sorted sequences that contains all the elements from both sequences.)

set_union assumes that the sequences are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// set_unin.cpp
//
#include <algorithm>
#include <set>
#include <iostream>
using namespace std;

int main()
 {

//Initialize some sets
int a2[6]  = {2,4,6,8,10,12};
int a3[4]  = {3,5,7,8};
set<int, less<int> >  even(a2+0, a2+6),
    result, small(a3+0,a3+4);
//Create an insert_iterator for result
insert_iterator<set<int, less<int> > >
    res_ins(result, result.begin());
//Demonstrate set_union
cout << "The result of:" << endl << "{";
copy(small.begin(),small.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} union {";
copy(even.begin(),even.end(),
     ostream_iterator<int,char>(cout," "));
cout << "} =" << endl << "{";
set_union(small.begin(), small.end(),
          even.begin(), even.end(), res_ins);
copy(result.begin(),result.end(),
     ostream_iterator<int,char>(cout," "));
cout << "}" << endl << endl;
return 0;
 }
```

Program Output

```
The result of:
{3 5 7 8 } union {2 4 6 8 10 12 } =
{2 3 4 5 6 7 8 10 12 }
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Compare` template argument and the `Allocator` template argument. For instance, you need to write:

```
set<int, less<int>, allocator<int> >
```

instead of:

```
set<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

includes, *set*, *set_intersection*, *set_difference*, *set_symmetric_difference*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

slice

`Valarray helpers`

Summary

A numeric array class for representing a BLAS-like ***slice*** from an array.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

size()
start()
stride()

Synopsis

```
#include <valarray>
class slice ;
```

Description

slice allows you to represent a BLAS-like slice from an array. A BLAS slice contains a starting index, a length, and a stride. The index indicates the first element in the slice, the length determines the number of elements, and the stride indicates the interval between elements in the original array. For instance, the ***slice*** (1,3,2) applied to the array (1,2,3,4,5,6,7) produces the array (2,4,6).

When applied to a ***valarray*** using the ***slice*** subscript operator (see ***valarray***) a ***slice*** produces a ***slice_array***. The ***slice_array*** gives a view into the original ***valarray*** that is tailored to match parameters of the ***slice***. The elements in a ***slice_array*** are references to the elements in the original array. This means you need to explicitly copy the ***slice_array*** into another ***valarray*** in order to have a distinct array.

Interface

```
class slice {
public:
  // constructors
  slice();
  slice(size_t, size_t, size_t);

  // Accessors
  size_t start() const;
  size_t size() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  size_t stride() const;
};
```

Constructors

slice();

Creates a ***slice*** specifying no elements. This constructor is only intended to allow the creation of arrays of slices.

slice(size_t start, size_t length, size_t stride);

Creates a ***slice*** with starting index, length, and stride as indicated by the arguments.

slice(const slice&)

Creates a ***slice*** with starting index, length, and stride as indicated by the slice argument.

Accessors

size_t **start**();

Returns the starting index of the slice.

size_t **size**();

Returns the length of the slice.

size_t **stride**();

Returns the stride of the slice.

Example

```
//
// slice.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
using namespace std;
int main(void)
{
  int ibuf[10] = {0,1,2,3,4,5,6,7,8,9};

  // create a valarray of ints
  valarray<int>         vi(ibuf,10);
  // print it out
  cout << vi << endl;

  // print out a slice
  cout << valarray<int>(vi[slice(1,3,2)]) << endl;
  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,9]
[1,3,5]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, ***slice_array***, ***gslice***, ***gslice_array***, ***mask_array***, ***indirect_array***

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

slice_array

`Valarray helpers`

Summary

A numeric array class for representing a BLAS-like ***slice*** from a ***valarray***.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator%=()
operator&=() operator-=()
operator>>=() operator/=()
operator<<=() operator=()
operator*=() operator^=()
operator+=()

Synopsis

```
#include <valarray>
template <class T>
class slice_array ;
```

Description

slice_array<T> gives a slice view into a ***valarray***. ***Slice_array***s are only produced by applying the slice subscript operator to a ***valarray***. The elements in a ***slice_array*** are references to selected elements in the ***valarray*** (so changing an element in the ***slice_array*** really changes the corresponding element in the ***valarray***). A ***slice_array*** does not itself hold any distinct elements. The template cannot be instantiated directly since all its constructors are private. However, you can easily copy a ***slice_array*** to a ***valarray*** using either the ***valarray*** copy constructor or the assignment operator. Reference semantics are lost at that point.

Interface

```
template <class T> class slice_array {
public:

  // types
  typedef T value_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // destructor
  ~slice_array();

  // public assignment
  void operator= (const valarray<T>& array) const;
  // computed assignment
  void operator*= (const valarray<T>& array) const;
  void operator/= (const valarray<T>& array) const;
  void operator%= (const valarray<T>& array) const;
  void operator+= (const valarray<T>& array) const;
  void operator-= (const valarray<T>& array) const;
  void operator^= (const valarray<T>& array) const;
  void operator&= (const valarray<T>& array) const;
  void operator|= (const valarray<T>& array) const;
  void operator<<= (const valarray<T>& array) const;
  void operator>>= (const valarray<T>& array) const;

  // other
  void operator= (const T&);

private:
  // constructors
  slice_array();
  slice_array(const slice_array<T>&);
  // operator =
  slice_array<T>& operator= (const slice_array<T>& array);
};
```

Constructors

```
slice_array();
slice_array(const slice_array&);
```

All ***slice_array*** constructors are private and cannot be called directly. This prevents copy construction of ***slice_array***s.

Assignment Operators

```
void operator=(const valarray<T>& x) const;
```

Assigns values from `x` to the selected elements of the ***valarray*** that self refers to. Remember that a ***slice_array*** never holds any elements itself, it simply refers to selected elements in the ***valarray*** used to generate it.

```
slice_array<T>&
operator=(const slice-_array<T>& x);
```

Private assignment operator. Cannot be called directly, thus preventing assignment between ***slice_array***s.

Computed Assignment Operators

```
void operator*=(const valarray<T>& val) const;
void operator/=(const valarray<T>& val) const;
void operator%=(const valarray<T>& val) const;
void operator+=(const valarray<T>& val) const;
void operator-=(const valarray<T>& val) const;
void operator^=(const valarray<T>& val) const;
void operator&=(const valarray<T>& val) const;
void operator|=(const valarray<T>& val) const;
void operator<<=(const valarray<T>& val) const;
void operator>>=(const valarray<T>& val) const;
```

Applies the indicated operation using elements from `val` to the selected elements of the ***valarray*** that self refers to. Remember that a ***slice_array*** never holds any elements itself; it simply refers to selected elements in the ***valarray*** used to generate it.

Member Functions

```
void operator= (const T& x);
```

©Copyright 1999 Rogue Wave Software, Inc.

Assigns x to the selected elements of the *valarray* that self refers to.

Example

```
//
// slice_array.cpp
//
#include "valarray.h"  // Contains a valarray
                       // stream inserter
using namespace std;

int main(void)
{
  int ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
  int ibuf2[5] = {1,3,5,7,9};

  // create a valarray of ints
  valarray<int>         vi(ibuf,10);
  valarray<int>         vi2(ibuf2,5);

  // print it out
  cout << vi << endl << vi2 << endl;

  // Get a slice and assign that slice to another array
  slice_array<int> sl = vi[slice(1,5,2)];
  valarray<int> vi3 = sl;

  // print out the slice
  cout << vi3 << endl;

  // Add slice from vi2 to slice of vi1
  sl += vi2;

  // print out vi1 again
  cout << vi << endl;

  return 0;
}
```

Program Output

```
[0,1,2,3,4,5,6,7,8,9]
[1,3,5,7,9]
[1,3,5,7,9]
[0,2,2,6,4,10,6,14,8,18]
```

Warnings

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

valarray, *slice*, *gslice*, *gslice_array*, *mask_array*, *indirect_array*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

smanip, smanip_fill

Summary

Helper classes used to implement parameterized manipulators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator>>()
operator<<()
resetiosflag()
setbase()
setfill()
setiosflag()
setprecision()
setw()

Synopsis

```
#include <iomanip>
template<class T> class smanip;
template<class T, class traits> class smanip_fill;
```

Description

The template classes ***smanip*** and ***smanip_fill*** are helper classes used to implement parameterized manipulators. The class ***smanip*** is used as the return type for manipulators that do not need to carry information about the character type of the stream they are applied to. This is the case for `resetiosflags`, `setiosflags`, `setbase`, `setprecision`, and `setw`. The class ***smanip_fill*** is used as the return type for manipulators that do need to carry information about the character type of the streamto which they are applied. This is the case for `setfill`.

smanip_fill is not described in the C++ standard, and is included as an extension.

Interface

```
template<class T>
class smanip {

 public:
  smanip(ios_base& (*pf) (ios_base&, T), T manarg);
};
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class T, class traits>
class smanip_fill {

 public:
  smanip_fill(basic_ios<T, traits>& (*pf)
              (basic_ios<T, traits>&, T), T manarg);
};

// parameterized manipulators

smanip<ios_base::fmtflags>
   resetiosflag(ios_base::fmtflags mask);
smanip<ios_base::fmtflags>
   setiosflag(ios_base::fmtflags mask);
smanip<int>                setbase(int base);
smanip<int>                setprecision(int n);
smanip<int>                setw(int n);

template <class charT>
smanip_fill<charT, char_traits<charT> > setfill(charT c);

// overloaded extractors

template <class charT, class traits, class T>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is,
   const smanip<T>& a);

template <class charT, class traits>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is,
   const smanip_fill<charT,char_traits<charT> >& a);

// overloaded inserters

template <class charT, class traits, class T>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>& is,
   const smanip<T>& a);

template <class charT, class traits>
basic_ostream<charT,traits>&
operator>>(basic_ostream<charT,traits>& is,
const smanip_fill<charT,char_traits<charT> >& a);
```

Class smanip Constructor

```
smanip(ios_base& (*pf) (ios_base&, T), T manarg);
```

Constructs an object of class `smanip` that stores a function pointer `pf` that is called with argument `manarg`, in order to perform the manipulator task. The call to `pf` is performed in the inserter or extractor overloaded on type `smanip`.

Class smanip_fill Constructor

```
smanip_fill(basic_ios<T, traits>& (*pf) (basic_ios<T, traits>&, T), T manarg);
```

Constructs an object of class `smanip_fill` that stores a function pointer `pf`, that is called with argument `manarg`, in order to perform the manipulator task. The call to `pf` is performed in the inserter or extractor overloaded on type `smanip_fill`.

Manipulators

```
smanip<ios_base::fmtflags>
resetiosflag(ios_base::fmtflags mask);
```

Resets the `ios_base::fmtflags` designated by `mask` in the stream to which it is applied.

```
smanip<int>
setbase(int base);
```

Sets the base for the output or input of integer values in the stream to which it is applied. The valid values for `mask` are 8, 10, 16.

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class charT>
smanip_fill<charT, char_traits<charT> >
setfill(charT c);
```

Sets the fill character in the stream to which it is applied.

```
smanip<ios_base::fmtflags>
setiosflag(ios_base::fmtflags mask);
```

Sets the `ios_base::fmtflags` designated by `mask` in the stream to which it is applied.

```
smanip<int>
setprecision(int n);
```

Sets the precision for the output of floating point values in the stream to which it is applied.

```
smanip<int>
setw(int n);
```

Set the field width in the stream to which it is applied.

Extractors

```
template <class charT, class traits, class T>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is, const smanip<T>& a);
```

Applies the function stored in the parameter of type `smanip<T>`, on the stream `is`.

```
template <class charT, class traits>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is,
         const smanip_fill<charT,char_traits<charT> >& a);
```

Applies the function stored in the parameter of type `smanip_fill<charT, char_traits<charT> >` on the stream `is`.

Inserters

```
template <class charT, class traits, class T>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>& os,
           const smanip<T>& a);
```

Applies the function stored in the parameter of type `smanip<T>` on the stream `os`.

```
template <class charT, class traits>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>& os,
           const smanip_fill<charT,char_traits<charT> >& a);
```

Applies the function stored in the parameter of type `smanip_fill<charT, char_traits<charT> >` on the stream `os`.

See Also

ios_base(3C++), *basic_ios*(3C++), *basic_istream*(3C++), *basic_ostream*(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.6.3

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

sort

`Algorithm`

Summary

A templatized algorithm for sorting collections of entities.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
void sort (RandomAccessIterator first,
           RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last, Compare comp);
```

Description

The ***sort*** algorithm sorts the elements in the range `[first, last)` using either the less than (`<`) operator or the comparison operator `comp`. If the worst case behavior is important, ***stable_sort*** or ***partial_sort*** should be used.

Complexity

On average**, *sort*** performs approximately `N(logN)` comparisons, where `N` equals `last - first`.

Example

```
//
// sort.cpp
//
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

struct associate
 {
int num;
char chr;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
associate(int n, char c) : num(n), chr(c) {};
associate() : num(0), chr(`\0'){};
 };

bool operator<(const associate &x, const associate &y)
 {
return x.num < y.num;
 }


ostream& operator<<(ostream &s, const associate &x)
 {
return s << "<" << x.num << ";" << x.chr << ">";
 }

int main ()
 {
vector<associate>::iterator i, j, k;
associate arr[20] =
  {associate(-4, ` `), associate(16, ` `),
   associate(17, ` `), associate(-3, `s'),
   associate(14, ` `), associate(-6, ` `),
   associate(-1, ` `), associate(-3, `t'),
   associate(23, ` `), associate(-3, `a'),
   associate(-2, ` `), associate(-7, ` `),
   associate(-3, `b'), associate(-8, ` `),
   associate(11, ` `), associate(-3, `l'),
   associate(15, ` `), associate(-5, ` `),
   associate(-3, `e'), associate(15, ` `)};

// Set up vectors
vector<associate> v(arr, arr+20), v1((size_t)20),
                  v2((size_t)20);
// Copy original vector to vectors #1 and #2
copy(v.begin(), v.end(), v1.begin());
copy(v.begin(), v.end(), v2.begin());

// Sort vector #1
sort(v1.begin(), v1.end());

// Stable sort vector #2
stable_sort(v2.begin(), v2.end());

// Display the results
cout << "Original    sort      stable_sort" << endl;
for(i = v.begin(), j = v1.begin(), k = v2.begin();
  i != v.end(); i++, j++, k++)
cout << *i << "     " << *j << "     " << *k << endl;
return 0;
 }
```

Program Output

```
Original    sort      stable_sort
<-4; >     <-8; >     <-8; >
<16; >     <-7; >     <-7; >
<17; >     <-6; >     <-6; >
<-3;s>     <-5; >     <-5; >
<14; >     <-4; >     <-4; >
<-6; >     <-3;e>     <-3;s>
<-1; >     <-3;s>     <-3;t>
<-3;t>     <-3;l>     <-3;a>
<23; >     <-3;t>     <-3;b>
<-3;a>     <-3;b>     <-3;l>
<-2; >     <-3;a>     <-3;e>
<-7; >     <-2; >     <-2; >
<-3;b>     <-1; >     <-1; >
<-8; >     <11; >     <11; >
<11; >     <14; >     <14; >
<-3;l>     <15; >     <15; >
<15; >     <15; >     <15; >
<-5; >     <16; >     <16; >
<-3;e>     <17; >     <17; >
<15; >     <23; >     <23; >
```

Warnings

©Copyright 1999 Rogue Wave Software, Inc.

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

stable_sort, ***partial_sort***, ***partial_sort_copy***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

sort_heap

`Algorithm`

Summary

Converts a heap into a sorted collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
void
sort_heap(RandomAccessIterator first,
RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
void
sort_heap(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by `pop_heap()` or a new element may be added by `push_heap()`, in `O(logN)` time.

These properties make heaps useful as priority queues.

The ***sort_heap*** algorithm converts a heap into a sorted collection over the range `[first, last)` using either the default operator (`<`) or the comparison function supplied with the algorithm. Note that ***sort_heap*** is not stable (in other words, the elements may not be in the same relative order after ***sort_heap*** is applied).

Complexity

sort_heap performs at most `NlogN` comparisons, where `N` is equal to `last - first`.

Example

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// heap_ops.cpp
//
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main(void)
{
int d1[4] = {1,2,3,4};
int d2[4] = {1,3,2,4};

// Set up two vectors
vector<int> v1(d1+0,d1 + 4), v2(d2+0,d2 + 4);
// Make heaps
make_heap(v1.begin(),v1.end());
make_heap(v2.begin(),v2.end(),less<int>());
// v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
// Note that x, y and z represent the remaining
// values in the container (other than 4).
// The definition of the heap and heap operations
// does not require any particular ordering
// of these values.
// Copy both vectors to cout
ostream_iterator<int,char> out(cout," ");
copy(v1.begin(),v1.end(),out);
cout << endl;
copy(v2.begin(),v2.end(),out);
cout << endl;

// Now let's pop
pop_heap(v1.begin(),v1.end());
pop_heap(v2.begin(),v2.end(),less<int>());
// v1 = (3,x,y,4) and v2 = (3,x,y,4)

// Copy both vectors to cout
copy(v1.begin(),v1.end(),out);
cout << endl;
copy(v2.begin(),v2.end(),out);
cout << endl;

// And push
push_heap(v1.begin(),v1.end());
push_heap(v2.begin(),v2.end(),less<int>());
// v1 = (4,x,y,z) and v2 = (4,x,y,z)

// Copy both vectors to cout
copy(v1.begin(),v1.end(),out);
cout << endl;
copy(v2.begin(),v2.end(),out);
cout << endl;

// Now sort those heaps
sort_heap(v1.begin(),v1.end());
sort_heap(v2.begin(),v2.end(),less<int>());
// v1 = v2 = (1,2,3,4)
// Copy both vectors to cout
copy(v1.begin(),v1.end(),out);
cout << endl;
copy(v2.begin(),v2.end(),out);
cout << endl;

return 0;
}
```

Program Output

```
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

©Copyright 1999 Rogue Wave Software, Inc.

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

make_heap, ***pop_heap***, ***push_heap***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

stable_partition

`Algorithm`

Summary

Places all of the entities that satisfy the given predicate before all of the entities that do not, while maintaining the relative order of elements in each group.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition (BidirectionalIterator first,
                      BidirectionalIterator last,
                      Predicate pred);
```

Description

For the range `[first, last)`, the ***stable_partition*** algorithm places all elements that satisfy `pred` before all elements that do not satisfy it. It returns an iterator `i` that is one past the end of the group of elements that satisfies `pred`. In other words ***stable_partition*** returns `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`, `pred(*j) == false`. The relative order of the elements in both groups is preserved.

The ***partition*** algorithm can be used when it is not necessary to maintain the relative order of elements within the groups that do and do not match the predicate.

Complexity

The ***stable_partition*** algorithm does at most `(last - first) * log(last - first)` swaps and applies the predicate exactly `last - first` times.

Example

```
//
// prtition.cpp
//
#include <functional>
#include <deque>
#include <algorithm>
#include <iostream>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
using namespace std;

//Create a new predicate from unary_function
template<class Arg>
class is_even : public unary_function<Arg, bool>
{
public:
bool operator()(const Arg& arg1)
{
return (arg1 % 2) == 0;
   }
 };


int main()
{
//Initialize a deque with an array of ints
int init[10] = {1,2,3,4,5,6,7,8,9,10};
deque<int> d(init, init+10);

//Print out the original values
cout << "Unpartitioned values: " << endl << "     ";
copy(d.begin(),d.end(),
     ostream_iterator<int,char>(cout," "));
cout << endl << endl;

//Partition the deque according to even/oddness
stable_partition(d.begin(), d.end(), is_even<int>());

//Output result of partition
cout << "Partitioned values: " << endl << "     ";
copy(d.begin(),d.end(),
     ostream_iterator<int,char>(cout," "));
return 0;
 }
```

Program Output

```
Unpartitioned values:           1 2 3 4 5 6 7 8 9 10
Partitioned values:             10 2 8 4 6 5 7 3 9 1
Stable partitioned values:      2 4 6 8 10 1 3 5 7 9
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

partition

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

stable_sort

`Algorithm`

Summary

A templatized algorithm for sorting collections of entities.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class RandomAccessIterator>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last, Compare comp);
```

Description

The ***stable_sort*** algorithm sorts the elements in the range `[first, last)`. The first version of the algorithm uses less than (`<`) as the comparison operator for the sort. The second version uses the comparison function `comp`.

The ***stable_sort*** algorithm is considered stable because the relative order of the equal elements is preserved.

Complexity

stable_sort does at most `N(logN)2` comparisons, where `N` equals `last - first`. If enough extra memory is available, it does at most `NlogN`.

Example

```
//
// sort.cpp
//
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

struct associate
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 {
int num;
char chr;

associate(int n, char c) : num(n), chr(c) {};
associate() : num(0), chr(`\0'){};
 };


bool operator<(const associate &x, const associate &y)
 {
return x.num < y.num;
 }


ostream& operator<<(ostream &s, const associate &x)
 {
return s << "<" << x.num << ";" << x.chr << ">";
 }

int main ()
 {
vector<associate>::iterator i, j, k;
associate arr[20] =
  {associate(-4, ` `), associate(16, ` `),
   associate(17, ` `), associate(-3, `s'),
   associate(14, ` `), associate(-6, ` `),
   associate(-1, ` `), associate(-3, `t'),
   associate(23, ` `), associate(-3, `a'),
   associate(-2, ` `), associate(-7, ` `),
   associate(-3, `b'), associate(-8, ` `),
   associate(11, ` `), associate(-3, `l'),
   associate(15, ` `), associate(-5, ` `),
   associate(-3, `e'), associate(15, ` `)};

// Set up vectors
vector<associate> v(arr, arr+20), v1((size_t)20),
                  v2((size_t)20);

// Copy original vector to vectors #1 and #2
copy(v.begin(), v.end(), v1.begin());
copy(v.begin(), v.end(), v2.begin());

// Sort vector #1
sort(v1.begin(), v1.end());

// Stable sort vector #2
stable_sort(v2.begin(), v2.end());

// Display the results
cout << "Original    sort      stable_sort" << endl;
for(i = v.begin(), j = v1.begin(), k = v2.begin();
  i != v.end(); i++, j++, k++)
  cout << *i << "     " << *j << "     " << *k << endl;


return 0;
 }
```

Program Output

```
Original    sort      stable_sort
<-4; >     <-8; >     <-8; >
<16; >     <-7; >     <-7; >
<17; >     <-6; >     <-6; >
<-3;s>     <-5; >     <-5; >
<14; >     <-4; >     <-4; >
<-6; >     <-3;e>     <-3;s>
<-1; >     <-3;s>     <-3;t>
<-3;t>     <-3;l>     <-3;a>
<23; >     <-3;t>     <-3;b>
<-3;a>     <-3;b>     <-3;l>
<-2; >     <-3;a>     <-3;e>
<-7; >     <-2; >     <-2; >
<-3;b>     <-1; >     <-1; >
<-8; >     <11; >     <11; >
<11; >     <14; >     <14; >
```

©Copyright 1999 Rogue Wave Software, Inc.

```
<-3;l>     <15; >     <15; >
<15; >     <15; >     <15; >
<-5; >     <16; >     <16; >
<-3;e>     <17; >     <17; >
<15; >     <23; >     <23; >
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator>
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

sort,*partial_sort*, *partial_sort_copy*

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

stack

`Container Adapter`

Summary

A container adapter that behaves like a stack (last in, first out).

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

empty()
operator!=()
operator>()
operator>=()
operator<()
operator<=()
operator==()
pop()
push()
size()
top()

Synopsis

```
#include <stack>
template <class T, class Container = deque<T> >
class stack ;
```

Description

The ***stack*** container adapter causes a container to behave like a "last in, first out" (LIFO) stack. The last item that was put ("pushed") onto the stack is the first item removed ("popped" off). The stack can adapt to any container that includes the operations `back()`, `push_back()`, and `pop_back()`. In particular, ***deque***, ***list***, and ***vector*** can be used.

Interface

```
template <class T, class Container = deque<T> >
class stack {
public:
// typedefs
typedef typename Container::value_type value_type;
typedef typename Container::size_type size_type;
typedef Container container_type;
// Construct
explicit stack (const Container& = Container());
// Accessors
bool empty () const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
size_type size () const;
value_type& top ();
const value_type& top () const;
void push (const value_type&);
void pop ();
};

// Non-member Operators
template <class T, class Container>
bool operator== (const stack<T, Container>&,
const stack<T, Container>&);
template <class T, class Container>
bool operator!= (const stack<T, Container>&,
const stack<T, Container>&);
template <class T, class Container>
bool operator< (const stack<T, Container>&,
const stack<T, Container>&);
template <class T, class Container>
bool operator> (const stack<T, Container>&,
const stack<T, Container>&);
template <class T, class Container>
bool operator<= (const stack<T, Container>&,
const stack<T, Container>&);
template <class T, class Container>
bool operator>= (const stack<T, Container>&,
const stack<T, Container>&);
```

Constructors

```
explicit
stack(const Container& = Container());
```

Constructs an empty stack. The stack uses the allocator `alloc` for all storage management.

Member Functions

```
bool
empty() const;
```

Returns `true` if the stack is empty, otherwise `false`.

```
void
pop();
```

Removes the item at the top of the stack.

```
void
push(const value_type& x);
```

Pushes `x` onto the stack.

```
size_type
size() const;
```

Returns the number of elements on the stack.

```
value_type&
top();
```

Returns a reference to the item at the top of the stack. This is the last item pushed onto the stack unless `pop()` has been called since then.

```
const value_type&
top() const;
```

Returns a constant reference to the item at the top of the stack as a `const value_type`.

Non-member Operators

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T, class Container>
bool operator==(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T, class Container>
bool operator!=(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `!(x==y)`.

```
template <class T, class Container>
bool operator<(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `true` if the stack defined by the elements contained in `x` is lexicographically less than the stack defined by the elements of `y`.

```
template <class T, class Container>
bool operator>(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `y < x`.

```
template <class T, class Container>
bool operator<=(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `!(y < x)`.

```
template <class T, class Container>
bool operator>=(const stack<T, Container>& x,
const stack<T, Container>& y);
```

Returns `!(x < y)`.

Example

```
//
// stack.cpp
//
#include <stack>
#include <vector>
#include <deque>
#include <string>
#include <iostream>
using namespace std;

int main(void)
 {
// Make a stack using a vector container
stack<int,vector<int> > s;
// Push a couple of values on the stack
s.push(1);
s.push(2);
cout << s.top() << endl;

// Now pop them off
s.pop();
cout << s.top() << endl;
s.pop();

// Make a stack of strings using a deque
stack<string,deque<string> > ss;
// Push a bunch of strings on then pop them off
int i;
for (i = 0; i < 10; i++)
   {
    ss.push(string(i+1,'a'));
    cout << ss.top() << endl;
   }
for (i = 0; i < 10; i++)
   {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    cout << ss.top() << endl;
    ss.pop();
   }

return 0;
 }
```

Program Output

```
2
1
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
```

Warnings

If your compiler does not support template parameter defaults, you are required to supply a template parameter for `Container`. For example:

You would not be able to write,

```
stack<int> var;
```

Instead, you would have to write,

```
stack<int, deque<int> > var;
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

allocator, ***Containers***, ***deque***, ***list***, ***vector***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Stream Iterators

`Iterators`

Summary

Stream iterators include iterator capabilities for ostreams and istreams. They allow generic algorithms to be used directly on streams.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

See the sections ***istream_iterator*** and ***ostream_iterator*** for a description of these iterators.

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

string

`String Library`

Summary

A typedef for:

```
basic_string<char, char_traits<char>, allocator<char>>
```

For more information about strings, see the entry ***basic_string***.

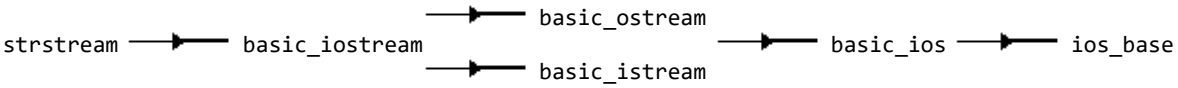
Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

strstream



Summary

Reads and writes to an array in memory.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
int_type
off_type
pos_type

traits

Member Functions

freeze()
pcount()
rdbuf()
str()

Synopsis

```
#include <strstream>
class strstream
: public basic_iostream<char>
```

Description

The class ***strstream*** reads and writes to an array in memory. It uses a private ***strstreambuf*** object to control the associated array. It inherits from ***basic_iostream<char>*** and therefore can use all the formatted and unformatted output and input functions.

This is a deprecated feature and might not be available in future versions.

Interface

©Copyright 1999 Rogue Wave Software, Inc.

```
class strstream
: public basic_iostream<char> {

 public:

  typedef char_traits<char>           traits;

  typedef char                       char_type;
  typedef typename traits::int_type  int_type;
  typedef typename traits::pos_type  pos_type;
  typedef typename traits::off_type  off_type;

  strstream();
  strstream(char *s, int n,
            ios_base::openmode =
            ios_base::out | ios_base::in);

  void freeze(int freezefl = 1);
  int pcount() const;

  virtual ~strstream();
  strstreambuf *rdbuf() const;

  char *str();

};
```

Types

`char_type`

The type `char_type` is a synonym of type `char`.

`int_type`

The type `int_type` is a synonym of type `traits::in_type`.

`off_type`

The type `off_type` is a synonym of type `traits::off_type`.

`pos_type`

The type `pos_type` is a synonym of type `traits::pos_type`.

`traits`

The type `traits` is a synonym of type `char_traits<char>`.

Constructors

`strstream``();`

Constructs an object of class `strstream`, initializing the base class `basic_iostream<char>` with the associated `strstreambuf` object. The `strstreambuf` object is initialized by calling its default constructor `strstreambuf()`.

`strstream``(char* s, int n, ios_base::openmode`
`mode = ios_base::out | ios_base::in);`

Constructs an object of class `strstream`, initializing the base class `basic_iostream<char>` with the associated `strstreambuf` object. The `strstreambuf` object is initialized by calling one of two constructors:

- If `mode & app == 0`, calls `strstreambuf(s,n,s)`
- Otherwise calls `strstreambuf(s,n,s + ::strlen(s))`

Destructors

`virtual` **`~strstream`**`();`

Destroys an object of class `strstream`.

©Copyright 1999 Rogue Wave Software, Inc.

Member Functions

```
void
freeze(int freezefl = 1);
```

If the mode is dynamic, alters the freeze status of the dynamic array object as follows:

- If `freezefl` is `false`, the function sets the freeze status to `frozen`.
- Otherwise, it clears the freeze status.

```
int
pcount() const;
```

Returns the size of the output sequence.

```
strstreambuf*
rdbuf() const;
```

Returns a pointer to the `strstreambuf` object associated with the stream.

```
char*
str();
```

Returns a pointer to the underlying array object, which may be null.

Examples

```
//
// stdlib/examples/manual/strstream.cpp
//
#include<strstream>
using namespace std;

void main ( )
{
  using namespace std;

  // create a bi-directional strstream object
  strstream inout;

  // output characters
  inout << "Das ist die rede von einem man" << endl;
  inout << "C'est l'histoire d'un home" << endl;
  inout << "This is the story of a man" << endl;

  char p[100];

  // extract the first line
  inout.getline(p,100);

  // output the first line to stdout
  cout << endl << "Deutch :" << endl;
  cout << p;

  // extract the second line
  inout.getline(p,100);

  // output the second line to stdout
  cout << endl << "Francais :" << endl;
  cout << p;

  // extract the third line
  inout.getline(p,100);

  // output the third line to stdout
  cout << endl << "English :" << endl;
  cout << p;

  // output the all content of the
  // strstream object to stdout
  cout << endl << endl << inout.str();
```

©Copyright 1999 Rogue Wave Software, Inc.

```
}
```

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***strstreambuf***(3C++), ***istrstream***(3C++), ***ostrstream***(3c++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Annex D Compatibility features Section D.6.4

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

strstreambuf

Summary

Associates either the input sequence or the output sequence with a tiny character array whose elements store arbitrary values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type	
int_type	pos_type
ios_type	traits
off_type	

Member Functions

freeze()	
overflow()	setbuf()
pbackfail()	str()
pcount()	underflow()
seekoff()	xsputn()
seekpos()	

Synopsis

```
#include <strstream>
class strstreambuf
: public basic_streambuf<char>
```

Description

The class ***strstreambuf*** is derived from ***basic_streambuf*** specialized on type `char` to associate either the input sequence or the output sequence with a tiny character array whose elements store arbitrary values.

Each object of type ***strstreambuf*** controls two character sequences:

- A character input sequence
- A character output sequence

©Copyright 1999 Rogue Wave Software, Inc.

Note: see ***basic_streambuf.***

The two sequences are related to each other, but are manipulated separately. This means that you can read and write characters at different positions in objects of type ***strstreambuf*** without any conflict (in opposition to the ***basic_filebuf*** objects).

The underlying array has several attributes:

- `allocated`, set when a dynamic array has been allocated, and hence should be freed by the destructor of the ***strstreambuf*** object.
- `constant`, set when the array has `const` elements, so the output sequence cannot be written.
- `dynamic`, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length.
- `frozen`, set when the program has requested that the array not be altered, reallocated, or freed.

This is a deprecated feature and might not be available in future versions.

Interface

```
class strstreambuf
: public basic_streambuf<char> {

 public:

  typedef char_traits<char>          traits;
  typedef basic_ios<char, traits>    ios_type;

  typedef char                       char_type;
  typedef typename traits::int_type  int_type;
  typedef typename traits::pos_type  pos_type;
  typedef typename traits::off_type  off_type;

  explicit strstreambuf(streamsize alsize = 0);
  strstreambuf(void *(*palloc)(size_t),
               void (*pfree)(void *));
  strstreambuf(char *gnext, streamsize n, char *pbeg = 0);

  strstreambuf(unsigned char *gnext, streamsize n,
               unsigned char *pbeg = 0);
  strstreambuf(signed char *gnext, streamsize n,
               signed char *pbeg = 0);

  strstreambuf(const char *gnext, streamsize n);
  strstreambuf(const unsigned char *gnext, streamsize n);
  strstreambuf(const signed char *gnext, streamsize n);

  virtual ~strstreambuf();

  void freeze(bool f = 1);
  char *str();
  int pcount() const;

 protected:

  virtual int_type overflow(int_type c = traits::eof());
  virtual int_type pbackfail(int_type c = traits::eof());
  virtual int_type underflow();

  virtual pos_type seekoff(off_type, ios_type::seekdir way,
                           ios_type::openmode which =
                           ios_type::in | ios_type::out);

  virtual pos_type seekpos(pos_type sp,
                         ios_type::openmode which =
                           ios_type::in | ios_type::out);

  virtual streambuf* setbuf(char *s, streamsize n);
  virtual streamsize xsputn(const char_type* s,
                            streamsize n);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
};
```

Types

char_type

The type char_type is a synonym of type char.

int_type

The type int_type is a synonym of type traits::in_type.

ios_type

The type ios_type is an instantiation of class basic_ios on type char.

off_type

The type off_type is a synonym of type traits::off_type.

pos_type

The type pos_type is a synonym of type traits::pos_type.

traits

The type traits is a synonym of type char_traits<char>.

Constructors

```
explicit strstreambuf(streamsize alsize = 0);
```

Constructs an object of class strstreambuf, initializing the base class with streambuf(). After initialization the strstreambuf object is in dynamic mode and its array object has a size of alsize.

```
strstreambuf(void* (*palloc)(size_t),
             void (*pfree)(void*));
```

Constructs an object of class strstreambuf, initializing the base class with streambuf(). After initialization the strstreambuf object is in dynamic mode. The function used to allocate memory is pointed to by void* (*palloc)(size_t) and the one used to free memory is pointed to by void (*pfree)(void*).

```
strstreambuf(char* gnext, streamsize n,
             char* pbeg = 0);
strstreambuf(signed char* gnext, streamsize n,
             signed char* pbeg = 0);
strstreambuf(unsigned char* gnext, streamsize n,
             unsigned char* pbeg = 0);
```

Constructs an object of class strstreambuf, initializing the base class with streambuf(). The argument gnext points to the first element of an array object whose number of elements is:

```
n, if n > 0
::strlen(gnext), if n == 0
INT_MAX, if n < 0
```

If pbeg is a null pointer, sets only the input sequence to gnext. Otherwise, also sets the output sequence to pbeg.

```
strstreambuf(const char* gnext, streamsize n);
strstreambuf(const signed char* gnext, streamsize n);
strstreambuf(const unsigned char* gnext, streamsize n);
```

Constructs an object of class strstreambuf, initializing the base class with streambuf(). The argument gnext points to the first element of an array object whose number of elements is:

```
n, if n > 0
::strlen(gnext), if n == 0
INT_MAX, if n < 0
```

©Copyright 1999 Rogue Wave Software, Inc.

Sets the input sequence to `gnext` and the mode to `constant`.

Destructors

```
virtual ~strstreambuf();
```

Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if allocated is set and frozen is not set.

Member Functions

```
void
freeze(bool freezefl = 1);
```

If the mode is dynamic, alters the `freeze` status of the dynamic array as follows:

- If `freezefl` is `false`, the function sets the `freeze` status to `frozen`.
- Otherwise, it clears the `freeze` status.

```
int_type
overflow( int_type c = traits::eof() );
```

If the output sequence has a put position available, and `c` is not `traits::eof()`, then writes `c` into it. If there is no position available, the function increases the size of the array object by allocating more memory, and then writes `c` at the new current put position. If dynamic is not set or if frozen is set, the operation fails. The function returns `traits::not_eof(c)`, except if it fails, in which case it returns `traits::eof()`.

```
int_type
pbackfail( int_type c = traits::eof() );
```

Puts back the character designated by `c` into the input sequence. If `traits::eq_int_type(c,traits::eof())` returns `true`, move the input sequence one position backward. If the operation fails, the function returns `traits::eof()`. Otherwise it returns `traits::not_eof(c)`.

```
int
pcount() const;
```

Returns the size of the output sequence.

```
pos_type
seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which =
        ios_base::in | ios_base::out);
```

If the open mode is `in | out`, alters the stream position of both the input and the output sequence. If the open mode is `in`, alters the stream position of only the input sequence. If the open mode is `out`, alters the stream position of only the output sequence. The new position is calculated by combining the two parameters `off` (displacement) and `way` (reference point). If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position.

```
pos_type
seekpos(pos_type sp, ios_base::openmode which =
        ios_base::in | ios_base::out);
```

If the open mode is `in | out`, alters the stream position of both the input and the output sequence. If the open mode is `in`, alters the stream position of only the input sequence. If the open mode is `out`, alters the stream position of only the output sequence. If the current position of the sequence is invalid before repositioning, the operation fails and the return value is `pos_type(off_type(-1))`. Otherwise the function returns the current new position.

```
strstreambuf*
setbuf(char* s, streamsize n);
```

If `dynamic` is set and `freeze` is not, proceed as follows:

If `s` is not a null pointer and `n` is greater than the number of characters already in the current array, replaces it (copy its contents) by the array of size `n` pointed to by `s`.

©Copyright 1999 Rogue Wave Software, Inc.

```
char*
str();
```

Calls `freeze()`, then returns the beginning pointer for the input sequence.

```
int_type
underflow();
```

If the input sequence has a read position available, returns the content of this position. Otherwise tries to expand the input sequence to match the output sequence and if possible returns the content of the new current position. The function returns `traits::eof()` to indicate failure.

In the case where `s` is a null pointer and `n` is greater than the number of characters already in the current array, resizes it to size `n`.

If the function fails, it returns a null pointer.

```
streamsize
xsputn(const char_type* s, streamsize n);
```

Writes up to `n` characters to the output sequence. The characters written are obtained from successive elements of the array whose first element is designated by `s`. The function returns the number of characters written.

Example

```
//
// stdlib/examples/manual/strstreambuf.cpp
//
#include<iostream>
#include<strstream>
#include<iomanip>

void main ( )
{
  using namespace std;

  // create a read/write strstream object
  // and attach it to an ostrstream object
  ostrstream out;

  // tie the istream object to the ostrstream object
  istream in(out.rdbuf());

  // output to out
  out << "anticonstitutionellement is a big word !!!";

  // create a NTBS
  const char *p ="Le rat des villes et le rat des champs";

  // output the NTBS
  out << p << endl;

  // resize the buffer
  if ( out.rdbuf()->pubsetbuf(0,5000) )
   cout << endl << "Success in allocating the buffer"
        << endl;

  // output the all buffer to stdout
  cout << in.rdbuf( );

  // output the decimal conversion of 100 in hex
  // with right padding and a width field of 200
  out << dec << setfill('!') << setw(200) << 0x100 << endl;

  // output the content of the input sequence to stdout
  cout << in.rdbuf( ) << endl;

  // number of elements in the output sequence
  cout << out.rdbuf()->pcount() << endl;

  // resize the buffer to a minimum size
  if ( out.rdbuf()->pubsetbuf(0,out.rdbuf()->pcount()) )
   cout << endl << "Success in resizing the buffer" << endl;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // output the content of the all array object
  cout << out.rdbuf()->str() << endl;

 }
```

See Also

char_traits(3C++), ***ios_base***(3C++), ***basic_ios***(3C++), ***basic_streambuf***(3C++), ***istrstream***(3c++), ***ostrstream***(3C++), ***strstream***(3c++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Annex D Compatibility features Section D.5

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

swap

`Algorithm`

Summary

Exchanges values.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class T>
void swap (T& a, T& b);
```

Description

The ***swap*** algorithm exchanges the values of `a` and `b`. The effect is:

```
T tmp = a
a = b
b = tmp
```

See Also

iter_swap, ***swap_ranges***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

swap_ranges

`Algorithm`

Summary

Exchanges a range of values in one location with those in another.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges (ForwardIterator1 first1,
                 ForwardIterator1 last1,
                 ForwardIterator2 first2);
```

Description

The ***swap_ranges*** algorithm exchanges corresponding values in two ranges, in the following manner:

For each non-negative integer `n < (last - first)`, the function exchanges `*(first1 + n)` with `*(first2 + n))`. After completing all exchanges, ***swap_ranges*** returns an iterator that points to the end of the second container (in other words, `first2 + (last1 -first1)`). The result of ***swap_ranges*** is undefined if the two ranges `[first, last)` and `[first2, first2 + (last1 - first1))` overlap.

Example

```
//
// swap.cpp
//
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
int d1[] = {6, 7, 8, 9, 10, 1, 2, 3, 4, 5};
// Set up a vector
vector<int> v(d1+0,d1 + 10);

// Output original vector
cout << "For the vector: ";
copy(v.begin(),v.end(),
```

©Copyright 1999 Rogue Wave Software, Inc.

```
     ostream_iterator<int,char>(cout," "));

// Swap the first five elements with the last five elements
swap_ranges(v.begin(),v.begin()+5, v.begin()+5);
// Output result
cout << endl << endl
<< "Swapping the first five elements "
<< "with the last five gives: "
<< endl << "     ";
copy(v.begin(),v.end(),
     ostream_iterator<int,char>(cout," "));
return 0;
}
```

Program Output

```
For the vector: 6 7 8 9 10 1 2 3 4 5
Swapping the first five elements with the last five gives:
1 2 3 4 5 6 7 8 9 10
Swapping the first and last elements gives:
10 2 3 4 5 6 7 8 9 1
```

Warnings

If your compiler does not support default template parameters, you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

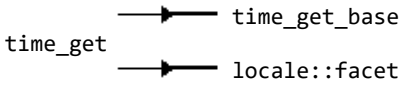
iter_swap, *swap*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

time_get



Summary

A time formatting facet for input.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
iter_type

Member Functions

date_order()	do_get_year()
do_date_order()	get_date()
do_get_date()	get_monthname()
do_get_monthname()	get_time()
do_get_time()	get_weekday()
do_get_weekday()	get_year()

Synopsis

```
#include <locale>
class time_base;
template <class charT, class InputIterator =
          istreambuf_iterator<charT> >
class time_get;
```

Description

The ***time_get*** facet extracts time and date components from a character string and stores the resulting values in a `struct tm` argument. The facet parses the string using a specific format as a guide. If the string does not fit the format, then the facet indicates an error by setting the `err` argument in the public member functions to `iosbase::failbit`. See member function descriptions for details.

©Copyright 1999 Rogue Wave Software, Inc.

The time_base class includes a set of values for specifying the order in which the three parts of a date appear. The `dateorder` function returns one of these five possible values:

Order	Meaning
noorder	Date format has no set ordering
dmy	Date order is day, month, year
mdy	Date order is month, day, year
ymd	Date order is year, month, day
ydm	Date order is year, day, month

Interface

```
class time_base {
public:
  enum dateorder { no_order, dmy, mdy, ymd, ydm };
};

template <class charT, class InputIterator =
          istreambuf_iterator<charT> >
class time_get : public locale::facet, public time_base {
public:
  typedef charT            char_type;
  typedef InputIterator    iter_type;
  explicit time_get(size_t = 0);
  dateorder date_order()  const;
  iter_type get_time(iter_type, iter_type, ios_base&,
                     ios_base::iostate&, tm*)  const;
  iter_type get_date(iter_type, iter_type, ios_base&,
                     ios_base::iostate&, tm*)  const;
  iter_type get_weekday(iter_type, iter_type, ios_base&,
                        ios_base::iostate&, tm*) const;
  iter_type get_monthname(iter_type, iter_type, ios_base&,
                          ios_base::iostate&, tm*) const;
  iter_type get_year(iter_type, iter_type, ios_base&,
                     ios_base::iostate&, tm*) const;
  static locale::id id;
protected:
  ~time_get();  // virtual
  virtual dateorder do_date_order()  const;
  virtual iter_type do_get_time(iter_type, iter_type,
          os_base&, ios_base::iostate&, tm*) const;
  virtual iter_type do_get_date(iter_type, iter_type,
          ios_base&, ios_base::iostate&, tm*) const;
  virtual iter_type do_get_weekday(iter_type, iter_type,
          os_base&, ios_base::iostate&, tm*) const;
  virtual iter_type do_get_monthname(iter_type, ios_base&,
          ios_base::iostate&, tm*) const;
  virtual iter_type do_get_year(iter_type, iter_type,
          ios_base&, ios_base::iostate&, tm*) const;
};
```

Types

`char_type`

Type of character the facet is instantiated on.

`iter_type`

Type of iterator used to scan the character buffer.

Constructors

`explicit` **`time_get`**`(size_t refs = 0)`

Constructs a ***time_get*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

©Copyright 1999 Rogue Wave Software, Inc.

Destructors

```
~time_get();  // virtual and protected
```

Destroys the facet.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

The public members of the ***time_get*** facet include an interface to protected members. Each public member `xxx` has a corresponding virtual protected member `do_xxx`. All work is delegated to these protected members. For instance, the long version of the public `get_time` function simply calls its protected cousin `do_get_time`.

```
dateorder
date_order()  const;
iter_type
get_date(iter_type s, iter_type end, ios_base& f,
         ios_base::iostate& err, tm* t)  const;
iter_type
get_monthname(iter_type s, iter_type end, ios_base& f,
              ios_base::iostate& err, tm* t) const;
iter_type
get_time(iter_type s, iter_type end, ios_base& f,
         ios_base::iostate& err, tm* t)  const;
iter_type
get_weekday(iter_type s, iter_type end, ios_base& f,
            ios_base::iostate& err, tm* t) const;
iter_type
get_year(iter_type s, iter_type end, ios_base& f,
         ios_base::iostate& err, tm* t) const;
```

Each of these public functions simply calls a corresponding protected virtual `do_` function.

Protected Member Functions

```
virtual dateorder
do_date_order()  const;
```

Returns the a value indicating the relative ordering of the three basic parts of a date. Possible return values are:

- `noorder`, indicating that the date format has no ordering, an ordering cannot be determined, or the date format contains variable components other than Month, Day and Year. `noorder` is never returned by the default time_put, but may be used by derived classes.
- One of `dmy, mdy, ymd, ydm`, indicating the relative ordering of Day, Month, and Year.

```
virtual iter_type
do_get_date(iter_type s, iter_type end, ios_base&,
            ios_base::iostate& err, tm* t) const;
```

Fills out the `t` argument with date values parsed from the character buffer specified by the range `(s,end]`. If the buffer does not contain a valid date representation, then the err argument is set to `iosbase::failbit`.

Returns an iterator pointing just beyond the last character that can be determined to be part of a valid date.

```
virtual iter_type
do_get_monthname(iter_type s, ios_base&,
                 ios_base::iostate& err, tm* t) const;
```

Fills out the `tm_mon` member of the `t` argument with a month name parsed from the character buffer specified by the range `(s,end]`. As with `do_get_weekday`, this name may be an abbreviation, but the function attempts to read a full name if valid characters are found after an abbreviation. For example, if a full name is "December", and an abbreviation is "Dec", then the string "Dece" causes an error. If an error occurs, then the `err` argument is set to `iosbase::failbit`.

©Copyright 1999 Rogue Wave Software, Inc.

Returns an iterator pointing just beyond the last character that can be determined to be part of a valid name.

```
virtual iter_type
do_get_time(iter_type s, iter_type end, os_base&,
            ios_base::iostate& err, tm* t) const;
```

Fills out the `t` argument with time values parsed from the character buffer specified by the range `(s,end]`. The buffer must contain a valid time representation. Returns an iterator pointing just beyond the last character that can be determined to be part of a valid time.

```
virtual iter_type
do_get_weekday(iter_type s, iter_type end, os_base&,
               ios_base::iostate& err, tm* t) const;
```

Fills out the `tm_wday` member of the `t` argument with a weekday name parsed from the character buffer specified by the range `(s,end]`. This name may be an abbreviation, but the function attempts to read a full name if valid characters are found after an abbreviation. For instance, if a full name is "Monday", and an abbreviation is "Mon", then the string "Mond" causes an error. If an error occurs, then the `err` argument is set to `iosbase::failbit`.

Returns an iterator pointing just beyond the last character that can be determined to be part of a valid name.

```
virtual iter_type
do_get_year(iter_type s, iter_type end, ios_base&,
            ios_base::iostate& err, tm* t) const;
```

Fills in the `tm_year` member of the `t` argument with a year parsed from the character buffer specified by the range `(s,end]`. If an error occurs, then the `err` argument is set to `iosbase::failbit`.

Returns an iterator pointing just beyond the last character that can be determined to be part of a valid year.

Example

```
//
// timeget.cpp
//
#include <locale>
#include <sstream>
#include <time.h>

  using namespace std;

// Print out a tm struct
ostream& operator<< (ostream& os, const struct tm& t)
{
  os << "Daylight Savings = " << t.tm_isdst << endl;
  os << "Day of year      = " << t.tm_yday << endl;
  os << "Day of week      = " << t.tm_wday << endl;
  os << "Year             = " << t.tm_year << endl;
  os << "Month            = " << t.tm_mon << endl;
  os << "Day of month     = " << t.tm_mday << endl;
  os << "Hour             = " << t.tm_hour << endl;
  os << "Minute           = " << t.tm_min << endl;
  os << "Second           = " << t.tm_sec << endl;
  return os;
}

int main ()
{
  typedef istreambuf_iterator<char,char_traits<char> >
    iter_type;

  locale loc;
  time_t tm = time(NULL);
  struct tm* tmb = localtime(&tm);
  struct tm timeb;
  memcpy(&timeb,tmb,sizeof(struct tm));
  ios_base::iostate state;
  iter_type end;

  // Get a time_get facet
  const time_get<char,iter_type>& tg =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<time_get<char,iter_type> >(loc);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#else
  use_facet(loc,(time_get<char,iter_type>*)0);
#endif

  cout << timeb << endl;
  {
    // Build an istringstream from the buffer and construct
    // beginning and ending iterators on it.
    istringstream ins("12:46:32");
    iter_type begin(ins);

    // Get the time
    tg.get_time(begin,end,ins,state,&timeb);
  }
  cout << timeb << endl;
  {
    // Get the date
    istringstream ins("Dec 6 1996");
    iter_type begin(ins);
    tg.get_date(begin,end,ins,state,&timeb);
  }
  cout << timeb << endl;
  {
    // Get the weekday
    istringstream ins("Tuesday");
    iter_type begin(ins);
    tg.get_weekday(begin,end,ins,state,&timeb);
  }
  cout << timeb << endl;
  return 0;
}
```

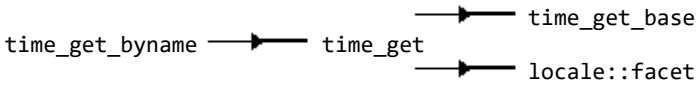
See Also

locale, *facets*, *time_put*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

time_get_byname



Summary

A time formatting facet for input, based on the named locales.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT, class InputIterator =
          istreambuf_iterator<charT> >
class time_get_byname;
```

Description

The ***time_get_byname*** template has the same function as the ***time_get.***template, but is specific to a particular named locale. For a description of the member functions of ***time_get_byname***, see the reference for ***time_get.***

Interface

```
template <class charT, class InputIterator =
          istreambuf_iterator<charT> >
class time_get_byname : public time_get<charT, InputIterator> {
public:
  explicit time_get_byname(const char*, size_t = 0);
protected:
  ~time_get_byname();  // virtual
  virtual dateorder do_date_order()  const;
  virtual iter_type do_get_time(iter_type, iter_type,
          ios_base&, ios_base::iostate&, tm*) const;
  virtual iter_type do_get_date(iter_type, iter_type,
          ios_base::iostate&, tm*) const;
  virtual iter_type do_get_weekday(iter_type, iter_type,
          ios_base&, ios_base::iostate& err, tm*) const;
  virtual iter_type do_get_monthname(iter_type, iter_type,
         ios_base&, ios_base::iostate&, tm*) const;
  virtual iter_type do_get_year(iter_type, iter_type,
          ios_base&, ios_base::iostate&, tm*) const;
};
```

Constructors

©Copyright 1999 Rogue Wave Software, Inc.

```
explicit time_get_byname(const char* name,
                         size_t refs = 0);
```

Constructs a ***time_get_byname*** facet, which is a time formatting facet for input relative to the named locale specified by the `name` argument. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

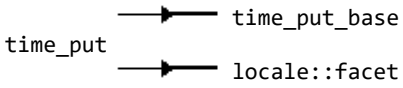
See Also

locale, ***facets***, ***time_get***, ***time_put_byname***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

time_put



Summary

A time formatting facet for output.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Data Types

char_type
id
iter_type

Member Functions

do_put()
put()

Synopsis

```
#include <locale>
template <class charT, class OutputIterator =
          ostreambuf_iterator<charT> >
class time_put;
```

Description

The ***time_put*** facet includes facilities for formatted output of date/time values. The member function of ***time_put*** takes a date/time in the form of a struct `tm` and translates this into a character string representation.

Interface

```
template <class charT, class OutputIterator =
          ostreambuf_iterator<charT> >
class time_put : public locale::facet {
public:
  typedef charT            char_type;
  typedef OutputIterator   iter_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  explicit time_put(size_t = 0);
  iter_type put(iter_type, ios_base&,
                char_type, const tm*,
                const charT*, const charT*) const;
  iter_type put(iter_type, ios_base&, char_type,
                const tm*, char, char = 0) const;
  static locale::id id;
protected:
  ~time_put();  // virtual
  virtual iter_type do_put(iter_type, ios_base&,
                           char_type, const tm*,
                           char, char) const;
};
```

Types

char_type

Type of character the facet is instantiated on.

iter_type

Type of iterator used to scan the character buffer.

Constructors

```
explicit time_put(size_t refs = 0);
```

Constructs a ***time_put*** facet. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

Destructors

```
~time_put();  // virtual and protected
```

Destroys the facet.

Facet ID

```
static locale::id id;
```

Unique identifier for this type of facet.

Public Member Functions

```
iter_type
put(iter_type s, ios_base& f,
    char_type fill, const tm* tmb,
    const charT* pattern, const charT* pat_end) const;
```

Creates a character string representing the Date/Time contained in `tmb`. The format of the string is determined by a sequence of format modifiers contained in the range `[pattern,pat_end)`. These modifiers are from the same set as those used by the `strftime` function and are applied in exactly the same way. The resulting string is written out to the buffer pointed to by the iterator `s`. See the table below for a description of `strftime` formatting characters.

The `fill` argument is used for any padding.

Returns an iterator pointing one past the last character written.

Protected Member Functions

```
iter_type
do_put(iter_type s, ios_base& f, char_type fill,
    const tm* tmb, char format, char modifier = 0) const;
```

Calls the protected virtual `do_put` function.

©Copyright 1999 Rogue Wave Software, Inc.

Writes out a character string representation of the Date/Time contained in `t`. The string is formatted according the specifier `format` and modifier `modifier`. These values are interpreted in exactly the same way as the `strftime` function interprets its format and modifier flags. See the table below for a description of `strftime` formatting characters.

The `fill` argument is used for any padding.

Returns an iterator pointing one past the last character written.

Table 1 -- Formatting characters used by strftime() . For those formats that do not use all members of the struct tm, only those members that are actually used are noted [in brackets].

Format character	Meaning	Example
a	Abbreviated weekday name [from tm::tm_wday]	Sun
A	Full weekday name [from tm::tm_wday]	Sunday
b	Abbreviated month name	Feb
B	Full month name	February
c	Date and time [may use all members]	Feb 29 14:34:56 1984
d	Day of the month	29
H	Hour of the 24-hour day	14
I	Hour of the 12-hour day	02
j	Day of the year, from 001 [from tm::tm_yday]	60
m	Month of the year, from 01	02
M	Minutes after the hour	34
p	AM/PM indicator, if any	AM
S	Seconds after the minute	56
U	Sunday week of the year, from 00 [from tm::tm_yday and tm::tm_wday]	
w	Day of the week, with 0 for Sunday	0
W	Monday week of the year, from 00 [from tm::tm_yday and tm::tm_wday]	
x	Date [uses tm::tm_yday in some locales]	Feb 29 1984
X	Time	14:34:56
y	Year of the century, from 00 (deprecated)	84
Y	Year	1984
Z	Time zone name [from tm::tm_isdst]	PST or PDT

Example

```
//
// timeput.cpp
//
#include <iostream>

int main ()
{
  using namespace std;

  typedef ostreambuf_iterator<char,char_traits<char> >
          iter_type;

  locale loc;
  time_t tm = time(NULL);
  struct tm* tmb = localtime(&tm);
  struct tm timeb;
  memcpy(&timeb,tmb,sizeof(struct tm));
  char pat[] = "%c";

  // Get a time_put facet
  const time_put<char,iter_type>& tp =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
  use_facet<time_put<char,iter_type> >(loc);
#else
  use_facet(loc,(time_put<char,iter_type>*)0);
#endif
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // Construct a ostreambuf_iterator on cout
  iter_type begin(cout);

  cout << " --> ";
  tp.put(begin,cout,' ',&timeb,pat,pat+2);

  cout << endl << " --> ";
  tp.put(begin,cout,' ',&timeb,'c',' ');

  cout <<  endl;

  return 0;
}
```

See Also

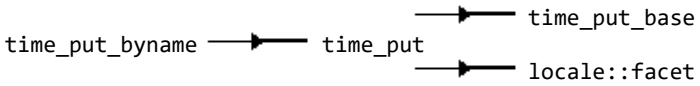
locale, *facets*, *time_get*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

time_put_byname



Summary

A facet that includes formatted time output facilities based on the named locales.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT, class OuputIterator =
          ostreambuf_iterator<charT> >
class time_put_byname;
```

Description

The ***time_put_byname*** template has the same function as the ***time_put.***template, but is specific to a particular named locale. For a description of the member functions of ***time_put_byname,*** see the reference for ***time_put.***

Interface

```
template <class charT, class OutputIterator =
  ostreambuf_iterator<charT> >
class time_put_byname : public time_put<charT,
  OutputIterator>
{
public:
  explicit time_put_byname(const char*, size_t refs = 0);
protected:
  ~time_put_byname();  // virtual
  virtual iter_type do_put(iter_type s, ios_base&,
                           char_type, const tm* t,
                           char format, char modifier) const;
};
```

Constructors

```
explicit time_put_byname(const char* name,
                           size_t refs = 0);
```

©Copyright 1999 Rogue Wave Software, Inc.

Constructs a ***time_put_byname*** facet, which is a time formatting facet for output relative to the named locale specified by the `name` argument. If the `refs` argument is `0`, then destruction of the object is delegated to the locale, or locales, containing it. This allows the user to ignore lifetime management issues. On the other Hand, if `refs` is `1`, then the object must be explicitly deleted; the locale does not do so. In this case, the object can be maintained across the lifetime of multiple locales.

See Also

locale, ***facets***, ***time_put***, ***time_get_byname***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

tolower

`Locale Convenience Function`

Summary

Converts a character to lower case.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT>
charT tolower (charT c, const locale& loc) const;
```

Description

The ***tolower*** function returns the parameter `c` after converting it to lower case. The conversion is made using the ***ctype*** facet from the locale parameter.

Example

```
//
// toupper.cpp
//
#include <iostream>

int main ()
{
  using namespace std;
  locale loc;
  cout << 'a' << toupper('c') << tolower('F') << endl;

  return 0;
}
```

See Also

toupper, ***locale***, ***ctype***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

toupper

`Locale Convenience Function`

Summary

Converts a character to upper case.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class charT>
charT toupper (charT c, const locale& loc) const;
```

Description

The ***toupper*** function returns the parameter `c` after converting it to upper case. The conversion is made using the ***ctype*** facet from the locale parameter.

Example

```
//
// toupper.cpp
//
#include <iostream>

int main ()
{
  using namespace std;
  locale loc;
  cout << 'a' << toupper('c') << tolower('F') << endl;

  return 0;
}
```

See Also

tolower, ***locale***, ***ctype***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

transform

`Algorithm`

Summary

Applies an operation to a range of values in a collection and stores the result.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class InputIterator, class OutputIterator,
          class UnaryOperation>
  OutputIterator
    transform (InputIterator first, InputIterator last,
               OutputIterator result, UnaryOperation op);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation>
  OutputIterator
    transform (InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, OutputIterator result,
               BinaryOperation binary_op);
```

Description

The ***transform*** algorithm has two forms. The first form applies unary operation `op` to each element of the range `[first, last)`, and sends the result to the output iterator `result`. For example, this version of ***transform*** could be used to square each element in a vector. If the output iterator (`result`) is the same as the input iterator used to traverse the range, ***transform*** performs its transformation in place.

The second form of ***transform*** applies a binary operation, `binary_op`, to corresponding elements in the range `[first1, last1)` and the range that begins at `first2`, and sends the result to `result`. For example, ***transform*** can be used to add corresponding elements in two sequences, and store the set of sums in a third. The algorithm assumes, but does not check, that the second sequence has at least as many elements as the first sequence. Note that the output iterator `result` can be a third sequence, or either of the two input sequences.

Formally, ***transform*** assigns through every iterator `i` in the range `[result, result + (last1 - first1))` a new corresponding value equal to:

```
op(*(first1 + (i - result))
```

or

```
binary_op(*(first1 + (i - result), *(first2 + (i - result)))
```

©Copyright 1999 Rogue Wave Software, Inc.

transform returns `result + (last1 - first1)`. `op` and `binary_op` must not have any side effects. `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

Complexity

Exactly `last1 - first1` applications of `op` or `binary_op` are performed.

Example

```
//
// trnsform.cpp
//
#include <functional>
#include <deque>
#include <algorithm>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
 {

//Initialize a deque with an array of ints
int arr1[5] = {99, 264, 126, 330, 132};
int arr2[5] = {280, 105, 220, 84, 210};
deque<int> d1(arr1+0, arr1+5), d2(arr2+0, arr2+5);

//Print the original values
cout << "The following pairs of numbers: "
     << endl << "     ";
deque<int>::iterator i1;
for(i1 = d1.begin(); i1 != d1.end(); i1++)
  cout << setw(6) << *i1 << " ";
cout << endl << "     ";
for(i1 = d2.begin(); i1 != d2.end(); i1++)
  cout << setw(6) << *i1 << " ";

// Transform the numbers in the deque to their
// factorials and store in the vector
transform(d1.begin(), d1.end(), d2.begin(),
          d1.begin(), multiplies<int>());

//Display the results
cout << endl << endl;
cout << "Have the products: " << endl << "     ";
for(i1 = d1.begin(); i1 != d1.end(); i1++)
  cout << setw(6) << *i1 << " ";
return 0;
 }
```

Program Output

```
The following pairs of numbers:
         99    264    126    330    132
        280    105    220     84    210
Have the products:
27720  27720  27720  27720  27720
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

unary_function

`Function Object`

Summary

A base class for creating unary function objects.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <functional>
template <class Arg, class Result>
struct unary_function{
  typedef Arg argument_type;
  typedef Result result_type;
};
```

Description

Function objects are objects with an `operator()` defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an `operator()` defined or a pointer to a function. The standard library includes both a standard set of function objects and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take one argument are called *unary function objects.* Unary function objects are required to include the typedefs `argument_type` and `result_type`. The ***unary_function*** class makes the task of creating templatized unary function objects easier by providing the necessary typedefs for a unary function object. You can create your own unary function objects by inheriting from ***unary_function***.

See Also

Function Objects, and Function Objects Section in User's Guide.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

unary_negate

`Function Adapter (Negator)`

Summary

A function object that returns the complement of the result of its unary predicate

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

operator()()

Synopsis

```
#include<functional>
template <class Predicate>
class unary_negate : public
      unary_function<typename Predicate::argument_type,
                     bool>;
```

Description

unary_negate is a function object class that includes a return type for the function adapter ***not1***. ***not1*** is a function adapter, known as a negator, that takes a unary predicate function object as its argument and returns a unary predicate function object that is the complement of the original.

Note that ***not1*** works only with function objects that are defined as subclasses of the class ***unary_function***.

Interface

```
template <class Predicate>
class unary_negate : public
   unary_function<typename Predicate::argument_type, bool> {
public:
   explicit unary_negate (const Predicate&);
   bool operator() (const typename
                    Predicate::argument_type&) const;
};

template<class Predicate>
unary_negate <Predicate> not1 (const Predicate&);
```

©Copyright 1999 Rogue Wave Software, Inc.

Constructors

```
explicit
unary_negate(const Predicate& pred);
```

Constructs a `unary_negate` object from predicate `pred`.

Operators

```
bool
operator()(const typename Predicate::argument_type& x)
           const;
```

Returns the result of `unary_negate`<predicate>`(pred)`.

See Also

not1, ***not2***, ***unary_function***, ***binary_negate***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

uninitialized_copy

`Memory Management`

Summary

An algorithm that uses ***construct*** to copy values from one range to another location.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <memory>
template <class InputIterator, class ForwardIterator>
  ForwardIterator
    uninitialized_copy (InputIterator first,
                        InputIterator last,
                        ForwardIterator result);
```

Description

uninitialized_copy copies all items in the range `[first, last)` into the location beginning at `result` using the ***construct*** algorithm.

See Also

construct

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

uninitialized_fill

Memory Management

Summary

An algorithm that uses the ***construct*** algorithm for setting values in a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <memory>
template <class ForwardIterator, class T>
  void
     uninitialized_fill(ForwardIterator first,
                        ForwardIterator last, const T& x);
```

Description

uninitialized_fill initializes all of the items in the range `[first, last)` to the value `x`, using the ***construct*** algorithm.

See Also

construct, ***uninitialized_fill_n***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents Index

uninitialized_fill_n

`Memory Management`

Summary

An algorithm that uses the ***construct*** algorithm for setting values in a collection.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <memory>
template <class ForwardIterator, class Size, class T>
  void
    uninitialized_fill_n (ForwardIterator first, Size n,
                          const T& x);
```

Description

unitialized_fill_n starts at the iterator `first` and initializes the first `n` items to the value `x`, using the ***construct*** algorithm.

See Also

construct, ***uninitialized_fill***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

unique, unique_copy

`Algorithm`

Summary

Removes consecutive duplicates from a range of values and places the resulting unique values into the result.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator>
  ForwardIterator
    unique (ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    unique (ForwardIterator first, ForwardIterator last,
            BinaryPredicate binary_pred);
template <class InputIterator, class OutputIterator>
  OutputIterator
    unique_copy (InputIterator first, InputIterator last,
                 OutputIterator result);
template <class InputIterator, class OutputIterator,
          class BinaryPredicate>
  OutputIterator
    unique_copy (InputIterator first, InputIterator last,
                 OutputIterator result,
                 BinaryPredicate binary_pred);
```

Description

The ***unique*** algorithm moves through a sequence and eliminates all but the first element from every consecutive group of equal elements. There are two versions of the algorithm-one that tests for equality and a second that tests adjacent elements against a binary predicate. An element is unique if it does not meet the corresponding condition listed here:

```
*i == *(i - 1)
```

or

```
binary_pred(*i, *(i - 1)) == true.
```

If an element is unique, it is copied to the front of the sequence, overwriting the existing elements. Once all unique elements have been identified. The remainder of the sequence is left unchanged, and ***unique*** returns the end of the resulting range.

©Copyright 1999 Rogue Wave Software, Inc.

The ***unique_copy*** algorithm copies the first element from every consecutive group of equal elements to an ***OutputIterator***. The ***unique_copy*** algorithm also has two versions-one that tests for equality and a second that tests adjacent elements against a binary predicate.

unique_copy returns the end of the resulting range.

Complexity

For ***unique_copy***, it is exactly `(last - first) - 1` applications of the corresponding predicate are performed.

Example

```
//
// unique.cpp
//
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
int main()
 {
//Initialize two vectors
int a1[20] = {4, 5, 5, 9, -1, -1, -1, 3, 7, 5,
              5, 5, 6, 7, 7, 7, 4, 2, 1, 1};
vector<int> v(a1+0, a1+20), result;

//Create an insert_iterator for results
insert_iterator<vector<int> > ins(result, result.begin());

//Demonstrate includes
cout << "The vector: " << endl << "    ";
copy(v.begin(),v.end(),
     ostream_iterator<int,char>(cout," "));

//Find the unique elements
unique_copy(v.begin(), v.end(), ins);

//Display the results
cout << endl << endl
     << "Has the following unique elements:"
     << endl << "     ";
copy(result.begin(),result.end(),
     ostream_iterator<int,char>(cout," "));
return 0;
}
```

Program Output

```
The vector:
4 5 5 9 -1 -1 -1 3 7 5 5 5 6 7 7 7 4 2 1 1
Has the following unique elements:
4 5 9 -1 3 7 5 6 7 4 2 1
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

upper_bound

`Algorithm`

Summary

Determines the last valid position for a value in a sorted container.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template <class ForwardIterator, class T, class Compare>
  ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

Description

The ***upper_bound*** algorithm is one of a set of binary search algorithms. All of these algorithms perform binary searches on ordered containers. Each algorithm has two versions. The first version uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version allows you to include a function object of type `Compare`, and assumes that `Compare` is the function used to sort the sequence. The function object must be a binary predicate.

The ***upper_bound*** algorithm finds the last position in a container that `value` can occupy without violating the container's ordering. ***upper_bound***'s return value is the iterator for the first element in the container that is greater than `value`, or, when the comparison operator is used, the first element that does NOT satisfy the comparison function. Because the algorithm is restricted to using the less than operator or the user-defined function to perform the search, ***upper_bound*** returns an iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the appropriate version of the following conditions holds:

```
!(value < *j)
```

or

```
comp(value, *j) == false
```

Complexity

©Copyright 1999 Rogue Wave Software, Inc.

upper_bound performs at most `log(last - first) + 1` comparisons.

Example

```
//
// ul_bound.cpp
//
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

int main()
 {
typedef vector<int>::iterator iterator;
int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};

// Set up a vector
vector<int> v1(d1 + 0,d1 + 11);

// Try lower_bound variants
iterator it1 = lower_bound(v1.begin(),v1.end(),3);
// it1 = v1.begin() + 4

iterator it2 =
   lower_bound(v1.begin(),v1.end(),2,less<int>());
// it2 = v1.begin() + 4

// Try upper_bound variants
iterator it3 = upper_bound(v1.begin(),v1.end(),3);
// it3 = vector + 5

iterator it4 =
   upper_bound(v1.begin(),v1.end(),2,less<int>());
// it4 = v1.begin() + 5
cout << endl << endl
     << "The upper and lower bounds of 3: ( "
     << *it1 << " , " << *it3 << " ]" << endl;
cout << endl << endl
     << "The upper and lower bounds of 2: ( "
     << *it2 << " , " << *it4 << " ]" << endl;
return 0;
 }
```

Program Output

```
The upper and lower bounds of 3: ( 3 , 4 ]
The upper and lower bounds of 2: ( 2 , 3 ]
```

Warnings

If your compiler does not support default template parameters, then you always need to supply the `Allocator` template argument. For instance, you need to write:

```
vector<int, allocator<int> >
```

instead of:

```
vector<int>
```

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

lower_bound, ***equal_range***

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

use_facet

Locale Function

Summary

A template function used to obtain a facet.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <locale>
template <class Facet> const Facet& use_facet(const locale&);
```

Description

use_facet returns a reference to the corresponding facet contained in the locale argument. You specify the facet type by explicitly including the template parameter (see the example below). If that facet is not present, then ***use_facet*** throws ***runtime_error.*** Otherwise, the reference remains valid for as long as any copy of the locale exists.

Note that if your compiler cannot overload function templates on return type, then you need to use an alternate ***use_facet*** template. The alternate template takes an additional argument that is a pointer to the type of facet you want to extract from the locale. The declaration looks like this:

```
template <class Facet>
const Facet& use_facet(const locale&, Facet*);
```

The example below shows the use of both variations of ***use_facet***.

Example

```
//
// usefacet.cpp
//
#include <iostream>

int main ()
{
  using namespace std;

  locale loc;

  // Get a ctype facet
  const ctype<char>& ct =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
     use_facet<ctype<char> >(loc);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#else
     use_facet(loc,(ctype<char>*)0);
#endif

  cout << 'a' << ct.toupper('c') << endl;

  return 0;
}
```

See Also

locale, ***facets***, ***has_facet***

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

valarray

`Container`

Summary

An optimized array class for numeric operations.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

abs()	operator!()	operator<()	operator^()
acos()	operator!=()	operator<=()	operator^=()
apply()	operator%()	operator*()	operator~()
asin()	operator%=()	operator*=()	pow()
atan()	operator&&()	operator+()	resize()
atan2()	operator&()	operator+=()	shift()
cos()	operator&=()	operator-()	sin()
cosh()	operator>>()	operator-=()	sinh()
cshift()	operator>>=()	operator/()	size()
exp()	operator>()	operator/=()	sqrt()
log()	operator>=()	operator=()	sum()
log10()	operator<<()	operator==()	tan()
max()	operator<<=()	operator[]()	tanh()
min()			

Synopsis

```
#include <valarray>
template <class T >
class valarray ;
```

Description

©Copyright 1999 Rogue Wave Software, Inc.

valarray<T> and associated classes (***slice_array***, ***gslice_array***, ***mask_array***, and ***indirect_array***) represent and manipulate one dimensional arrays of values. Elements in a ***valarray*** are indexed sequentially beginning with zero.

Unlike other classes in the Standard Library, ***valarray<T>*** can only be used with a fairly narrow range of types. This restriction ensures that numeric operations on a ***valarray*** can be as efficient as possible by avoiding aliasing ambiguities and excess temporaries.

Interface

```
template <class T> class valarray {
public:

// types
typedef T value_type;

// constructors
valarray( );
explicit valarray(size_t);
valarray(const T&, size_t);
valarray(const T* , size_t);
valarray(const valarray<T>&);
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);

// destructor
~valarray();

// operator =
valarray<T>& operator= (const valarray<T>&);
valarray<T>& operator= (const slice_array<T>&);
valarray<T>& operator= (const gslice_array<T>&);
valarray<T>& operator= (const mask_array<T>&);
valarray<T>& operator= (const indirect_array<T>&);
valarray<T>& operator= (const T&);
// operator[]
T operator[] (size_t) const;
T& operator[] (size_t);
valarray<T> operator[](slice) const;
inline slice_array<T> operator[](slice);
valarray<T> operator[](const gslice&) const;
inline gslice_array<T> operator[](const gslice&);
valarray<T> operator[](const valarray<bool>&) const;
inline mask_array<T> operator[](const valarray<bool>&);
valarray<T> operator[](const valarray<size_t>&) const;
inline indirect_array<T> operator[](const valarray<size_t>&);

// unary operators
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;

// computed assignment
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator<<= (const valarray<T>&);
valarray<T>& operator>>= (const valarray<T>&);
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<= (const T&);
valarray<T>& operator>>= (const T&);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// others
size_t size() const;
T sum() const;
T min() const;
T max() const;

valarray<T> shift(int) const;
valarray<T> cshift(int) const;

valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
void free();
void resize(size_t, const T& = T() );
};

// Non-member binary operators
template<class T> valarray<T>
operator* (const valarray<T>& , const valarray<T>&  );
template<class T> valarray<T>
operator/ (const valarray<T>& , const valarray<T>& );
template<class T> valarray<T>
operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T>
operator+ (const valarray<T>& , const valarray<T>& );
template<class T> valarray<T>
operator- (const valarray<T>& , const valarray<T>& );
template<class T> valarray<T>
operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T>
operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T>
operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T>
operator<< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T>
operator>> (const valarray<T>&, const valarray<T>&);

template<class T> valarray<T>
operator* (const valarray<T>& , const T& );
template<class T> valarray<T>
operator/ (const valarray<T>& , const T& );
template<class T> valarray<T>
operator% (const valarray<T>&, const T&);
template<class T> valarray<T>
operator+ (const valarray<T>& , const T& );
template<class T> valarray<T>
operator- (const valarray<T>& , const T& );
template<class T> valarray<T>
operator^ (const valarray<T>&, const T&);
template<class T> valarray<T>
operator& (const valarray<T>&, const T&);
template<class T> valarray<T>
operator| (const valarray<T>&, const T&);
template<class T> valarray<T>
operator<< (const valarray<T>&, const T&);
template<class T> valarray<T>
operator>> (const valarray<T>&, const T&);

template<class T> valarray<T>
operator* (const T& , const valarray<T>& );
template<class T> valarray<T>
operator/ (const T& , const valarray<T>& );
template<class T> valarray<T>
operator% (const T&, const valarray<T>&);
template<class T> valarray<T>
operator+ (const T& , const valarray<T>& );
template<class T> valarray<T>
operator- (const T& , const valarray<T>& );
template<class T> valarray<T>
operator^ (const T&, const valarray<T>&);
template<class T> valarray<T>
operator& (const T&, const valarray<T>&);
template<class T> valarray<T>
operator| (const T&, const valarray<T>&);
template<class T> valarray<T>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
operator<< (const T&, const valarray<T>&);
template<class T> valarray<T>
operator>> (const T&, const valarray<T>&);

// Non-member logical operators

template<class T> valarray<bool>
operator== (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator!= (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator< (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator> (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator<= (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator>= (const valarray<T>& , const valarray<T>& );
template<class T> valarray<bool>
operator|| (const valarray<T>& , const valarray<T>&);
template<class T> valarray<bool>
operator&& (const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool>
operator== (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator!= (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator< (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator> (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator<= (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator>= (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator|| (const valarray<T>& , const T& );
template<class T> valarray<bool>
operator&& (const valarray<T>&, const T&);

template<class T> valarray<bool>
operator== (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator!= (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator< (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator> (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator<= (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator>= (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator|| (const T& , const valarray<T>& );
template<class T> valarray<bool>
operator&& (const T&, const valarray<T>&);

// non-member transcendental functions

template<class T> valarray<T> abs(const valarray<T>& );
template<class T> valarray<T> acos(const valarray<T>& );
template<class T> valarray<T> asin(const valarray<T>& );
template<class T> valarray<T> atan(const valarray<T>& );
template<class T> valarray<T> cos(const valarray<T>& );
template<class T> valarray<T> cosh(const valarray<T>& );
template<class T> valarray<T> exp(const valarray<T>& );
template<class T> valarray<T> log(const valarray<T>& );
template<class T> valarray<T> log10(const valarray<T>& );
template<class T> valarray<T> sinh(const valarray<T>& );
template<class T> valarray<T> sin(const valarray<T>& );
template<class T> valarray<T> sqrt(const valarray<T>& );
template<class T> valarray<T> tan(const valarray<T>& );
template<class T> valarray<T> tanh(const valarray<T>& );

template<class T> valarray<T>
atan2(const valarray<T>& , const valarray<T>& );
template<class T> valarray<T>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
atan2(const valarray<T>& , const T& );
template<class T> valarray<T>
atan2(const T& , const valarray<T>& );
template<class T> valarray<T>
pow(const valarray<T>& , const valarray<T>& );
template<class T> valarray<T>
pow(const valarray<T>& , const T& );
template<class T> valarray<T>
pow(const T& , const valarray<T>& );
```

Constructors

valarray();

Creates a ***valarray*** of length zero.

explicit **valarray**(size_t n);

Creates a ***valarray*** of length n, containing n values initialized with the default value for type T. T must have a default constructor.

explicit **valarray**(const T& value, size_t n);

Creates a ***valarray*** of length n, containing n values initialized with value.

explicit **valarray**(const T* value, size_t n);

Creates a ***valarray*** of length n, containing n values initialized with the first n elements pointed to by value. The array pointed to by value must contain at least n values.

valarray(const valarray<T>& x);

Creates a copy of x.

valarray(const slice_array<T>& x);

Creates a ***valarray*** from the ***slice_array*** x.

valarray(const gslice_array<T>& x);

Creates a ***valarray*** from the ***gslice_array*** x.

valarray(const mask_array<T>& x);

Creates a ***valarray*** from the ***mask_array*** x.

valarray(const indirect_array<T>& x);

Creates a ***valarray*** from the ***indirect_array*** x.

Destructors

~valarray();

Applies ~T() to every element in the ***valarray*** and returns all allocated memory.

Assignment Operators

```
valarray<T>&
operator=(const valarray<T>& x);
```

Assigns to each element of self the corresponding value from x. If self has more or fewer elements than x, then self is resized to match the size of x. Returns a reference to self.

```
valarray<T>&
operator=(const T& x);
```

©Copyright 1999 Rogue Wave Software, Inc.

Assigns to each element of self the value of x. Returns a reference to self.

```
valarray<T>&
operator=(const slice_array<T>& x);
```

Copies elements from x into self by stepping through each slice consecutively. If self has more or fewer elements than x, then self is resized to match the size of x. Returns a reference to self.

```
valarray<T>&
operator=(const gslice_array<T>& x);
```

Copies elements from x into self by stepping through each slice consecutively. If self has more or fewer elements than x, then self is resized to match the size of x. Returns a reference to self.

```
valarray<T>&
operator=(const mask<T>& x);
```

Copies each consecutive element from x into self. If self has more or fewer elements than x, then self is resized to match the size of x. Returns a reference to self.

```
valarray<T>&
operator=(const indirect_array<T>& x);
```

Copies each consecutive element from x into self. If self has more or fewer elements than x, then self is resized to match the size of x. Returns a reference to self.

Reference Operators

```
T& operator[](size_type n);
```

Returns a reference to element n of self. The result can be used as an lvalue. This reference is valid until the resize function is called or the array is destroyed. The index n must be between 0 and size less one.

```
T operator[](size_type n) const;
```

Returns the value at element n of self. The index n must be between 0 and size less one.

Subset Operators

```
valarray<T> operator[](slice s) const;
```

Returns a subset of the self as specified by s. The return value is a new ***valarray*** object. See ***slice*** for a description of a BLAS-like slice.

```
slice_array<T> operator[](slice s);
```

Returns a subset of the self as specified by s. The return value is a ***slice_array*** referencing elements inside self. See ***slice*** and ***slice_array***.

```
valarray<T> operator[](const gslice& s) const;
```

Returns a subset of the self as specified by s. The return value is a new ***valarray*** object. See ***gslice*** for a description of a generalized slice.

```
gslice_array<T> operator[](const gslice& s);
```

Returns a subset of the self as specified by s. The return value is a ***gslice_array*** referencing elements inside self. See ***gslice*** and ***gslice_array***.

```
valarray<T> operator[](const valarray<bool>& v) const;
```

Returns a subset of the self as specified by s. The return value is a new ***valarray*** object.

```
mask_array<T> operator[](const valarray<bool>& v);
```

Returns a subset of the self as specified by s. The return value is a ***mask_array*** referencing elements inside self. See ***mask_array***.

```
valarray<T> operator[](const valarray<size_t>& v) const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns a subset of the self as specified by `s`. The return value is a new ***valarray*** object.

```
Indirect_array<T> operator[](const valarray<size_t>& v);
```

Returns a subset of the self as specified by `s`. The return value is a ***indirect_array*** referencing elements inside self. See ***indirect_array***.

Unary Operators

```
valarray<T> operator+() const;
```

Returns a new ***valarray*** object of the same size as the array in self where each element has been initialized by applying `operator+` to the corresponding element in self. This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports an `operator+` that returns `T` or a type convertible to `T`.

```
valarray<T> operator-() const;
```

Returns a new ***valarray*** object of the same size as the array in self where each element has been initialized by applying `operator-` to the corresponding element in self. This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports an `operator-` returning `T` or a type convertible to `T`.

```
valarray<T> operator~() const;
```

Returns a new ***valarray*** object of the same size as the array in self where each element has been initialized by applying `operator~` to the corresponding element in self. This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports an `operator~` returning `T` or a type convertible to `T`.

```
valarray<bool> operator!() const;
```

Returns a new ***valarray*** object of the same size as the array in self where each element has been initialized by applying `operator!` to the corresponding element in self. This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports an `operator!` returning `bool` or a type convertible to `bool`.

Computed Assignment Operators

```
valarray<T>& operator*=(const valarray<T>& val);
valarray<T>& operator/=(const valarray<T>& val);
valarray<T>& operator%=(const valarray<T>& val);
valarray<T>& operator+=(const valarray<T>& val);
valarray<T>& operator-=(const valarray<T>& val);
valarray<T>& operator^=(const valarray<T>& val);
valarray<T>& operator&=(const valarray<T>& val);
valarray<T>& operator|=(const valarray<T>& val);
valarray<T>& operator<<=(const valarray<T>& val);
valarray<T>& operator>>=(const valarray<T>& val);
```

Applies the indicated operation to each element in self, using the corresponding element from `val` as the right hand argument (for example, for all `0 <= n < *this.size(), *this[n] += val[n]`). This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports the indicated operation. The length of `val` must also equal the length of self. Returns self.

```
valarray<T>& operator*=(const T& val);
valarray<T>& operator/=(const T& val);
valarray<T>& operator%=(const T& val);
valarray<T>& operator+=(const T& val);
valarray<T>& operator-=(const T& val);
valarray<T>& operator^=(const T& val);
valarray<T>& operator&=(const T& val);
valarray<T>& operator|=(const T& val);
valarray<T>& operator<<=(const T& val);
valarray<T>& operator>>=(const T& val);
```

Applies the indicated operation to each element in self, using `val` as the right hand argument (for example, for all `0 <= n < *this.size(), *this[n] += val`). This operation can only be applied to a ***valarray*** instantiated on a type `T` that supports the indicated operation. Returns self.

Member Functions

```
size_t size() const;
```

©Copyright 1999 Rogue Wave Software, Inc.

Returns the number of elements.

```
T sum() const;
```

This function uses `operator+=` to sum all the elements of the array. Sum can only be called for a ***valarray*** instantiated on a type that supports `operator+=`. The array must also have at least one element. Returns the sum of all elements in the array.

```
T min() const;
```

This function uses `operator<` to find the minimum element in the array. The array must have at least one element. Returns the minimum of all elements in the array.

```
T max() const;
```

This function uses `operator>` to find the maximum element in the array. The array must have at least one element. Returns the maximum of all elements in the array.

```
valarray<T> shift(int n) const;
```

This function returns a new ***valarray*** object whose elements have all been shifted `n` places to left or right with respect to self. A positive value of `n` shifts the elements to the left, a negative value to the right. The default constructor for `T` is used to fill in behind the shifting elements. For example, applying `shift(2)` to an array corresponding to [3,4,5,6] results in [5,6,0,0], and applying `shift(-1)` to [3,4,5,6] results in [0,3,4,5].

```
valarray<T> cshift(int n) const;
```

This function returns a new ***valarray*** object whose elements have all been rotated `n` places to left or right with respect to self. A positive value of `n` shifts the elements to the left, a negative value to the right. For example, applying `shift(2)` to an array corresponding to [3,4,5,6] results in [5,6,3,4], and applying `shift(-1)` to [3,4,5,6] results in [6,3,4,5].

```
valarray<T> apply(T func(T)) const;
```

This function returns a new ***valarray*** object with the same length as the array in self but whose elements have all been initialized by applying the argument function `func` to the corresponding element in self (in other words, for all `n < *this.size()`, the `nth` element of the returned array equals `func(*this[n])`.

```
valarray<T> apply(T func(const T&)) const;
```

This function returns a new ***valarray*** object with the same length as the array in self but whose elements have all been initialized by applying the argument function `func` to the corresponding element in self (in other words, for all `0 <= n < *this.size()`, the `nth` element of the returned array equals `func(*this[n])`.

```
void
resize(size_type sz, T c = T());
```

Changes the length of self to `sz`, and assigns `c` to every element. This function also invalidates all outstanding references to self.

Non-member Binary Operators

```
template <class T> valarray<T>
operator*(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator/(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator%(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator+(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator-(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator^(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator&(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator|(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator<<(const valarray<T>& lhs, const valarray<T>& rhs);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T> valarray<T>
operator>>(const valarray<T>& lhs, const valarray<T>& rhs);
```

Returns a new ***valarray*** object of the same size as the argument arrays where each element has been initialized by applying the indicated operation to the corresponding elements in the argument arrays. The operation can only be applied to a ***valarray*** instantiated on a type `T` that supports a form of the indicated operation that returns `T` or a type convertible to `T`. The argument arrays must have the same length.

```
template <class T> valarray<T>
operator*(const valarray<T>& lhs, T& rhs);
template <class T> valarray<T>
operator/(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator%(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator+(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator-(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator^(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator&(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator|(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator<<(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<T>
operator>>(const valarray<T>& lhs, const T& rhs);
```

Returns a new ***valarray*** object of the same size as the ***valarray*** `lhs` where each element has been initialized by applying the indicated operation to the corresponding element in `lhs` and the value `rhs`. The operation can only be used with a type `T` that supports a form of the indicated operation that returns `T` or a type convertible to `T`.

```
template <class T> valarray<T>
operator*(const T& rhs, valarray<T>& rhs);
template <class T> valarray<T>
operator/(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator%(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator+(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator-(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator^(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator&(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator|(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator<<(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<T>
operator>>(const T& lhs, const valarray<T>& rhs);
```

Returns a new ***valarray*** object of the same size as the ***valarray*** `rhs` where each element has been initialized by applying the indicated operation to the corresponding element in `rhs` and the value `lhs`. The operation can only be used with a type `T` that supports a form of the indicated operation that returns `T` or a type convertible to `T`.

Non-member Logical Operators

```
template <class T> valarray<bool>
operator==(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator!=(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator<(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator>(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator<=(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator>=(const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator&&(const valarray<T>& lhs, const valarray<T>& rhs);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T> valarray<bool>
operator||(const valarray<T>& lhs, const valarray<T>& rhs);
```

Returns a ***valarray<bool>*** object of the same size as the argument arrays where each element has been initialized by applying the indicated operation to the corresponding elements in the argument arrays. The operation can only be applied to a ***valarray*** instantiated on a type `T` that support a form of the indicated operation that returns `bool` or a type convertible to `bool`. The argument arrays must have the same length.

```
template <class T> valarray<bool>
operator==(const valarray<T>& lhs, T& rhs);
template <class T> valarray<bool>
operator!=(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator<(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator>(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator<=(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator>=(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator&&(const valarray<T>& lhs, const T& rhs);
template <class T> valarray<bool>
operator||(const valarray<T>& lhs, const T& rhs);
```

Returns a ***valarray<bool>*** object of the same size as the ***valarray*** `lhs` where each element has been initialized by applying the indicated operation to the corresponding element in `lhs` and the value `rhs`. The operation can only be used with a type `T` that supports a form of the indicated operation that returns `bool` or a type convertible to `bool`.

```
template <class T> valarray<bool>
operator==(const T& rhs, valarray<T>& rhs);
template <class T> valarray<bool>
operator!=(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator<(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator>(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator<=(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator>=(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator&&(const T& lhs, const valarray<T>& rhs);
template <class T> valarray<bool>
operator||(const T& lhs, const valarray<T>& rhs);
```

Returns a ***valarray<bool>*** object of the same size as the valarray `rhs` where each element has been initialized by applying indicated operation to the corresponding element in `rhs` and the value `lhs`. The operation can only be used with a type `T` that supports a form of the indicated operation that returns `bool` or a type convertible to `bool`.

Non-member Transcendental Functions

```
template <class T> valarray<T> abs(const valarray<T>& v);
template <class T> valarray<T> acos(const valarray<T>& v);
template <class T> valarray<T> asin(const valarray<T>& v);
template <class T> valarray<T> atan(const valarray<T>& v);
template <class T> valarray<T> cos(const valarray<T>& v);
template <class T> valarray<T> cosh(const valarray<T>& v);
template <class T> valarray<T> exp(const valarray<T>& v);
template <class T> valarray<T> log(const valarray<T>& v);
template <class T> valarray<T> log10(const valarray<T>& v);
template <class T> valarray<T> sin(const valarray<T>& v);
template <class T> valarray<T> sinh(const valarray<T>& v);
template <class T> valarray<T> sqrt(const valarray<T>& v);
template <class T> valarray<T> tan(const valarray<T>& v);
template <class T> valarray<T> tanh(const valarray<T>& v);
```

Returns a new ***valarray*** object of the same size as the argument array where each element has been initialized by applying the indicated transcendental function to the corresponding elements in the argument array. The operation can only be applied to a ***valarray*** instantiated on a type `T` that supports a unique form of the indicated function that returns `T` or a type convertible to `T`.

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class T> valarray<T>
atan2(const valarray<T>& v, const valarray<T>& v2);
template <class T> valarray<T>
pow(const valarray<T>& v, const valarray<T>& v2);
```

Returns a new ***valarray*** object of the same size as the argument arrays where each element has been initialized by applying the indicated transcendental function to the corresponding elements in the argument arrays. The operation can only be applied to a ***valarray*** instantiated on a type `T` that supports a unique form of the indicated function that returns `T` or a type convertible to `T`.

```
template <class T> valarray<T>
atan2(const valarray<T>& v, const T& v2);
template <class T> valarray<T>
pow(const valarray<T>& v, const T& v2);
```

Returns a new ***valarray*** object of the same size as the argument array `v` where each element has been initialized by applying the indicated transcendental function to the corresponding elements in `v` along with the value `v2`. The operation can only be applied to a ***valarray*** instantiated on a type `T` that supports a unique form of the indicated function that returns `T` or a type convertible to `T`.

```
template <class T> valarray<T>
atan2(const T& v, const valarray<T> v2);
template <class T> valarray<T>
pow(const T& v, const valarray<T> v2);
```

Returns a new ***valarray*** object of the same size as the argument array `v2` where each element has been initialized by applying the indicated transcendental function to the corresponding elements in `v2` along with the value `v`. The operation can only be applied to a ***valarray*** instantiated on a type `T` that supports a unique form of the indicated function that returns `T` or a type convertible to `T`.

Example

```
//
// valarray.cpp
//
#include "valarray.h" // Contains a valarray stream inserter
using namespace std;
int main(void)
{
  int ibuf[10] = {0,1,2,3,4,5,6,7,8,9};
  int ibuf2[10] = {10,11,12,13,14,15,16,17,18,19};

  // create 2 valarrays of ints
  valarray<int>         vi(ibuf,10);
  valarray<int>         vi2(ibuf2,10);

  // print them out
  cout << vi << endl << vi2 << endl;

  vi += vi2;
  vi2 *= 2;
  valarry<int> vi3 = vi2 % vi;

  // print them out again
  cout << vi << endl << vi2 << endl << vi3 << endl;

  return 0;
}
```

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

vector

`Container`

Summary

A sequence that supports random access iterators.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

Member Functions

assign()	front()	operator==()
at()	get_allocator()	operator[]()
back()	insert()	pop_back()
begin()	max_size()	push_back()
capacity()	operator!=()	rbegin()
clear()	operator>()	rend()
empty()	operator>=()	reserve()
end()	operator<()	resize()
erase()	operator<=()	size()
flip()	operator=()	swap()

Synopsis

```
#include <vector>
template <class T, class Allocator = allocator<T> >
class vector ;
```

Description

vector<T, Allocator> is a type of sequence that supports random access iterators. In addition, it supports amortized constant time insert and erase operations at the end. Insert and erase in the middle take linear time. Storage management is handled automatically. In ***vector***, `iterator` is a random access iterator referring to `T`. `const_iterator` is a constant

©Copyright 1999 Rogue Wave Software, Inc.

random access iterator that returns a `const T&` when dereferenced. A constructor for `iterator` and `const_iterator` is guaranteed. `size_type` is an unsigned integral type. `difference_type` is a signed integral type.

Any type used for the template parameter `T` must provide the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
  Copy constructors     T(t) and T(u)
  Destructor            t.~T()
  Address of            &t and &u yielding T* and const T*
                        respectively
  Assignment            t = a where a is a
                        (possibly const) value of T
```

Special Case

Vectors of bit values, that is boolean 1/0 values, are handled as a special case by the standard library, so that they can be efficiently packed several elements to a word. The operations for a boolean vector, ***vector<bool>***, are a superset of those for an ordinary vector, only the implementation is more efficient.

Two member functions are available to the boolean vector data type. One is `flip()`, which inverts all the bits of the vector. Boolean vectors also return as reference an internal value that also supports the `flip()` member function. The other ***vector<bool>***-specific member function is a second form of the `swap()` function

Interface

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
// Types
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
class iterator;
class const_iterator;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename std::reverse_iterator<iterator>
                 reverse_iterator;
typedef typename std::reverse_iterator<const iterator>
                 const_reverse_iterator;

// Construct/Copy/Destroy
explicit vector (const Allocator& = Allocator());
explicit vector (size_type, const Allocator& = Allocator ());
vector (size_type, const T&, const Allocator& = Allocator());
vector (const vector<T, Allocator>&);
template <class InputIterator>
vector (InputIterator, InputIterator,
const Allocator& = Allocator ());
~vector ();
vector<T,Allocator>& operator= (const vector<T, Allocator>&);
template <class InputIterator>
void assign (InputIterator first, InputIterator last);
void assign (size_type, const);
allocator_type get_allocator () const;
// Iterators
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator rend ();
const_reverse_iterator rend () const;

// Capacity
size_type size () const;
size_type max_size () const;
void resize (size_type);
void resize (size_type, T);
size_type capacity () const;
bool empty () const;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
void reserve (size_type);

// Element Access
reference operator[] (size_type);
const_reference operator[] (size_type) const;
reference at (size_type);
const_reference at (size_type) const;
reference front ();
const_reference front () const;
reference back ();
const_reference back () const;

// Modifiers
void push_back (const T&);
void pop_back ();
iterator insert (iterator, const T&);
void insert (iterator, size_type, const T&);
template <class InputIterator>
void insert (iterator, InputIterator, InputIterator);
iterator erase (iterator);
iterator erase (iterator, iterator);
void swap (vector<T, Allocator>&);
void clear()
};

// Non-member Operators
template <class T>
bool operator== (const vector<T,Allocator>&,
const vector <T,Allocator>&);
template <class T>
bool operator!= (const vector<T,Allocator>&,
const vector <T,Allocator>&);
template <class T>
bool operator< (const vector<T,Allocator>&,
const vector<T,Allocator>&);
template <class T>
bool operator> (const vector<T,Allocator>&,
const vector<T,Allocator>&);
template <class T>
bool operator<= (const vector<T,Allocator>&,
const vector<T,Allocator>&);
template <class T>
bool operator>= (const vector<T,Allocator>&,
const vector<T,Allocator>&);

// Specialized Algorithms
template <class T, class Allocator>
void swap (const vector<T,Allocator>&, const vector<T,Allocator>&);
```

Constructors

```
explicit vector(const Allocator& alloc = Allocator());
```

The default constructor. Creates a vector of length zero. The vector will use the allocator `alloc` for all storage management.

```
explicit vector(size_type n);
```

Creates a vector of length `n`, containing `n` copies of the default value for type `T`. Requires that `T` have a default constructor. The vector will use the allocator `Allocator()` for all storage management.

```
vector(size_type n, const T& value,
const Allocator& alloc = Allocator());
```

Creates a vector of length `n`, containing `n` copies of value. The vector will use the allocator `alloc` for all storage management.

```
vector(const vector<T, Allocator>& x);
```

Creates a copy of `x`.

©Copyright 1999 Rogue Wave Software, Inc.

```
template <class InputIterator>
vector(InputIterator first, InputIterator last,
const Allocator& alloc = Allocator());
```

Creates a vector of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The vector will use the allocator `alloc` for all storage management.

Destructor

```
~vector();
```

The destructor. Releases any allocated memory for this vector.

Iterators

```
iterator
begin();
```

Returns a random access `iterator` that points to the first element.

```
const_iterator
begin() const;
```

Returns a random access `const_iterator` that points to the first element.

```
iterator
end();
```

Returns a random access `iterator` that points to the past-the-end value.

```
const_iterator
end() const;
```

Returns a random access `const_iterator` that points to the past-the-end value.

```
reverse_iterator
rbegin();
```

Returns a random access `reverse_iterator` that points to the past-the-end value.

```
const_reverse_iterator
rbegin() const;
```

Returns a random access `const_reverse_iterator` that points to the past-the-end value.

```
reverse_iterator
rend();
```

Returns a random access `reverse_iterator` that points to the first element.

```
const_reverse_iterator
rend() const;
```

Returns a random access `const_reverse_iterator` that points to the first element.

Assignment Operator

```
vector<T, Allocator>&
operator=(const vector<T, Allocator>& x);
```

Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to self.

Allocator

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

©Copyright 1999 Rogue Wave Software, Inc.

Reference Operators

```
reference
operator[](size_type n);
```

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the `size` less one.

```
const_reference
operator[](size_type n) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size` less one.

Member Functions

```
template <class InputIterator>
void
assign(InputIterator first, InputIterator last);
```

Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

```
void
assign(size_type, const T& t);
```

Erases all elements contained in self, then inserts `n` instances of the value of `t`.

```
reference
at(size_type n);
```

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the `size` less one.

```
const_reference
at(size_type) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size` less one.

```
reference
back();
```

Returns a reference to the last element.

```
const_reference
back() const;
```

Returns a constant reference to the last element.

```
size_type
capacity() const;
```

Returns the size of the allocated storage, as the number of elements that can be stored.

```
void
clear() ;
```

Deletes all elements from the vector.

```
bool
empty() const;
```

Returns `true` if the `size` is zero.

```
iterator
erase(iterator position);
```

Deletes the vector element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()` if the deleted element was the last one in this vector.

```
iterator
erase(iterator first, iterator last);
```

©Copyright 1999 Rogue Wave Software, Inc.

Deletes the vector elements in the range (first, last). Returns an `iterator` pointing to the element following the last deleted element, or `end()` if there were no elements in the deleted range.

```
void
flip();
```

Flips all the bits in the vector. *This member function is only defined for* ***vector<bool>***.

```
reference
front();
```

Returns a reference to the first element.

```
const_reference
front() const;
```

Returns a constant reference to the first element.

```
iterator
insert(iterator position, const T& x);
```

Inserts `x` before `position`. The return value points to the inserted `x`.

```
void
insert(iterator position, size_type n, const  T& x);
```

Inserts `n` copies of `x` before `position`.

```
template <class InputIterator>
void
insert(iterator position, InputIterator first,
InputIterator last);
```

Inserts copies of the elements in the range `[first, last]` before `position`.

```
size_type
max_size() const;
```

Returns `size()` of the largest possible vector.

```
void
pop_back();
```

Removes the last element of self.

```
void
push_back(const T& x);
```

Inserts a copy of `x` to the end of self.

```
void
reserve(size_type n);
```

Increases the capacity of self in anticipation of adding new elements. `reserve` itself does not add any new elements. After a call to `reserve`, `capacity()` is greater than or equal to `n` and subsequent insertions will not cause a reallocation until the size of the vector exceeds `n`. Reallocation does not occur if `n` is less than `capacity()`. If reallocation does occur, then all iterators and references pointing to elements in the vector are invalidated. `reserve` takes at most linear time in the size of self. `reserve` throws a `length_error` exception if `n` is greater than `max_size()`.

```
void
resize(size_type sz);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` instances of the default value of type `T` are inserted at the end of the vector. If the new size is smaller than the current `capacity`, then the vector is truncated by erasing `size()-sz` elements off the end. If `sz` is equal to `capacity` then no action is taken.

```
void
resize(size_type sz, T c);
```

©Copyright 1999 Rogue Wave Software, Inc.

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` `c`'s are inserted at the end of the vector. If the new size is smaller than the current `capacity`, then the vector is truncated by erasing `size()-sz` elements off the end. If `sz` is equal to `capacity` then no action is taken.

```
size_type
size() const;
```

Returns the number of elements.

```
void
swap(vector<T, Allocator>& x);
```

Exchanges self with `x`, by swapping all elements.

```
void
swap(reference x, reference y);
```

Swaps the values of `x` and `y`. *This is a member function of* ***vector<bool>*** *only*.

Non-member Operators

```
template <class T, class Allocator>
bool operator==(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool operator!=(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class T>
bool operator<(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

```
template <class T>
bool operator>(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `y < x`.

```
template <class T>
bool operator<=(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class T>
bool operator>=(const vector<T, Allocator>& x,
const vector<T, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class T, class Allocator>void
swap(vector <T, Allocator>& a, vector <T, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// vector.cpp
//
#include <vector>
```

©Copyright 1999 Rogue Wave Software, Inc.

```
#include <iostream>

ostream& operator<< (ostream& out,
const vector<int, allocator>& v)
 {
  copy(v.begin(),v.end(),ostream_iterator<int,char>(out," "));
  return out;
 }
int main(void)
 {
// create a vector of doubles
   vector<int>         vi;
   int                 i;

for(i = 0; i < 10; ++i) {
   // insert values before the beginning
   vi.insert(vi.begin(), i);
   }

// print out the vector
cout << vi << endl;

// now let's erase half of the elements
int half = vi.size() >> 1;
for(i = 0; i < half; ++i) {
   vi.erase(vi.begin());
   }

// print ir out again
cout << vi << endl;

return 0;
 }

Output :
9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Warnings

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for ***vector<T, Allocator>*** that takes two templated iterators:

```
template <class InputIterator>
vector (InputIterator, InputIterator,
const Allocator = Allocator());
```

vector also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a vector in the following two ways:

```
int intarray[10];
vector<int> first_vector(intarray, intarray + 10);
vector<int> second_vector(first_vector.begin(),
first_vector.end());
```

but not this way:

```
vector<long>
long_vector(first_vector.begin(),first_vector.end());
```

since the `long_vector` and `first_vector` are not the same type.

Additionally, if your compiler does not support default template parameters, you will need to supply the `Allocator` template argument. For instance, you will need to write :

```
vector<int, allocator<int> >
```

instead of :

©Copyright 1999 Rogue Wave Software, Inc.

```
vector<int>
```

See Also

allocator, *Containers*, *Iterators*, *lexicographical_compare*

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

wcerr

`Pre-defined stream`

Summary

Controls output to an unbuffered stream buffer associated with the object `stderr` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern wostream wcerr;
wostream wcerr;
```

Description

The object `wcerr` controls output to an unbuffered stream buffer associated with the object `stderr` declared in `<cstdio>`. By default the standard C and C++ streams are synchronized, but you can improve performance by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

`wcerr` uses the locale `codecvt` facet to convert the wide characters it receives to the tiny characters it outputs to `stderr`.

Formatting

The formatting is done through member functions or manipulators. See `cout`, `wcout`, or `basic_ostream` for details.

Example

```
//
// wcerr example
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  wifstream in("file_name.txt");

  // output the all file to stdout
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  if ( in )
    wcout << in.rdbuf();
  else
    // if the wifstream object is in a bad state
    // output an error message to stderr
    wcerr << L"Error while opening the file" << endl;
}
```

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cout***(3C++), ***cin***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wclog***(3C++), ***iomanip***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

wcin

`Pre-defined stream`

Summary

Controls input from a stream buffer associated with the object `stdin` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern wistream wcin;
wistream wcin;
```

Description

The object `wcin` controls input from a stream buffer associated with the object `stdin` declared in `<cstdio>`. By default the standard C and C++ streams are synchronized, but performance improvement can be achieved by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`, which implies that `wcin` and `wcout` are synchronized. `wcin` uses the locale `codecvt` facet to convert the tiny characters extracted from `stdin` to the wide characters stored in the `wcin` buffer.

Example

```
//
// wcin example #1
//
#include <iostream>

void main ( )
{
  using namespace std;

  int i;
  float f;
  wchar_t c;

  //read an integer, a float and a wide character from stdin
  wcin >> i >> f >> c;

  // output i, f and c to stdout
  wcout << i << endl << f << endl << c << endl;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

```
//
// wcin example #2
//
#include <iostream>

void main ( )
{
  using namespace std;

  wchar_t p[50];

  // remove all the white spaces
  wcin >> ws;

  // read characters from stdin until a newline
  // or 49 characters have been read
  wcin.getline(p,50);

  // output the result to stdout
  wcout << p;
}
```

When inputting `" Grendel the monster" (newline)` in the previous test, the output is: `"Grendel the monster"`. The manipulator `ws` removes spaces.

See Also

basic_istream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cin***(3C++), ***cout***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***wclog***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

wclog

`Pre-defined stream`

Summary

Controls output to a stream buffer associated with the object `stderr` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern wostream wclog;
wostream wclog;
```

Description

The object `wclog` controls output to a stream buffer associated with the object `stderr` declared in `<cstdio>`. The difference between `wclog` and `wcerr` is that `wclog` is buffered, but `wcerr` isn't. Therefore, commands like `wclog << L"ERROR !!";` and `fprintf(stderr,"ERROR !!");` are not synchronized. `wclog` uses the locale `codecvt` facet to convert the wide characters it receives to the tiny characters it outputs to `stderr`.

Formatting

The formatting is done through member functions or manipulators. See ***cout***, ***wcout*** or ***basic_ostream*** for details.

Example

```
//
// wclog example
//
#include<iostream>
#include<fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  wifstream in("file_name.txt");
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // output the all file to stdout
  if ( in )
    wcout << in.rdbuf();
  else
    // if the wifstream object is in a bad state
    // output an error message to stderr
    wclog << L"Error while opening the file" << endl;
}
```

Warnings

`wclog` can be used to redirect some of the errors to another recipient. For example, you might want to redirect them to a file named `my_err`:

```
wofstream out("my_err");
if ( out )
  wclog.rdbuf(out.rdbuf());
else
  cerr << "Error while opening the file" << endl;
```

Then when you are doing something like `wclog << L"error number x";` the error message is output to the file `my_err`. You can use the same scheme to redirect `wclog` to other devices.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cout***(3C++), ***cin***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcout***(3C++), ***wcerr***(3C++), ***iomanip***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

wcout

`Pre-defined stream`

Summary

Controls output to a stream buffer associated with the object `stdout` declared in `<cstdio>`.

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

Synopsis

```
#include <iostream>
extern wostream wcout;
wostream wcout;
```

Description

The object `wcout` controls output to a stream buffer associated with the object `stdout` declared in `<cstdio>`. By default the standard C and C++ streams are synchronized, but performance can be improved by using the `ios_base` member function `synch_with_stdio` to desynchronize them.

After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`, which implies that `wcin` and `wcout` are synchronized. `wcout` uses the locale `codecvt` facet to convert the wide characters it receives to the tiny characters it outputs to `stdout`.

Formatting

The formatting is done through member functions or manipulators.

Manipulators	Member Functions
`showpos`	`setf(ios_base::showpos)`
`noshowpos`	`unsetf(ios_base::showpos)`
`showbase`	`setf(ios_base::showbase)`
`noshowbase`	`unsetf(ios_base::showbase)`
`uppercase`	`setf(ios_base::uppercase)`
`nouppercase`	`unsetf(ios_base::uppercase)`
`showpoint`	`setf(ios_base::showpoint)`

©Copyright 1999 Rogue Wave Software, Inc.

noshowpoint	unsetf(ios_base::showpoint)
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unsetf(ios_base::boolalpha)
unitbuf	setf(ios_base::unitbuf)
nounitbuf	unsetf(ios_base::unitbuf)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)
resetiosflags (ios_base::fmtflags flag)	setf(0,flag)
setiosflags (ios_base::fmtflags flag)	setf(flag)
setbase (int base)	see above
setfill(char_type c)	fill(c)
setprecision (int n)	precision(n)
setw (int n)	width(n)
endl	
ends	
flush	flush( )

Description

showpos	Generates a + sign in non-negative generated numeric output.
showbase	Generates a prefix indicating the numeric base of generated integer output
uppercase	Replaces certain lowercase letters with their uppercase equivalents in generated output
showpoint	Generates a decimal-point character unconditionally in generated floating-point output
boolalpha	Inserts and extracts bool type in alphabetic format
unitbuf	Flushes output after each output operation
internal	Adds fill characters at a designated internal point in certain generated output. If no such point is designated, it's identical to `right`.
left	Adds fill characters on the right (final positions) of certain generated output
right	Adds fill characters on the left (initial positions) of certain generated output
dec	Converts integer input or generates integer output in decimal base
hex	Converts integer input or generates integer output in hexadecimal base
oct	Converts integer input or generates integer output in octal base
fixed	Generates floating-point output in fixed-point notation
scientific	Generates floating-point output in scientific notation
resetiosflags (ios_base::fmtflags flag)	Resets the `fmtflags` field `flag`
setiosflags (ios_base::fmtflags	Sets up the flag `flag`

©Copyright 1999 Rogue Wave Software, Inc.

flag)

setbase(int base)	Converts integer input or generates integer output in base base. The parameter base can be 8, 10 or 16.
setfill(char_type c)	Sets the character used to pad (fill) an output conversion to the specified field width
setprecision(int n)	Sets the precision (number of digits after the decimal point) to generate on certain output conversions
setw(int n)	Sets the field with (number of characters) to generate on certain output conversions
endl	Inserts a newline character into the output sequence and flush the output buffer.
ends	Inserts a null character into the output sequence.
flush	Flush the output buffer.

Default Values

```
precision()           6
width()               0
fill()                the space character
flags()               skipws | dec
getloc()              locale::locale()
```

Example

```
//
// wcout example #1
//
#include<iostream>
#include<iomanip>

void main ( )
{
  using namespace std;

  int i;
  float f;

  // read an integer and a float from stdin
  cin >> i >> f;

  // output the integer and goes at the line
  wcout << i << endl;

  // output the float and goes at the line
  wcout << f << endl;

  // output i in hexa
  wcout << hex << i << endl;

  // output i in octal and then in decimal
  wcout << oct << i << dec << i << endl;

  // output i preceded by its sign
  wcout << showpos << i << endl;

  // output i in hexa
  wcout << setbase(16) << i << endl;

  // output i in dec and pad to the left with character
  // @ until a width of 20
  // if you input 45 it outputs 45@@@@@@@@@@@@@@@@@@
  wcout << setfill(L'@') << setw(20) << left << dec << i;
  wcout << endl;

  // output the same result as the code just above
  // but uses member functions rather than manipulators
  wcout.fill('@');
  wcout.width(20);
  wcout.setf(ios_base::left, ios_base::adjustfield);
  wcout.setf(ios_base::dec, ios_base::basefield);
  wcout << i << endl;

  // outputs f in scientific notation with
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  // a precision of 10 digits
  wcout << scientific << setprecision(10) << f << endl;

  // change the precision to 6 digits
  // equivalents to wcout << setprecision(6);
  wcout.precision(6);

  // output f and goes back to fixed notation
  wcout << f << fixed << endl;

}

//
// wcout example #2
//
#include <iostream>

void main ( )
{
  using namespace std;

  wchar_t p[50];

  wcin.getline(p,50);

  wcout << p;
}

//
// wcout example #3
//
#include <iostream>
#include <fstream>

void main ( )
{
  using namespace std;

  // open the file "file_name.txt"
  // for reading
  wifstream in("file_name.txt");

  // output the all file to stdout
  if ( in )
    wcout << in.rdbuf();
  else
    {
      wcout << "Error while opening the file";
      wcout << endl;
    }
}
```

Warnings

Keep in mind that the manipulator `endl` flushes the stream buffer. Therefore, use `L`\n'` if your only intent is to go at the line. It greatly improves performance when C and C++ streams are not synchronized.

If your compiler does not support namespaces, then you do not need the using declaration for `std`.

See Also

basic_ostream(3C++), ***iostream***(3C++), ***basic_filebuf***(3C++), ***cin***(3C++), ***cout***(3C++), ***cerr***(3C++), ***clog***(3C++), ***wcin***(3C++), ***wcerr***(3C++), ***wclog***(3C++), ***iomanip***(3C++), ***ios_base***(3C++), ***basic_ios***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 27.3.2

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

wstring

`String Library`

Summary

A typedef for:

```
basic_string<wchar_t, char_traits<wchar_t>,
            allocator<wchar_t> >
```

Data Type and Member Function Indexes (exclusive of constructors and destructors)

None

For more information about strings, see the entry ***basic_string***.

Top Contents Index

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

Top Contents

Endnotes

1

The notation is similar to the notation used in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides.

Return

©Copyright 1999, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
OEM Release

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.

If you are accessing this for the first time, please read the licensing statement.

Standard C++ Library User's Guide - OEM Edition

Welcome to the Standard C++ Library User's Guide. The User's Guide has been divided into these two volumes:

General User's Guide
Locales and Iostreams User's Guide

The Standard C++ Library also has a Class Reference.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.
Documentation Tips

If you are accessing this for the first time, please read the licensing statement.

Standard C++ Library General User's Guide - OEM Edition

Welcome to the Standard C++ Library General User's Guide. The Standard C++ Library documentation includes these companion volumes:

Locales and Iostreams User's Guide
Class Reference

In this document, you can view a

Comprehensive Table of Contents

showing all chapters, first- and second-level headings, or click on one of the chapter names below to go directly to that chapter. Each chapter begins with a chapter-level table of contents.

There is also a topic index.

Chapters

©Copyright 1999 Rogue Wave Software, Inc.

Part IV: Algorithms

Part V: Special Techniques

Part VI: Special Classes

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part I: Introduction

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 1: Overview

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Welcome

Congratulations on choosing the Rogue Wave implementation of the Standard C++ Library. You can use this product with confidence: it is based on the final standard for the C++ language and library ratified in March, 1998, by the American National Standards Institute (ANSI) and the International Standards Organization (ISO).

Since its development by Dr. Bjarne Stroustrup in the 80s, the C++ language has been widely used by professional programmers for the world's big, complex applications in telecommunications, finance, business, embedded systems, and computer-aided design. The final standardization of the C++ library now makes it easier to learn C++ and to use it across a wide variety of platforms.

Standardization improves portability and stability. You can build reliable applications faster, and maintain them with less cost and effort, using the Standard C++ Library.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Product Overview

Components

The Standard C++ Library is a large and comprehensive collection of classes and functions for fine-grained, low-level programming. Within this library, you will find the following components:

- The large set of data structures and algorithms formerly known as the Standard Template Library (STL)
- An iostream facility
- A locale facility
- A templatized ***string*** class
- A templatized ***complex*** class for representing complex numbers
- A ***valarray*** class optimized for handling numeric arrays
- A uniform framework for describing the execution environment, using a template class named ***numeric_limits*** and specializations for each fundamental datatype
- Memory management features
- Extensive support for national character sets
- Exception handling features

Unique Features

The STL portion of the Standard C++ Library is not object-oriented. If you are accustomed to the benefits of object-oriented programming, their absence may necessitate some adjustment. Encapsulation of data and functionality in objects is a hallmark of object-oriented programming. In the Standard C++ Library, however, the data structures are separate from the algorithms you use to manipulate them.

This feature can provide a number of advantages, such as smaller source code, and the flexibility of using algorithms with C++ pointers and arrays as well as conventional objects. It can also lead to more efficient coding and faster execution, since it creates a direct, nuts-and-bolts approach to solving problems.

The main disadvantage of using the Standard C++ Library directly is increased risk of error. For example, the library's iterators must not be mismatched or invalidated, and iterators in multithreaded environments should be wrapped before being shared among threads. The templates can cause less precise diagnostics, and code that grows unexpectedly large. Experience with the library and your own compiler will help diminish these problems.

Relationship to Tools.h++

This Rogue Wave implementation of the Standard C++ Library is certified for use with **Tools.h++**. The Rogue Wave product **Tools.h++**, version 7.0 and later, encapsulates the Standard C++ Library with an object-oriented interface. Used together, **Tools.h++** and the **Standard C++ Library** are designed to provide the benefits of both low-level generic programming and object-orientation.

Tools.h++ also contains features not included in the standard, like time, date, and regular expression classes, enhanced strings, object persistence, and virtual streams. The new product **Tools.h++ Professional** extends the functionality of the original **Tools.h++** with modules on Java interoperability, networks, and CORBA. For many programming tasks, you may find it easier and more convenient to use **Tools.h++** and **Tools.h++ Professional** without accessing the Standard C++ Library directly.

Compatibility Issues

©Copyright 1999 Rogue Wave Software, Inc.

This implementation of the Standard C++ Library is largely compatible with previous 2.x implementations, and largely incompatible with 1.x versions.

Please note that some compiler and library vendors have not yet implemented the whole range of language features defined by the ANSI/ISO standard. In the *User's Guide* and *Locales and Iostreams*, we draw your attention to places in the code where this might be a problem. If your vendor is still in the process of meeting the standard, some of the techniques demonstrated in this *User's Guide* may not work. We include them anyway to demonstrate the full range of capabilities of the Standard C++ language. Compilers will catch up, and this guide will be more useful to you if it is as complete as possible.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Documentation Overview

You are reading the *Standard C++ Library User's Guide*, part of the complete documentation available for this Rogue Wave implementation of the Standard C++ Library. This documentation is summarized in Table 1:

Table 1 -- Documentation for Rogue Wave's Standard C++ Library

Documentation format	Description
The Standard C++ Library User's Guide	Covers all the essentials, including iterators, containers, algorithms, allocators, and error handling, and explanations of the tutorials. Online version ships with product; hard copy available for purchase.
The Standard C++ Library Locales and Iostreams	Covers the familiar iostreams and the newer implementation of locales; basics of internationalization. Online version ships with product; hard copy available for purchase.
The Standard C++ Library Class Reference	An alphabetical listing of all classes, algorithms, and function objects. Online version ships with product; hard copy available for purchase.
Readme files	Information on specific compilers and operating systems, how to use shared libraries and DLLs, solutions to common problems, and late-breaking product news.
Tutorials	Examples you can use to learn about the Standard C++ Library and to write your own applications.
man pages	For Unix platforms only.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Overview of This Manual

Assumptions

This manual is *The Standard C++ Library User's Guide*, an introduction to the Rogue Wave implementation of the Standard C++ Library. It assumes that you are familiar with the basic features of the C++ programming language. If you are new to C++, you may want to start with an introductory text, such as *The C++ Programming Language* by Bjarne Stroustrup (Addison-Wesley, 1997).

Everything in the Standard C++ Library is contained in the std namespace. You must specify a using declaration or explicitly scope any name from the library; for example, `using std::vector;` or `std::vector<int> v;`. This is assumed in the examples.

Conventions

This manual uses some distinctive terms and conventions. The Standard C++ Library consists mostly of class and function templates, so abbreviations for these templates are common. For example, in the iostreams part of the documentation, `fstream` stands for `template <class charT, class traits> class basic_fstream`. A slightly more succinct notation for a class template is also frequently used: `basic_fstream <charT, traits>`. *File stream* stands for the abstract notion of the file stream class template; `badbit` stands for the state flag `ios_base::badbit`.

The term *algorithm* indicates functions in the generic algorithms portion of the Standard C++ Library, so as to avoid confusion with member functions, argument functions, and user-defined functions.

An empty pair of parentheses `()` follows function names and algorithm names, so as to avoid emphasizing their arguments. An underline character _ is used as a separator in both class names and function names.

Special fonts set off class names, code samples, and special meanings, as shown in Table 2:.

Table 2 -- Typographic conventions

Convention	Purpose	Example
`Courier`	Code, examples, function names, file names, directory names, operating system commands	`return result;`
italic	Emphasis. New terms. Titles.	operating system *family* special *ending* iterator *User's Guide*
bold	Emphasis. Commands from an interface. Rogue Wave product names.	do this **before** that the **OK** button a **Standard C++ Library** file
bold italic	Class names.	***priority_queue***

Organization

This manual is organized into six parts:

- **Part I Introduction** provides overviews of the product, the documentation that explains the product, and this manual, *The Standard C++ Library User's Guide*.
- **Part II Fundamentals** explains the fundamental Standard C++ Library concepts of iterators, functions, and predicates.
- The product overview noted that data structures are separated from algorithms in the Standard C++ Library.

©Copyright 1999 Rogue Wave Software, Inc.

In this manual, **Part III Containers** gives an overview of the data structures or *containers*, and devotes a chapter to describing each in detail with an example. **Part IV Algorithms** deals with the algorithms, devoting one chapter to the generic algorithms, and another to the generic algorithms specific to ordered collections. Within the algorithm chapters, each kind of algorithm is explained in its own section with an example and a reference to the file containing the complete source code.

- **Part V Special Techniques** describes techniques such as using allocators, building containers and generic algorithms, using the traits technique, and dealing with exceptions.
- **Part VI Special Classes** devotes a chapter to each of the four unique classes ***auto_ptr***, ***complex***, ***numeric_limits***, and ***valarray***.

Reading Suggestion

The first time you read this manual it may be hard to know where to start. The container definitions form the heart of the library, but you can't really appreciate them without understanding the algorithms that so greatly extend their functionality. On the other hand, you can't really understand the algorithms without some appreciation of the containers.

A good approach is to read **Part II Fundamentals** carefully. Next, skim the definitions of the containers in **Part III Containers** and the descriptions of the algorithms in **Part IV Algorithms**, then go back and read these parts in more detail. You can then proceed to the more specialized areas of the manual.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Index

Table of Contents

©Copyright 1999 Rogue Wave Software, Inc.

Part III: Containers

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Part IV: Algorithms

Chapter 13: Generic Algorithms

Chapter 14: Ordered Collection Algorithms

©Copyright 1999 Rogue Wave Software, Inc.

Part V: Special Techniques

Chapter 15: Using Allocators

Chapter 16: Building Containers and Generic Algorithms

Chapter 17: The Traits Parameter

Chapter 18: Exception Handling

Part VI: Special Classes

Chapter 19: auto_ptr

Chapter 20: complex

©Copyright 1999 Rogue Wave Software, Inc.

Top Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top

©Copyright 1998, Rogue Wave Software, Inc.

Use your browser's **Back** button to return to where you were.

Feedback on the Documentation

Write to us at onlinedocs@roguewave.com

Please use this only to:

1. Report errors in the documentation.
2. Make suggestions on how to improve the documentation.
3. Tell us how much you like the documentation.

Support Issues

The Rogue Wave support Web site at support.roguewave.com offers several ways to contact Rogue Wave technical support. The Web site includes an ever-expanding, searchable KnowledgeBase that can answer many support questions immediately.

You may also write to us at support@roguewave.com.

Telephone support is available at (303) 545 3205.

Top

©Copyright 1999 Rogue Wave Software, Inc.

Top

©Copyright 1998, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
Banner | Buttons | Comments | User's Guide Features | Reference Guide Features | Want Books

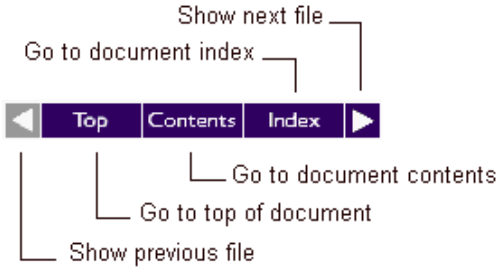
Tips on Using Rogue Wave Documentation

The banner

You can always click on the **Rogue Wave Online Documentation** banner to go to the master page for the Standard C++ Library User's Guide.

The buttons

The buttons do the following:



Use the **Show previous/next file** buttons to move sequentially through a document. A grayed-out button is inactive.

Contacting Rogue Wave

Don't like something? Or, more charitably, would you like make a suggestion or point out an error? Use the **Contact ...** link at the top of the first page and the bottom of all subsequent pages to view a contact page. This page provides a mailto link to the documentation team at Rogue Wave.

The contact page also provides ways to contact technical support. Please use these contact points and **not** the documentation mailto link for technical issues.

User's guide navigation features

The user's guides (also build guides, platform guides, and installation guides) have these navigation features:

- A chapter-name table of contents on the opening page, providing links to the manual's major topics.
- A section-name table of contents at the beginning of each chapter, providing links to the chapter's detailed topics.
- A comprehensive table of contents in case you want to see everything. The **Contents** button leads here.
- Usually, a topic index based on the index markup in the print document. The links are based on the name of the section to which the link leads to assist you in deciding which link to follow.

Reference guide navigation features

The reference guides have these navigation features:

- Links to the major sections on the opening page, with the final link being to the section of class descriptions.
- A hypertext list of class descriptions. The **Contents** button leads here.
- A section-name table of contents at the beginning of each class description, providing links to the categories of descriptions.
- A per-class method/data type hypertext index at the beginning of each class description.
- A comprehensive hypertext index of methods and data types. The links are based on the name of the class description to which the link leads, so for methods or data types that appear in multiple classes you can easily

©Copyright 1999 Rogue Wave Software, Inc.

select the link to class in which you are interested.

want books

If you prefer books, these may be purchased from Rogue Wave. Contact Rogue Wave by:

Telephone:(303) 473-9118 or (800) 487-3217

Email: sales@roguewave.com

WWW: http://www.roguewave.com

Top

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part II: Fundamentals

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 2: Iterators

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Introduction to Iterators

The concept of iterators is fundamental to using the container classes and the associated algorithms provided by the Standard C++ Library. An *iterator* is a pointer-like object used to cycle through all the elements stored in a container. Because different algorithms need to traverse containers in a variety of fashions, there are different forms of iterators. Each container class in the Standard C++ Library can generate an iterator with functionality appropriate to the storage technique used in implementing the container. It is the category of iterators required as arguments that chiefly distinguishes which algorithms in the Standard C++ Library can be used with which container classes.

Just as pointers can be used in a variety of ways in traditional programming, iterators can be used for a number of different purposes. An iterator can be used to denote a specific value, just as a pointer can be used to reference a specific memory location. A *pair* of iterators can be used to define a *range* or sequence of values held in a container, just as two pointers can be used to describe a contiguous region of memory. With iterators, however, the values being described may be only logically rather than physically in sequence because they are derived from the same container, and the second follows the first in the order in which the elements are maintained by the container.

Conventional pointers can sometimes be *null*, meaning they point at nothing. Iterators, as well, can fail to denote any specific value. Just as it is a logical error to dereference a null pointer, it is an error to dereference an iterator that is not denoting a value.

When two pointers that describe a region in memory are used in a C++ program, it is conventional that the ending pointer *not* be considered part of the region. For example, an array named `x` of length `10` is sometimes described as extending from `x` to `x+10`, even though the element at `x+10` is not part of the array. Instead, the pointer value `x+10` is the *past-the-end* value *after* the end of the range being described. Iterators are used similarly to describe a range. The second value is not considered part of the range being denoted. Instead, the second value is a *past-the-end* element describing the next value in sequence after the final value of the range. Sometimes, as with pointers to memory, this will be an actual value in the container. Other times it may be a special value, specifically constructed for the purpose. In either case, it is not proper to dereference an iterator that is being used to specify the end of a range.

Just as with conventional pointers, the fundamental operation used to modify an iterator is the increment operator `++`. When the increment operator is applied to an iterator that denotes the final value in a sequence, it is changed to the past-the-end value. An iterator `j` is said to be *reachable* from an iterator `i` if, after a finite sequence of applications of the expression `++i`, the iterator `i` becomes equal to `j`.

Ranges can be used to describe the entire contents of a container by constructing an iterator to the initial element and a special *ending* iterator. Ranges can also be used to describe sub-sequences within a single container by employing two iterators to specific values.

NOTE: Whenever two iterators are used to describe a range, it is assumed that the second iterator is reachable from the first, but this is not verified. Errors can occur if this expectation is not satisfied.

In the remainder of this chapter, we describe the different forms of iterators used by the Standard C++ Library, as well as various other iterator-related functions.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Varieties of Iterators

As shown in Table 3, there are five basic forms of iterators used in the Standard C++ Library:

Table 3 -- Iterator forms in the Standard C++ Library

Iterator form	Description
input iterator	Read only, forward moving
output iterator	Write only, forward moving
forward iterator	Both read and write, forward moving
bidirectional iterator	Read and write, forward and backward moving
random access iterator	Read and write, random access

Iterator categories are hierarchical. *Forward iterators* can be used wherever input or output iterators are required, *bidirectional iterators* can be used in place of forward iterators, and *random access iterators* can be used in situations requiring bidirectionality.

A second characteristic of iterators is whether or not they can be used to modify the values held by their associated container. A *constant iterator* is one that can be used for access only, and cannot be used for modification. *Output iterators* are never constant, and *input* iterators always are. Other iterators may or may not be constant, depending upon how they are created. There are both constant and non-constant bidirectional iterators, both constant and non-constant random access iterators, and so on.

Table 4 summarizes specific ways that various categories of iterators are generated by the containers in the Standard C++ Library.

Table 4 -- Iterators generated in the Standard C++ Library

Iterator form	Produced by
input iterator	`istream_iterator`

output iterator	`ostream_iterator`
	`inserter`
	`front_inserter`
	`back_inserter`

bidirectional iterator	***list*** ***set*** and ***multiset*** ***map*** and ***multimap***
random access iterator	ordinary pointers ***vector*** ***deque***

In the following sections we describe the capabilities and construction of each form of iterator.

Input Iterators

Input iterators are the simplest form of iterator. To understand their capabilities, consider an example program:

```
template <class InputIterator, class T>
InputIterator
   find (InputIterator first, InputIterator last, const T& value)
{
   while (first != last && *first != value)
      ++first;
   return first;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

In this program, the `find()` generic algorithm (Section 13.3.1) performs a simple linear search, looking for a specific value being held within a container. The contents of the container are described using two iterators, `first` and `last`. While `first` is not equal to `last`, the element denoted by `first` is compared to the test value. If this element is equal to the test value, the iterator, which now denotes the located element, is returned. If it is not equal, the `first` iterator is incremented and the loop cycles once more. If the entire region of memory is examined without finding the desired value, then the algorithm returns the end-of-range iterator.

This algorithm illustrates three features of an *input iterator*:

- An input iterator can be compared for equality to another iterator. They are equal when they point to the same position, and not equal otherwise.
- An input iterator can be dereferenced, using the operator `*` to obtain the value being denoted by the iterator.
- An input iterator can be incremented, so that it refers to the next element in sequence, using the operator `++`.

Notice that these features can all be provided with new meanings in a C++ program, since the behavior of the given functions can all be modified by overloading the appropriate operators. Because of this overloading, iterators are possible.

Kinds of Input Iterators

There are three main kinds of input iterators: ordinary pointers, container iterators, and input streams iterators.

Ordinary pointers. Ordinary pointers can be used as input iterators. In fact, since we can subscript and add to ordinary pointers, they are random access values, and thus can be used either as input or output iterators. The end-of-range pointer describes the end of a contiguous region of memory, and the dereference and increment operators have their conventional meanings. For example, the following searches for the value 7 in an array of integers:

```
int data[100];
   ...
int * where = find(data, data+100, 7);
```

Note that constant pointers, which do not permit the underlying array to be modified, can be created by simply placing the keyword `const` in a declaration:

```
const int * first = data;
const int * last = data + 100;
   // can't modify location returned by the following
const int * where = find(first, last, 7);
```

Because ordinary pointers have the same functionality as random access iterators, most of the generic algorithms in the Standard C++ Library can be used with conventional C++ arrays, as well as with the containers provided by the Standard C++ Library.

Container iterators. All of the iterators constructed for the various containers provided by the Standard C++ Library are at *least* as general as input iterators. The iterator for the first element in a collection is always constructed by the member function `begin()`, while the iterator that denotes the past-the-end location is generated by the member function `end()`. For example, the following iterator searches for the value 7 in a list of integers:

```
list<int>::iterator where = find(aList.begin(), aList.end(), 7);
```

Each container that supports iterators provides a type with the name `iterator` within the class declaration. Using this type, iterators can uniformly be declared in the fashion shown. If the container being accessed is constant, or if the description `const_iterator` is used, then the iterator is a constant iterator.

Input stream iterators. The Standard C++ Library provides a mechanism to operate on an input stream using an input iterator. This ability is provided by the class `istream_iterator`, described in more detail in Section 2.3.1.

Output Iterators

An *output iterator* has the opposite function of an input iterator. Output iterators can be used to assign values in a sequence, but cannot be used to access values. For example, we can use an output iterator in a generic algorithm that copies values from one sequence into another:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   (InputIterator first, InputIterator last, OutputIterator result)
{
    while (first != last)
      *result++ = *first++;
    return result;
}
```

A number of the generic algorithms manipulate two parallel sequences. Frequently the second sequence is described using only a beginning iterator, rather than an iterator pair. It is assumed, but not checked, that the second sequence has at least as many elements as the first.

In the algorithm shown here, two ranges are being manipulated: the range of source values specified by a pair of input iterators, and the destination range. The latter, however, is specified by only a single argument. It is assumed that the destination is large enough to include all values, and errors will ensue if this is not the case.

As illustrated by this algorithm, an output iterator can modify the element to which it points by being used as the target for an assignment. Output iterators can use the dereference operator only in this fashion; they cannot be used to return or to access the elements they denote.

As we noted earlier, ordinary pointers, like all iterators constructed by containers in the Standard C++ Library, can be used as output iterators. (Ordinary pointers are random access iterators, which are a superset of output iterators.) For example, in this code fragment elements from an ordinary C-style array are copied into a Standard C++ Library vector:

```
int data[100];
vector<int> newdata(100);
   ...
copy (data, data+100, newdata.begin());
```

Just as the `istream_iterator` provides a way to operate on an input stream using the input iterator mechanism, the Standard C++ Library provides a datatype, `ostream_iterator`, that permits values to be written to an output stream in an iterator-like fashion (Section 2.3.2).

Yet another form of output iterator is an *insert iterator* (Section 2.4). An insert iterator changes the output iterator operations of dereferencing/assignment and increment into insertions into a container. This permits operations such as `copy()` to be used with variable length containers, such as ***list***s and ***set***s.

Forward Iterators

A *forward iterator* combines the features of an input iterator and an output iterator. It permits values to be both accessed and modified. One function that uses forward iterators is the `replace()` generic algorithm, which replaces occurrences of specific values with other values. This algorithm is written as follows:

```
template <class ForwardIterator, class T>
void
   replace (ForwardIterator first, ForwardIterator last,
            const T& old_value, const T& new_value)
{
    while (first != last)
    {
    if (*first == old_value)
        *first = new_value;
        ++first;
    }
}
```

Ordinary pointers, like all iterators produced by containers in the Standard C++ Library, can be used as forward iterators. For example, in the following code instances of the value 7 are replaced with the value 11 in a vector of integers:

```
replace (aVec.begin(), aVec.end(), 7, 11);
```

Bidirectional Iterators

Bidirectional iterators are similar to forward iterators, except that bidirectional iterators support the decrement operator --, permitting movement in either a forward or a backward direction through the elements of a container. For example, we can use bidirectional iterators in a function that reverses the values of a container, placing the results into a new container:

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator
   reverse_copy (BidirectionalIterator first,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                BidirectionalIterator last,
                OutputIterator result)
{
    while (first != last)
    *result++ = *--last;
    return result;
}
```

As always, the value initially denoted by the `last` argument is not considered part of the collection.

The `reverse_copy()` function could be used, for example, to reverse the values of a linked list, and place the result into a vector:

```
list<int> aList;
 ....
vector<int> aVec (aList.size());
reverse_copy (aList.begin(), aList.end(), aVec.begin() );
```

Random Access Iterators

Some algorithms require more functionality than simply accessing values in either a forward or backward direction. Random access iterators permit values to be accessed by subscript, subtracted one from another (to yield the number of elements between their respective values), or modified by arithmetic operations, all in a manner similar to conventional pointers.

With conventional pointers, arithmetic operations can be related to the underlying memory; that is, `x+10` is the memory ten elements after the beginning of `x`. With iterators the logical meaning is preserved (`x+10` is the tenth element after `x`), however different the physical addresses being described.

Algorithms that use random access iterators include generic operations like sorting and binary search. For example, the following algorithm randomly shuffles the elements of a container. This is similar to, although simpler than, the function `random_shuffle()` provided by the Standard C++ Library.

```
template <class RandomAccessIterator>
void
   mixup (RandomAccessIterator first, RandomAccessIterator last)
{
   while (first < last)
   {
      iter_swap(first, first + randomInteger(last - first));
      ++first;
   }
}
```

NOTE: The function randomInteger described here appears in a number of the example programs presented in later sections.

The program will cycle as long as `first` denotes a position that occurs earlier in the sequence than the one denoted by `last`. Only random access iterators can be compared using relational operators; all other iterators can be compared only for equality or inequality. On each cycle through the loop, the expression
`last - first` yields the number of elements between the two limits. The function `randomInteger()` is assumed to generate a random number between 0 and the argument. Using the standard random number generator, this function could be written as follows:

```
unsigned int randomInteger (unsigned int n)
   // return random integer greater than
   // or equal to 0 and less than n
{
   return rand() % n;
}
```

This random value is added to the iterator `first`, resulting in an iterator to a randomly selected value in the container. This value is then swapped with the element denoted by the iterator `first`.

Reverse Iterators

An iterator naturally imposes an order on an underlying container of values. For a ***vector*** or a ***map***, the order is imposed by increasing index values; for a ***set***, by the increasing order of the elements held in the container. For a ***list***, the order is explicitly derived from the way values are inserted.

©Copyright 1999 Rogue Wave Software, Inc.

A *reverse iterator* yields values in exactly the reverse order of values given by the standard iterators. For a vector or a list, a reverse iterator generates the last element first, and the first element last. For a set it generates the largest element first, and the smallest element last. Strictly speaking, reverse iterators do not constitute a new category of iterator, but an adaptation of another iterator type. Consequently, we have reverse bidirectional iterators and reverse random access iterators. Any bidirectional or random access iterator can be adapted by the `reverse_iterator` template.

The ***list***, ***set***, and ***map*** datatypes provide a pair of member functions that produce reverse bidirectional iterators. The functions `rbegin()` and `rend()` generate iterators that cycle through the underlying container in reverse order. Increments to such iterators move backward, and decrements move forward through the sequence.

Similarly, the ***vector*** and ***deque*** datatypes provide functions, also named `rbegin()` and `rend()`, that produce reverse random access iterators. Subscript and addition operators, as well as increments to such iterators, move backward within the sequence.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Stream Iterators

Stream iterators are used to access an existing input or output stream using iterator operations. An input stream iterator permits an input stream to be read using iterator operations. Similarly, an output stream iterator writes to an output stream when iterator operations are executed.

Input Stream Iterators

As we noted in the discussion of input iterators, the Standard C++ Library provides a mechanism for turning an input stream into an input iterator through class `istream_iterator`. When declared, the four template arguments are the element type, the stream character type, the character traits type, and a type that measures the distance between elements. The latter two default to `char_traits<charT >` and `ptrdiff_t`. The default is almost always the appropriate behavior. The single argument provided to the constructor for an `istream_iterator` is the stream to be accessed. Each time the `++` operator is invoked on an input stream iterator, a new value from the stream is read (using the operator `>>`) and stored. This value is then available through the use of the dereference operator `*`. The value constructed by `istream_iterator` when no arguments are provided to the constructor can be used as an ending iterator value. For example, the following code finds the first value 7 in a file of integer values:

```
istream_iterator<int, char> intstream(cin), eof;
istream_iterator<int, char>::iterator where =
         find(intstream, eof, 7);
```

The element denoted by an iterator for an input stream is valid only until the next element in the stream is requested. Also, since an input stream iterator is an input iterator, elements can only be accessed, not modified by assignment. Finally, elements can be accessed only once, and only in a forward moving direction. If you want to read the contents of a stream more than one time, you must create a separate iterator for each pass.

Output Stream Iterators

The output stream iterator mechanism is analogous to the input stream iterator. Each time a value is assigned to the iterator, it is written on the associated output stream using the operator `>>`. To create an output stream iterator, you must specify the associated output stream as an argument with the constructor. Values written to the output stream must recognize the stream `>>` operation. An optional second argument to the constructor is a string used as a separator between each pair of values. For example, the following code copies all the values from a vector into the standard output, and separates each value by a space:

```
copy (newdata.begin(), newdata.end(),
      ostream_iterator<int,char> (cout, " "));
```

Simple file transformation algorithms can be created by combining input and output stream iterators and the various algorithms provided by the Standard C++ Library. The following short program reads a file of integers from the standard input, removes all occurrences of the value 7, and copies the remainder to the standard output, separating each value by a new line:

```
void main()
{
   istream_iterator<int, char> input (cin), eof;
   ostream_iterator<int, char> output (cout, "\n");

   remove_copy (input, eof, output, 7);
}
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Insert Iterators

Assignment to the dereferenced value of an output iterator is normally used to *overwrite* the contents of an existing location. For example, the following invocation of the function `copy()` transfers values from one vector to another, although the space for the second vector was already set aside and even initialized by the declaration statement:

```
vector<int> a(10);
vector<int> b(10);
   ...
copy (a.begin(), a.end(), b.begin());
```

Even structures such as lists can be overwritten in this fashion. In the following code, the list named `c` is assumed to have at least ten elements. The initial ten locations in the list will be replaced by the contents of the vector `a`.

```
list<int> c;
   ...
copy (a.begin(), a.end(), c.begin());
```

With structures such as lists and sets, which are dynamically enlarged as new elements are added, it is frequently more appropriate to *insert* new values into the structure, rather than to *overwrite* existing locations. A type of adaptor called an *insert iterator* allows us to use algorithms such as `copy()` to insert into the associated container, rather than overwrite elements in the container. The output operations of the iterator are changed into insertions into the associated container. For example, the following code inserts the values of the vector `a` into an initially empty list:

```
list<int> d;

copy (a.begin(), a.end(), front_inserter(d));
```

There are three forms of insert iterators, all of which can be used to change a *copy* operation into an *insert* operation. The iterator generated using `front_inserter`, shown above, inserts values into the front of the container. The iterator generated by `back_inserter` places elements into the back of the container. Both forms can be used with ***list***s and ***deque***s, but not with ***set***s or ***map***s. The iterator `back_inserter`, but not `front_inserter`, can be used with ***vector***.

The third and most general form of insert iterator is `inserter`, which takes two arguments: a container and an iterator within the container. This form copies elements into the specified location in the container. (For a ***list***, this means elements are copied immediately before the specified location). This form can be used with all the structures for which the previous two forms work, as well as with sets and maps.

The following simple program illustrates the use of all three forms of insert iterators. First, the values 3, 2, and 1 are inserted into the front of an initially empty list. Note that as each value is inserted, it becomes the new front, so that the resultant list is ordered 1, 2, 3. Next, the values 7, 8, and 9 are inserted into the end of the list. Finally, the `find()` operation is used to locate an iterator that denotes the 7 value, and the numbers 4, 5, and 6 are inserted immediately prior. The result is the list of numbers from 1 to 9 in order.

```
void main() {
   int threeToOne [ ] = {3, 2, 1};
   int fourToSix [ ] = {4, 5, 6};
   int sevenToNine [ ] = {7, 8, 9};

   list<int> aList;

                          // first insert into the front
                          // note that each value becomes new front
   copy (threeToOne, threeToOne+3, front_inserter(aList));

                          // then insert into the back
   copy (sevenToNine, sevenToNine+3, back_inserter(aList));


                          // find the seven, and insert into middle
   list<int>::iterator seven = find(aList.begin(), aList.end(), 7);
   copy (fourToSix, fourToSix+3, inserter(aList, seven));

                          // copy result to output
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   copy (aList.begin(), aList.end(),
         ostream_iterator<int,char>(cout, " "));
   cout << endl;
}
```

Note that there is an important and subtle difference between the iterators created by `inserter(aList, aList.begin())` and `front_inserter(aList)`. The call on `inserter(aList, aList.begin())` copies values in sequence, adding each one to the front of a list, whereas `front_inserter(aList)` copies values making each value the new front. The result is that `front_inserter(aList)` reverses the order of the original sequence, while `inserter(aList, aList.begin())` retains the original order.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Iterator Operations

The Standard C++ Library provides two functions that can be used to manipulate iterators. The function `advance()` takes an iterator and a numeric value as argument, and modifies the iterator by moving the given amount.

```
void advance (InputIterator & iter, Distance & n);
```

For random access iterators this is the same as `iter + n;` however, the function is useful because it is designed to operate with all forms of iterators. For forward iterators the numeric distance must be positive, whereas for bidirectional or random access iterators the value can be either positive or negative. The operation is efficient (constant time) only for random access iterators. In all other cases, it is implemented as a loop that invokes either the operators `++` or `--` on the iterator, and therefore takes time proportional to the distance traveled. The `advance()` function does not check to ensure the validity of the operations on the underlying iterator.

The second function, `distance()`, returns the number of iterator operations necessary to move from one element in a sequence to another. The description of this function is as follows:

```
Distance distance (InputIterator first, InputIterator last);
```

The result is returned in the third argument, which is passed by reference. Distance will *increment* this value by the number of times the operator `++` must be executed to move from `first` to `last`. Always be sure that the variable passed through this argument is properly initialized before invoking the function.

NOTE: The above definition of distance assumes that your compiler supports partial specialization. If it does not, then you must use the following alternate definition:
void distance (InputIterator first, InputIterator last, Distance &n);

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 3: Functions and Predicates

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Functions

A number of algorithms provided in the Standard C++ Library require functions as arguments. A simple example is the algorithm `for_each()`, which invokes a function, passed as an argument, on each value held in a container. For example, the following code applies the `printElement()` function to produce output describing each element in a list of integer values:

```
void printElement (int value)
{
   cout << "The list contains " << value << endl;
}

main ()
{
   list<int> aList;
      ...
   for_each (aList.begin(), aList.end(), printElement);
}
```

Binary functions take two arguments, and are often applied to values from two different sequences. For example, suppose we have a list of strings and a list of integers. For each element in the first list we wish to replicate the string the number of times given by the corresponding value in the second list. We can perform this easily using the function `transform()` from the Standard C++ Library. First, we define a binary function with the desired characteristics:

```
string stringRepeat (const string & base, int number)
                    // replicate base the given number of times
{
   string result;   // initially the result is empty
   while (number--)  result += base;
   return result;
}
```

The following call on `transform()` then produces the desired effect:

```
list<string> words;
list<int> counts;
   ...
transform (words.begin(), words.end(),
   counts.begin(), words.begin(), stringRepeat);
```

Transforming the words `one,two,three` with the values 3, 2, 3 would yield the result `oneoneone,twotwo,threethreethree`.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Predicates

A *predicate* is simply a function that returns either a boolean value or an integer value. Following the normal C convention, an integer value is assumed to be true if non-zero, and false otherwise. Here is an example of a predicate, which takes an integer as argument and returns `true` if the number represents a leap year, and `false` otherwise:

```
bool isLeapYear (unsigned int year)
                    // return true if year is leap year
{
                    // millennia are leap years
   if (0 == year % 1000) return true;
                   // every fourth century is
   if (0 == year % 400) return true;
                   // every fourth year is
   if (0 == year % 4) return true;
                   // otherwise not
   return false;
}
```

A predicate is used as an argument, for example, in the generic algorithm named `find_if()`. This algorithm returns the first value that satisfies the predicate, returning the end-of-range value if no such element is found. Using this algorithm, the following locates the first leap year in a list of years:

```
list<int>::iterator firstLeap =
   find_if(aList.begin(), aList.end(), isLeapYear);
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Function Objects

Definition

A *function object* is an instance of a class that defines the parenthesis operator as a member function. When a function object is used as a function, the parenthesis operator is invoked whenever the function is called. Consider the following class definition:

```
class biggerThanThree
 {
   public:
   bool operator () (int val)
   { return val > 3; }
};
```

If we create an instance of class ***biggerThanThree***, every time we reference this object using the function call syntax, the parenthesis operator member function is invoked. To generalize this class, we add a constructor and a constant data field, which is set by the constructor:

```
class biggerThan {
   public:
      const int testValue;
      biggerThan (int x) : testValue(x) { }

      bool operator () (int val)
      { return val > testValue; }
};
```

The result is a general ***biggerthanX*** function, where the value of X is determined when we create an instance of the class. We can do so, for example, as an argument to one of the generic functions that require a predicate. In this manner the following code finds the first value in a list that is larger than 12:

```
list<int>::iterator firstBig =
   find_if (aList.begin(), aList.end(), biggerThan(12));
```

Use

There are a number of situations where it is convenient to substitute function objects in place of functions: to use an existing function object provided by the Standard C++ Library instead of a new function; to improve execution by using inline function calls; and to allow a function object to access or set state information that is held by an object. Let's deal with each of these in the next three sections.

To Employ Existing Standard Library Function Objects

Table 5 illustrates the function objects provided by the Standard C++ Library.

Table 5 -- Function objects provided by the Standard C++ Library

Function object	Implemented operations
Arithmetic functions	
`plus`	addition `x + y`
`minus`	subtraction `x - y`
`multiplies`	multiplication `x * y`
`divides`	division `x / y`
`modulus`	remainder `x % y`
`negate`	negation `- x`
Comparison functions	

©Copyright 1999 Rogue Wave Software, Inc.

`equal_to`	equality test `x == y`
`not_equal_to`	inequality test `x != y`
`greater`	greater comparison `x > y`
`less`	less-than comparison `x < y`
`greater_equal`	greater than or equal comparison `x >= y`
`less_equal`	less than or equal comparison `x <= y`
Logical functions	
`logical_and`	logical conjunction `x && y`
`logical_or`	logical disjunction `x || y`
`logical_not`	logical negation `! x`

Let's look at a couple of examples that show how these might be used. The first example uses `plus()` to compute the by-element addition of two lists of integer values, placing the result back into the first list. This can be performed by the following code:

```
transform (listOne.begin(), listOne.end(), listTwo.begin(),
   listOne.begin(), plus<int>() );
```

The second example negates every element in a vector of boolean values:

```
transform (aVec.begin(), aVec.end(), aVec.begin(),
   logical_not<bool>() );
```

The base classes used by the Standard C++ Library to define the functions in Table 5 are also available for creating new unary and binary function objects. The class definitions for ***unary_function*** and ***binary_function*** can be incorporated by #including `functional`.

The base classes are defined as follows:

```
template <class Arg, class Result>
struct unary_function {
   typedef Arg argument_type;
   typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

An example of the use of these functions is found in Section 6.3. There we want to take a binary function of type `Widget` and an argument of type `integer`, and compare the widget identification number against the integer value. A function to do this is written in the following manner:

```
struct WidgetTester : binary_function<Widget, int, bool> {
public:
   bool operator () (const Widget & wid, int testid) const
      { return wid.id == testid; }
};
```

To Improve Execution

A second reason to consider using function objects instead of functions is faster code. In many cases an invocation of a function object, as in the examples on `transform()` in Section 3.3.2.1, can be expanded in-line, eliminating the overhead of a function call.

To Access or Set State Information

The third major reason to use a function object in place of a function is when each invocation of the function must remember some state set by earlier invocations. An example of this occurs in the creation of a generator, to be used with the generic algorithm `generate()`. A *generator* is simply a function that returns a different value each time it is invoked. The most commonly used form of generator is a *random number generator*, but there are other uses for the concept. A *sequence generator* simply returns the values of an increasing sequence of natural numbers (1, 2, 3, 4 and so on). We can call this object ***iotaGen*** after the similar operation in the programming language APL, and define it as follows:

©Copyright 1999 Rogue Wave Software, Inc.

```
class iotaGen {
public:
   iotaGen (int start = 0) : current(start) { }
   int operator () () { return current++; }
private:
   int current;
};
```

An iota object maintains a current value, which can be set by the constructor, or defaults to zero. Each time the function-call operator is invoked, the current value is returned, and also incremented. Using this object, the following call on the Standard C++ Library function `generate()` initializes a vector of 20 elements with the values 1 through 20:

```
vector<int> aVec(20);
generate (aVec.begin(), aVec.end(), iotaGen(1));
```

A more complex example of using a function object occurs in the radix sorting example program, which is given as an example of using the list datatype in Section 6.3. In this program references are initialized in the function object, so that during the sequence of invocations the function object can access and modify local values in the calling program.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Function Adaptors

Definition

A *function adaptor* is an instance of a class that adapts a global or member function so that the function can be used as a function object. A function adaptor may also be used to alter the behavior of a function or function object, as shown in Section 3.5. Each function adaptor provides a constructor that takes a global or member function. The adaptor also provides a parenthesis operator that forwards its call to that associated global or member function.

Adapting Global Functions

The ***pointer_to_unary_function*** and ***pointer_to_binary_function*** templates adapt global functions of one or two arguments. These adaptors can be applied directly, or the ***ptr_fun*** function template can be used to construct the appropriate adaptor automatically. For instance, a simple `times3` function can be adapted and applied to a vector of integers as follows:

```
int times3(int x) {
  return 3*x;
}

int a{} {1,2,3,4,5};
vector<int> v(a,a+5), v2;

transform(v.begin(),v.end(),v2.end(),ptr_fun(times3));
```

Alternatively, the adapter could have been applied, and the new, adapted function object passed to the vector:

```
pointer_to_unary_function<int,int> pf(times3);
transform(v.begin(),v.end(),v2.end(),pf);
```

This example points out the advantage of allowing the compiler to deduce the types needed by ***pointer_to_unary_function*** through the use of `ptr_fun`.

Adapting Member Functions

The ***mem_fun*** family of templates adapts member functions, rather than global functions. For instance, to sort each list in a given set of lists, ***mem_fun_t*** or simply ***mem_fun*** can be used to apply the list sort member function to each element in the set:

```
set<list<int>* > s;

// Initialize the set with lists
...
// Sort each list in the set.
for_each(s.begin(),s.end(),mem_fun(&list<int>::sort));

// Now each list in the set is sorted
```

Using ***mem_fun*** is necessary because the generic sort algorithm cannot be used on a list. It is also the simplest way to access any polymorphic characteristics of an object held in a standard container. For instance, a virtual draw function might be invoked on a collection of objects that are all part of the canonical `shape hierarchy` like this:

```
// shape hierarchy
class shape {
   virtual void draw();
};

class circle : public shape {
   void draw();
};
```

©Copyright 1999 Rogue Wave Software, Inc.

```
class square : public shape {
  void draw();
};

//  Assemble a vector of shapes
circle c;
square s;
vector<shape*> v;
v.push_back(&s);
v.push_back(&c);

// Call draw on each one
for_each(v.begin(),v.end(), mem_fun(&shape::draw));
```

Like the global function adaptors, each member function adaptor consists of a class template and an associated function template. The class is the actual adaptor, while the function simplifies the use of the class by constructing instances of that class on the fly. For instance, in the above example, a user could construct a ***mem_fun_t***, and pass that to the `for_each` algorithm:

```
mem_fun_t<shape> mf(&shape::draw);
for_each(v.begin(),v.end(),mf);
```

Here again the ***mem_fun*** function template simplifies the use of the ***mem_fun_t*** adaptor by allowing the compiler to deduce the type needed by ***mem_fun_t***.

The Standard C++ Library provides member function adaptors for functions with zero arguments as above and with one argument. They can be easily extended to member functions with more arguments.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Negators and Binders

Negators and *binders* are function adaptors that are used to build new function objects out of existing function objects. Almost always, negators and binders are applied to functions as part of the process of building an argument list prior to invoking yet another function or generic algorithm.

The negators ***not1()*** and ***not2()*** take a unary and a binary predicate function object, respectively, and create a new function object that yields the complement of the original. For example, using the widget tester function object defined in Section 3.3.2.1, the function object

```
not2(WidgetTester())
```

yields a binary predicate which takes exactly the same arguments as the widget tester. It is true when the corresponding widget tester is false, and false otherwise. Negators work only with function objects defined as subclasses of the classes ***unary_function*** and ***binary_function***, given earlier.

A binder takes a two-argument function, and binds either the first or second argument to a specific value, thereby yielding a one-argument function. The underlying function must be a subclass of class ***binary_function***. The binder ***bind1st()*** binds the first argument, while the binder ***bind2nd()*** binds the second.

For example, the binder ***bind2nd(greater<int>(), 5)*** creates a function object that tests for being larger than 5. This could be used in the following, which yields an iterator representing the first value in a list larger than 5:

```
list<int>::iterator where = find_if(aList.begin(), aList.end(),
          bind2nd(greater<int>(), 5));
```

Combining a binder and a negator, we can create a function that is true if the argument is divisible by 3, and false otherwise. This function can be used to remove all the multiples of 3 from a list.

```
list<int>::iterator where = remove_if (aList.begin(), aList.end(),
          not1(bind2nd(modulus<int>(), 3)));
```

A binder is used to tie the widget number of a call to the binary function `WidgetTester()`, yielding a one-argument function that takes only a widget as argument. This is used to find the first widget that matches the given widget type:

```
list<Widget>::iterator wehave =
   find_if(on_hand.begin(), on_hand.end(),
      bind2nd(WidgetTester(), wid));
```

NOTE: The idea described here by the term binder is in other contexts often described by the term *curry*. It is named after the computer scientist Haskell P. Curry, who used the concept extensively in an influential book on the theory of computation in the 1930s. Curry himself attributed the idea to Moses Schönfinkel, leaving one to wonder why we don't instead refer to binders as Schönfinkels.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part III: Containers

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 4: Container Classes

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Overview

The Standard C++ Library provides ten alternative forms of containers and three container adaptors. In this chapter we briefly identify these forms, consider their characteristics, and discuss how you might go about selecting which form to use in solving a particular problem. Subsequent chapters then go over each of the different forms in more detail.

Table 6 lists the container types provided by the Standard C++ Library, and describes the most significant characteristic of each. Table 7 does the same for the container adaptors.

Table 6 -- Container types provided by the Standard C++ Library

Name	Characteristic
`vector`	Random access to elements, efficient insertion at end
vector<bool>	Specialization of `vector` optimized for `bool`
`list`	Efficient insertion and removal throughout
`deque`	Random access, efficient insertion at front or back
`set`	Elements maintained in order, efficient test for inclusion, insertion, and removal
`multiset`	Set with repeated copies
bitset	Bit container templated on size rather than contained type
`map`	Access to values via keys, efficient insertion and removal
`multimap`	Map permitting duplicate keys
string	Character container enhanced for string operations

Table 7 -- Container adaptors of the Standard C++ Library

Name	Characteristic
`stack`	Insertions and removals only from top
`queue`	Insertions at back, removals from front
`priority_queue`	Efficient access and removal of largest values

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Selecting a Container

Given ten Standard C++ Library containers, which type of container is best suited for solving a particular problem? Sometimes the answer is obvious, but other times there can be several viable alternatives. For the difficult cases, you may want to compare the actual execution timings using different containers to determine the best alternative. For most other cases, these simple criteria can help you decide:

How are values going to be accessed?	If random access is important, then a ***vector*** or a ***deque*** should be used. If sequential access is sufficient, then one of the other structures may be suitable.
Is the order in which values are maintained in the collection important?	There are a number of different ways values can be sequenced. If a strict ordering is important throughout the life of the container, then the ***set*** data structure is an obvious choice, as insertions into a set are automatically placed in order. If this ordering is important only at one point--for example, at the end of a long series of insertions--then it might be easier to place the values into a ***list*** or ***vector***, and sort the resulting structure at the appropriate time. If the order that values are held in the structure is related to the order of insertion, then a ***stack***, ***queue***, or ***list*** may be the best choice.
Will the size of the structure vary widely over the course of execution?	If so, a ***list*** or ***set*** might be the best choice. A ***vector*** or ***deque*** will continue to maintain a large buffer even after elements have been removed from the collection. Conversely, if the size of the collection remains relatively fixed, than a vector or deque will use less memory than a list or set holding the same number of elements.
Is it possible to estimate the size of the collection?	The ***vector*** data structure provides a way to pre-allocate a block of memory of a given size, using the `reserve()` member function. This ability is not provided by the other containers.
Is testing to see whether a value is contained in the collection a frequent operation?	If so, then the ***set*** or ***map*** containers would be a good choice. Testing to see whether a value is contained in a set or map can be performed in a very small number of steps, logarithmic in the size of the container, whereas testing to see if a value is contained in one of the other types of collections might require comparing the value against every element being stored by the container.
Is the collection indexed? That is, can the collection be viewed as a series of key/value pairs?	If the keys are integers between 0 and some upper limit, a ***vector*** or ***deque*** should be used. On the other hand, if the key values are some other ordered datatype--like character, string, or user-defined type--the ***map*** container can be used.
Can values be related to each other?	All values stored in any container provided by the Standard C++ Library must be able to test for equality against another similar value, but not all need to recognize the relational less-than operator. However, if values cannot be ordered using the relational less-than operator, they cannot be stored in a ***set*** or a ***map***.
Is finding and removing the largest value from the collection a frequent operation?	If the answer is yes, the ***priority_queue*** is the best data structure to use.
At what positions are values inserted into or removed from the structure?	If values are inserted into or removed from the middle, then a ***list*** is the best choice. If values are inserted only at the beginning, a ***deque*** or a ***list*** is the preferred choice. If values are inserted or removed only at the end, a ***stack*** or ***queue*** may be a logical choice.
Is the merging of two or more sequences into one a frequent operation?	If so, a ***set*** or a ***list*** would seem to be the best choice, depending whether the collection is maintained in order. Merging two sets is a very efficient operation. If the collections are not ordered, but the efficient `splice()` member function from class list can be used, then the list datatype is to be preferred, since this operation is not provided in the other containers.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Memory Management Issues

Containers in the Standard C++ Library can maintain a variety of different types of elements. These include the fundamental datatypes (`integer`, `char`, and so on), pointers, or user-defined types. Containers cannot hold references. In general, memory management is handled automatically by the standard container classes through the allocator template parameter type.

An allocator type can be provided as a second template parameter when declaring any container. Of course, if the allocator template parameter is provided explicitly, then its type must match the contained type. By default, all containers use the standard default allocator. See Section 15.3 for a discussion of user-defined allocators.

Values are placed into a container using the copy constructor. Some operations on a container require a default constructor. Generic algorithms that copy into a container, like `copy()`, use the assignment operator.

When an entire container is duplicated by invoking a copy constructor or by assignment, for example, every value is copied into the new structure using either the copy constructor or the assignment operator. Whether a deep copy or a shallow copy results is controlled by the programmer, who can provide the assignment operator with whatever meaning is desired. Memory for structures used internally by the various container classes is allocated and released automatically and efficiently.

If a destructor is defined for the element type, this destructor is invoked when values are removed from a container. When an entire collection is destroyed, the destructor is invoked for each remaining value being held by the container.

A few words should be said about containers that hold pointer values. Such collections are not uncommon. For example, a collection of pointers is the only way to store values that can potentially represent either instances of a class or instances of a subclass. Such a collection is encountered in an example problem discussed in Section 11.3.

In these cases, the container is responsible only for maintaining the pointer values themselves. It is the responsibility of the programmer to manage the memory for the values being referenced by the pointers. This includes making certain that the memory values are properly allocated, usually by invoking operator `new`; that they are not released while the container holds references to them; and that they are properly released once they have been removed from the container.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Container Types Not Found in the Standard Library

The Standard C++ Library provides the containers used in the solution of most programming problems. However, there are a number of classic container types that are not included. In most cases, this is because the provided containers can be easily adapted to a wide variety of uses, including the uses traditionally provided by the omitted containers. Table 8 lists the container types that are not contained in the library, and the simple substitution.

Table 8 -- Container types not given in the Standard C++ Library

Container type NOT given	Standard C++ Library substitution
tree	The ***set*** datatype is internally implemented using a form of binary search tree. For most problems that would be solved using trees, the ***set*** datatype is an adequate substitute.
multidimensional array	Since ***vector***s can hold other ***vector***s as elements, such structures can be easily constructed.
graph	One representation for graphs can be easily constructed as a map that holds other maps. This type of structure is described in the sample problem discussed in Section 9.3.2.
sparse array	A novel substitution is the graph representation discussed in Section 9.3.2.
hash table	A hash table provides amortized constant time access, and insertion and removal of elements, by converting access and removal operations into indexing operations. In the Standard C++ Library, a hash table can be easily constructed as a ***vector*** (or ***deque***) that holds ***list***s (or even ***set***s) as elements. A similar structure is described in the radix sort sample problem discussed in Section 7.3, although this example does not include invoking the hash function to convert a value into an index.
some ***set*** functionality	In the Standard C++ Library, the ***set*** datatype is specifically ordered, and ***set*** operations (union, intersection, and so on) cannot be performed on a collections of unordered values (for example, a set of complex numbers). A ***list*** can be used as a substitute, although it is still necessary to write special set operation functions, as the generic algorithms cannot be used with ***list***s.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 5: vector and vector<bool>

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

The vector Data Abstraction

The ***vector*** container class generalizes the concept of an ordinary C array. Like an array, a ***vector*** is an indexed data structure, with index values that range from 0 to one less than the number of elements contained in the structure. Also like an array, values are most commonly assigned to and extracted from the ***vector*** using the `subscript` operator. However, the ***vector*** differs from an array in the following important respects:

- A ***vector*** has more self-knowledge than an ordinary array. In particular, a ***vector*** can be queried about its size, about the number of elements it can potentially hold, which can differ from its current size, and so on.
- The size of the ***vector*** can change dynamically. New elements can be inserted on to the end of a ***vector***, or into the middle. Storage management is handled efficiently and automatically. It is important to note, however, that while these abilities are provided, insertion into the middle of a ***vector*** is not as efficient as insertion into the middle of a ***list*** (Section 6). If many insertion operations are to be performed, the ***list*** container should be used instead of the ***vector*** datatype.

The ***vector*** container class in the Standard C++ Library should be compared and contrasted to the ***deque*** container class described in Chapter 7. Like a ***vector***, a ***deque*** (pronounced *deck*) is an indexed data structure. The major difference between the two is that a ***deque*** provides efficient insertion at either the beginning or the end of the container, while a ***vector*** provides efficient insertion only at the end. In many situations, either structure can be used. Use of a ***vector*** generally results in a smaller executable file, while use of a ***deque*** may result in a slightly faster program, depending upon the particular set of operations being performed.

Include Files

Whenever you use a ***vector***, you must include the ***vector*** header file:

```
# include <vector>
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

vector Operations

In this section, each of the member functions provided by the ***vector*** datatype are described in more detail. These member functions provide the basic operations for ***vector***s. They can be greatly extended through the generic algorithms described in Chapter 13.

Declaration and Initialization of vectors

Because it is a template class, the declaration of a ***vector*** must include a designation of the component type. This can be a primitive language type, like an `integer` or `double`, a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a copy constructor, as this constructor is used to initialize newly created elements.

REMEMBER: Elements that are held by a vector must define a copy constructor. Although not used by functions in the vector class, some of the generic algorithms also require vector elements to recognize either the equivalence operator == or the relational less-than operator <.

Like an ***array***, a ***vector*** is most commonly declared with an integer argument that describes the number of elements the ***vector*** will hold and an initial value for each element:

```
vector<int> vec_one(10,0);
```

For ***vector***s as well as other Standard Library containers, an allocator type can also be provided as an additional template parameter:

```
vector<int,allocator<int>>
```

These two declarations are synonymous. Note that if the allocator template parameter is provided explicitly, then its type must match the contained type.

If the type has a default constructor, as it does in this case, then the second argument can be omitted. The constructor used to create the ***vector*** here is declared as `explicit`, which prevents it being used as a conversion operator. This is generally a good idea, since otherwise an `integer` might unintentionally be converted into a ***vector*** in certain situations.

There are a variety of other forms of constructor that can also be used to create ***vector***s. If no size is provided, the ***vector*** initially contains no elements, and increases in size automatically as elements are added. The copy constructor creates a clone of a ***vector*** from another ***vector***.

```
vector<int> vec_two(5, 3);      // copy constructor
vector<int> vec_three;
vector<int> vec_four(vec_two);  //  initialization by assignment
```

A ***vector*** can also be initialized using elements from another collection, by means of a beginning and ending iterator pair. The arguments can be any form of iterator; thus collections can be initialized with values drawn from any of the container classes in the Standard C++ Library that support iterators.

```
vector <int> vec_five (aList.begin(), aList.end());
```

NOTE: Because it requires the ability to define a method with a template argument different from the class template, some compilers may not yet support the initialization of containers using iterators. While compiler technology is catching up with the Standard C++ Library definition, the Rogue Wave version of the Standard C++ Library will support conventional pointers and vector iterators in this manner.

A ***vector*** can be assigned the values of another ***vector***, in which case the target receives a copy of the argument ***vector***.

```
vec_three = vec_five;
```

The `assign()` member function is similar to an assignment, but is more versatile and, in some cases, requires more arguments. Like an assignment, the existing values in the container are deleted, and replaced with the values specified by the arguments. There are two forms of `assign()`. The first takes two iterator arguments that specify a sub-sequence of an existing container. The values from this sub-sequence then become the new elements in the receiver. The second version

©Copyright 1999 Rogue Wave Software, Inc.

of `assign()` takes a count and an optional value of the container element type. After the call the container holds only the number of elements specified by the count, which are equal to either the default value for the container type or the initial value specified.

```
vec_six.assign(list_ten.begin(), list_ten.end());
vec_four.assign(3, 7);      // three copies of the value 7
vec_five.assign(12);        // twelve copies of value zero
```

If a destructor is defined for the container element type, the destructor is called for each value removed from the collection.

Finally, two ***vector***s can exchange their entire contents by means of the `swap()` operation. The argument container takes on the values of the receiver, while the receiver assumes those of the argument. A swap is very efficient, and should be used in preference to an explicit element-by-element transfer where appropriate.

```
vec_three.swap(vec_four);
```

Type Definitions

The class ***vector*** includes a number of type definitions, most commonly used in declaration statements. For example, an iterator for a ***vector*** of `integers` can be declared in the following fashion:

```
vector<int>::iterator location;
```

In addition to `iterator`, Table 9 defines the following types:

Table 9 -- Type definitions for class vector

Type	Definition
`value_type`	The type associated with the elements the ***vector*** maintains.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying element.
`const_reference`	A reference to an underlying element that will not permit the element to be modified.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`difference_type`	A signed integer type, used to describe distances between iterators.
`allocator_type`	The type of allocator used to manage memory for the ***vector***.

Subscripting a vector

The value being maintained by a ***vector*** at a specific index can be accessed or modified using the subscript operator, just like an ordinary array. Also like arrays, there are no attempts to verify the validity of the index values. Indexing a constant ***vector*** yields a constant reference. Attempts to index a ***vector*** outside the range of legal values generates unpredictable and spurious results:

```
cout << vec_five[1] << endl;
vec_five[1] = 17;
```

The member function `at()` can be used in place of the subscript operator. It takes exactly the same arguments as the subscript operator, and returns exactly the same values, but it will throw an out-of-range exception if the argument is invalid.

The member function `front()` returns the first element in the ***vector***, while the member function `back()` yields the last. Both also return constant references when applied to constant ***vector***s.

```
cout << vec_five.front() << " ... " << vec_five.back() << endl;
```

Extent and Size-Changing Operations

In general, there are three different *sizes* associated with any ***vector***:

- the number of elements currently being held by the ***vector***

©Copyright 1999 Rogue Wave Software, Inc.

- the maximum size to which the ***vector*** can be expanded without requiring that new storage be allocated
- the upper limit on the size of any ***vector***.

These three values are yielded by the member functions `size()`, `capacity()`, and `max_size()`, respectively:

```
cout << "size: " << vec_five.size() << endl;
cout << "capacity: " << vec_five.capacity() << endl;
cout << "max_size: " << vec_five.max_size() << endl;
```

The maximum size is usually limited only by the amount of available memory, or the largest value that can be described by the datatype `size_type`. The current size and capacity are more difficult to characterize. As noted in Section 5.2.5, elements can be added to or removed from a ***vector*** in a variety of ways. When elements are removed from a ***vector***, the memory for the ***vector*** is generally not reallocated, and thus the size is decreased but the capacity remains the same. A subsequent insertion does not force a reallocation of new memory if the original capacity is not exceeded.

An insertion that causes the size to exceed the capacity generally results in a new block of memory being allocated to hold the ***vector*** elements. Values are then copied into this new memory using the assignment operator appropriate to the element type, and the old memory is deleted. Because this can be a potentially costly operation, the ***vector*** datatype provides a means for the programmer to specify a value for the capacity of a ***vector***. The member function `reserve()` is a directive to the ***vector***, indicating that the ***vector*** is expected to grow to at least the given size. If the argument used with `reserve()` is larger than the current capacity, a reallocation occurs and the argument value becomes the new capacity. (It may subsequently grow even larger; the value given as the argument need not be a bound, just a guess.) If the capacity is already in excess of the argument, no reallocation takes place. Invoking `reserve()` does not change the size of the ***vector***, nor the element values themselves, with the exception that they may potentially be moved should reallocation take place.

```
vec_five.reserve(20);
```

A reallocation invalidates all references, pointers, and iterators referring to elements being held by a ***vector***.

The member function `empty()` returns true if the ***vector*** currently has a size of zero, regardless of the capacity of the ***vector***. Using this function is generally more efficient than comparing the result returned by `size()` to zero.

```
cout << "empty is " << vec_five.empty() << endl;
```

The member function `resize()` changes the size of the ***vector*** to the value specified by the argument. Values are either added to or erased from the end of the collection as necessary. An optional second argument can be used to provide the initial value for any new elements added to the collection, although this argument is not optional if the contained type does not have a default constructor. If a destructor is defined for the element type, the destructor is called for any values that are removed from the collection:

```
// become size 12, adding values of 17 if necessary
vec_five.resize (12, 17);
```

NOTE: A vector stores values in a single large block of memory. A deque, on the other hand, employs a number of smaller blocks. This difference may be important on machines that limit the size of any single block of memory, because in such cases a deque will be able to hold much larger collections than a vector.

Inserting and Removing Elements

As noted earlier, the class ***vector*** differs from an ordinary array in that a ***vector*** can increase or decrease in size in certain circumstances. When an insertion causes the number of elements being held in a ***vector*** to exceed the capacity of the current block of memory being used to hold the values, a new block is allocated and the elements are copied to the new storage.

NOTE: Even adding a single element to a vector can, in the worst case, require time proportional to the number of elements in the vector, as each element is moved to a new location. If insertions are a prominent feature of your current problem, you should explore the possibility of using containers, such as lists or sets, which are optimized for insert operations.

A new element can be added to the back of a ***vector*** using the function `push_back()`. If there is space in the current allocation, this operation is very efficient (constant time).

```
vec_five.push_back(21);   // add element 21 to end of collection
```

©Copyright 1999 Rogue Wave Software, Inc.

The corresponding removal operation is `pop_back()`, which decreases the size of the ***vector***, but does not change its capacity. If the container type defines a destructor, the destructor is called on the value being eliminated. Again, this operation is very efficient. The class ***deque*** permits values to be added and removed from both the back and the front of the collection, as described in Chapter 7.

More general insertion operations can be performed using the `insert()` member function. The location of the insertion is described by an iterator; insertion takes place immediately preceding the location denoted. A fixed number of constant elements can be inserted by a single function call. It is much more efficient to insert a block of elements in a single call than to perform a sequence of individual insertions, because with a single call at most one allocation will be performed.

```
                                // find the location of the 7
vector<int>::iterator where =
      find(vec_five.begin(), vec_five.end(), 7);
                               // then insert the 12 before the 7
vec_five.insert(where, 12);
vec_five.insert(where, 6, 14);    // insert six copies of 14
```

The most general form of the `insert()` member function takes a position and a pair of iterators that denote a sub-sequence from another container. The range of values described by the sequence is inserted into the ***vector***. Again, using this function is preferable to using a sequence of individual insertions because at most a single allocation is performed.

```
vec_five.insert (where, vec_three.begin(), vec_three.end());
```

NOTE: Once more, it is important to remember that should an insertion cause reallocation, all references, pointers, and iterators that denoted a location in the now-deleted memory block become invalid.

In addition to the `pop_back()` member function, which removes elements from the end of a ***vector***, a function exists that removes elements from the middle of a ***vector***, using an iterator to denote the location. The member function that performs this task is `erase()`, which has two forms:

- the first takes a single iterator and removes an individual value
- the second takes a pair of iterators and removes all values in the given range. The size of the ***vector*** is reduced, but the capacity is unchanged. If the container type defines a destructor, the destructor is invoked on the eliminated values:

```
vec_five.erase(where);
                               // erase from the 12 to the end
where = find(vec_five.begin(), vec_five.end(), 12);
vec_five.erase(where, vec_five.end());
```

Iteration

The member functions `begin()` and `end()` yield random access iterators for the container. Again, we note that the iterators yielded by these operations can become invalidated after insertions or removals of elements. The member functions `rbegin()` and `rend()` return similar iterators, but these access the underlying elements in reverse order. Constant iterators are returned if the original container is declared as constant, or if the target of the assignment or parameter is constant.

Test for Inclusion

A ***vector*** does not directly provide any method that can be used to determine if a specific value is contained in the collection. However, the generic algorithms `find()` or `count()` can be used for this purpose (see Section 13.3.1 and Section 13.6.1). For example, the following statement tests to see whether an integer ***vector*** contains the element 17:

```
count (vec_five.begin(), vec_five.end(), 17);
```

If your compiler does not support partial specialization, then you must use the following interface instead:

```
int num = 0;
count (vec_five.begin(), vec_five.end(), 17, num);

if (num)
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;
```

©Copyright 1999 Rogue Wave Software, Inc.

NOTE: count() returns its result through an argument that is passed by reference. It is important that this value be properly initialized before invoking this function.

Sorting and Sorted vector Operations

A ***vector*** does not automatically maintain its values in sequence. However, a ***vector*** can be placed in order using the generic algorithm `sort()` (see Section 14.2). The simplest form of sort uses for its comparisons the less-than operator for the element type. An alternative version of the generic algorithm permits the programmer to specify the comparison operator explicitly. This can be used, for example, to place the elements in descending rather than ascending order:

```
   // sort ascending
sort (aVec.begin(), aVec.end());

   // sort descending, specifying the ordering function explicitly
sort (aVec.begin(), aVec.end(), greater<int>() );

   // alternate way to sort descending
sort (aVec.rbegin(), aVec.rend());
```

A number of the operations described in Chapter 14 can be applied to a ***vector*** holding an ordered collection. For example, two ***vector***s can be merged using the generic algorithm `merge()` (see Section 14.5).

```
// merge two vectors, printing output
merge (vecOne.begin(), vecOne.end(), vecTwo.begin(), vecTwo.end(),
   ostream_iterator<int,char> (cout, " "));
```

Sorting a ***vector*** also permits the more efficient binary search algorithms (see Section 14.4), instead of a linear traversal algorithm such as `find()`.

Useful Generic Algorithms

Most of the algorithms described in Part IV can be used with ***vector***s, but some are more useful than others. For example, the maximum value in a ***vector*** can be determined as follows:

```
vector<int>::iterator where =
   max_element (vec_five.begin(), vec_five.end());
cout << "maximum is " << *where << endl;
```

Table 10 summarizes some of the algorithms that are especially useful with ***vector***s:

Table 10 -- Generic algorithms useful with vectors

Algorithm	Purpose
`fill`	Fill a ***vector*** with a given initial value
`copy`	Copy one sequence into another
`generate`	Copy values from a generator into a ***vector***
`find`	Find an element that matches a condition
`adjacent_find`	Find consecutive duplicate elements
`search`	Find a sub-sequence within a ***vector***
`max_element, min_element`	Locate maximum or minimum element
`reverse`	Reverse order of elements
`replace`	Replace elements with new values
`rotate`	Rotate elements around a midpoint
`partition`	Partition elements into two groups
`next_permutation`	Generate permutations
`inplace_merge`	Inplace merge within a ***vector***
`random_shuffle`	Randomly shuffle elements in ***vector***
`count`	Count number of elements that satisfy condition
`accumulate`	Reduce ***vector*** to a single value
`inner_product`	Inner product of two ***vector***s
`equal`	Test two ***vector***s for pair-wise equality
`lexicographical_compare`	Lexical comparison

©Copyright 1999 Rogue Wave Software, Inc.

`transform`	Apply transformation to a ***vector***
`partial_sum`	Partial sums of values
`adjacent_difference`	Adjacent differences of value
`for_each`	Execute function on each element

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Boolean Vectors

In the Standard C++ Library, ***vector***s of bit values (boolean 1/0 values) are handled as a special case, so that the values can be efficiently packed with several elements to a word. The operations for a boolean ***vector***, ***vector<bool>***, are a superset of those for an ordinary ***vector***, only the implementation is more efficient.

One new member function added to the boolean ***vector*** datatype is `flip()`. When invoked, this function inverts all the bits of the ***vector***. Boolean ***vector***s also return as reference an internal value that also supports the `flip()` member function:

```
vector<bool> bvec(27);
bvec.flip();               // flip all values
bvec[17].flip();           // flip bit 17
```

The ***vector<bool>*** also supports an additional `swap()` member function that allows swapping the values indicated by a pair of references:

```
bvec.swap(bvec [17], bvec [16]);
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Example Program: The Sieve of Eratosthenes

An example program that illustrates the use of vectors is the classic algorithm called the *sieve of Eratosthenes*, used to discover prime numbers.

NOTE: Source code for this program is in the file sieve.cpp.

In the sieve of Eratosthenes, a list of all the numbers up to some bound is represented by an integer vector. The basic idea is to strike out (set to zero) all values that cannot be primes; thus all the remaining values are prime numbers. To do this, a loop examines each value in turn; for those that are set to one and thus not yet excluded from the set of candidate primes, it strikes out all multiples of the number. When the outermost loop is finished, all remaining prime values have been discovered. Here is the program:

```
void main() {
      // create a sieve of integers, initially set
   const int sievesize = 100;
   std::vector<int> sieve(sievesize, 1);

      // now search for 1 bit positions
   for (int i = 2; i * i < sievesize; i++)
      if (sieve[i])
         for (int j = i + i; j < sievesize; j += i)
            sieve[j] = 0;

      // finally, output the values that are set
   for (int j = 2; j < sievesize; j++)
      if (sieve[j])
         std::cout << j << " ";
   std::cout << std::endl;
}
```

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 6: list

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

The list Data Abstraction

The ***vector*** data structure is a container of relatively fixed size. While the Standard C++ Library provides facilities for dynamically changing the size of a vector, such operations are costly and should be used only rarely. Yet in many problems, the size of a collection may be difficult to predict in advance, or may vary widely during the course of execution. For cases that suggest an alternative data structure, we examine the ***list*** datatype in this chapter.

A ***list*** corresponds to the intuitive idea of holding elements in a linear sequence that is not necessarily ordered. New values can be added to or removed from either the front or the back of the ***list***. By using an iterator to denote a position, elements can also be added to or removed from the middle of a ***list***. In all cases the insertion or removal operations are efficient; they are performed in a constant amount of time that is independent of the number of elements being maintained in the collection.

A ***list*** is a linear structure. In general, elements of a ***list*** can only be accessed by a linear traversal of all values, not by subscript.

Include files

Whenever you use a ***list***, you must include the `list` header file:

```
# include <list>
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

list Operations

In this section, each of the member functions provided by the ***list*** datatype are described in more detail. These member functions provide the basic operations for ***list***s. They can be greatly extended through the generic algorithms described in Chapter 13.

Declaration and Initialization of lists

There are a variety of ways to declare a ***list***. In the simplest form, a ***list*** is declared by simply stating the type of element the collection will maintain. This can be a primitive language type, such as an `integer` or `double`, a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a copy constructor, as this constructor is used to initialize newly created elements. A collection declared in this fashion initially contains no elements:

```
list <int> list_one;
list <Widget *> list_two;
list <Widget> list_three;
```

NOTE: If you declare a container as holding pointers, you are responsible for managing the memory for the objects pointed to. The container classes will not automatically free memory for these objects when an item is erased from the container.

An alternative form of declaration creates a collection that initially contains some number of equal elements. The constructor for this form is declared as `explicit`, meaning it cannot be used as a conversion operator. This prevents integers from inadvertently being converted into ***list***s. The constructor takes two arguments, a *size* and an *initial value*. The second argument is optional if the contained type has a default constructor. If only the number of initial elements to be created is given, these values are initialized with the default constructor; otherwise the elements are initialized with the value of the second argument:

```
list <int> list_four (5);  // five elements, initialized to zero
list <double> list_five (4, 3.14);   // 4 values, initially 3.14
list <Widget> wlist_six (4);  // default constructor, 4 elements
list <Widget> list_six (3, Widget(7));  // 3 copies of Widget(7)
```

These ***list***s can also be initialized using elements from another collection, with a beginning and ending iterator pair. The arguments can be any form of iterator; thus collections can be initialized with values drawn from any of the container classes in the Standard C++ Library that support iterators. Because this requires the ability to specialize a member function using a template, some compilers may not yet support this feature. In these cases an alternative technique using the `copy()` generic algorithm can be employed. When a ***list*** is initialized using `copy()`, an *insert iterator* must be constructed to convert the output operations performed by the copy operation into ***list*** insertions. (Section 2.4.) The inserter requires two arguments: the ***list*** into which the value is to be inserted, and an iterator indicating the location at which values will be placed. Insert iterators can also be used to copy elements into an arbitrary location in an existing ***list***.

```
list <double> list_seven (aVector.begin(), aVector.end());

                  // the following is equivalent to the above
list <double> list_eight;
copy (aVector.begin(), aVector.end(),
       inserter(list_eight, list_eight.begin()));
```

The `insert()` operation can also be used to place values denoted by an iterator into a ***list*** (Section 6.2.3). Insert iterators can be used to initialize a ***list*** with a sequence of values produced by a *generator* (Section 13.2.3). This is illustrated by the following:

```
list <int> list_nine;
                                  // initialize list 1 2 3 ... 7
generate_n (inserter(list_nine, list_nine.begin()),
    7, iotaGen(1));
```

A *copy constructor* can be used to initialize a ***list*** with values drawn from another ***list***. The assignment operator performs the same actions. In both cases the assignment operator for the element type is used to copy each new value.

©Copyright 1999 Rogue Wave Software, Inc.

```
list <int> list_ten (list_nine);          // copy constructor
list <Widget> list_eleven;
list_eleven = list_six;        // values copied by assignment
```

The `assign()` member function is similar to the assignment operator, but is more versatile and, in some cases, requires more arguments. Like an assignment, the existing values in the container are deleted, and replaced with the values specified by the arguments. If a destructor is provided for the container element type, it is invoked for the elements being removed. There are two forms of `assign()`:

- The first form takes two iterator arguments that specify a sub-sequence of an existing container. The values from this sub-sequence then become the new elements in the receiver.
- The second form takes a count and an optional value of the container element type. After the call the container holds the number of elements specified by the count, which is equal to either the default value for the container type or the initial value specified. The initial value must be specified if the contained type does not have a default constructor.

```
list_six.assign(list_ten.begin(), list_ten.end());
list_four.assign(3, 7);        // three copies of value seven
list_five.assign(12);            // twelve copies of value zero
```

Finally, two ***list***s can exchange their entire contents by means of the operation `swap()`. The argument container takes on the values of the receiver, while the receiver assumes those of the argument. A swap is very efficient, and should be used in preference to an explicit element-by-element transfer where appropriate.

```
list_ten.swap(list_nine);        // exchange lists nine and ten
```

Type Definitions

The class ***list*** includes a number of type definitions, most commonly used in declaration statements. For example, an iterator for a ***list*** of `integers` can be declared as follows:

```
list<int>::iterator location;
```

In addition to `iterator`, Table 11 defines the following types:

Table 11 -- Type definitions for class list

Type	Definition
`value_type`	The type associated with the elements the list maintains.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying element.
`const_reference`	A reference to an underlying element that will not permit the element to be modified.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`difference_type`	A signed integer type, used to describe distances between iterators.
`allocator_type`	The allocator type used for all storage management by the list.

Placing Elements into a list

Values can be inserted into a ***list*** in a variety of ways. Elements are most commonly added to the front or back of a ***list***. These tasks are provided by the `push_front()` and `push_back()` operations, respectively. These operations are efficient (constant time) for both types of containers:

```
list_seven.push_front(1.2);
list_eleven.push_back (Widget(6));
```

In Section 6.2.1 we noted how values can be placed into a ***list*** at a location denoted by an iterator with the aid of an insert iterator and the `copy()` or `generate()` generic algorithm. There is also a member function, named `insert()`, that avoids the need to construct the inserter. As we will describe shortly, the values returned by the iterator generating functions `begin()` and `end()` denote the beginning and end of a ***list***, respectively. An insert using one of these is equivalent to `push_front()` or `push_back()`, respectively. If we specify only one iterator, the default element value is inserted:

©Copyright 1999 Rogue Wave Software, Inc.

```
// insert default type at beginning of list
list_eleven.insert(list_eleven.begin());
// insert widget 8 at end of list
list_eleven.insert(list_eleven.end(), Widget(8));
```

An iterator can denote a location in the middle of a ***list***. There are several ways to produce this iterator. For example, we can use the result of any of the searching operations described in Section 13.3, such as an invocation of the `find()` generic algorithm:

```
//  find the location of the first occurrence of the
//  value 5 in list
list<int>::iterator location =
  find(list_nine.begin(), list_nine.end(), 5);
// and insert an 11 immediate before it
location = list_nine.insert(location, 11);
```

Here the new value is inserted immediately *prior* to the location denoted by the iterator. The `insert()` operation itself returns an iterator denoting the location of the inserted value. This result value was ignored in the invocations shown above.

NOTE: Unlike a vector or deque, insertions or removals from the middle of a list will not invalidate references or pointers to other elements in the container. This property can be important if two or more iterators are being used to refer to the same container.

It is also possible to insert a fixed number of copies of an argument value. This form of `insert()` does not yield the location of the values:

```
line_nine.insert (location, 5, 12);       // insert five twelves
```

Finally, an entire sequence denoted by an iterator pair can be inserted into a ***list***. Again, no useful value is returned as a result of the `insert()`.

```
// insert entire contents of list_ten into list_nine
list_nine.insert (location, list_ten.begin(), list_ten.end());
```

Splicing

There are a variety of ways to *splice* one list into another. A splice differs from an insertion in that the item is simultaneously added to the receiver list and removed from the argument list. For this reason, a splice can be performed very efficiently, and should be used whenever appropriate. As with an insertion, the member function `splice()` uses an iterator to indicate the location in the receiver list where the splice should be made. The argument is either an entire ***list***, a single element in a ***list*** (denoted by an iterator), or a sub-sequence of a ***list*** (denoted by a pair of iterators).

```
// splice the last element of list ten
list_nine.splice (location, list_ten, list_ten.end());
// splice all of list ten
list_nine.splice (location,  list_ten);
// splice list 9 back into list 10
list_ten.splice (list_ten.begin(), list_nine,
   list_nine.begin(), location);
```

Two ordered ***list***s can be combined into one using the `merge()` operation. Values from the argument ***list*** are merged into the ordered ***list***, leaving the argument ***list*** empty. The merge is stable; that is, elements retain their relative ordering from the original ***list***s. As with the generic algorithm of the same name (Section 14.5), two forms are supported. The first form uses the operator `<` defined for the contained type. The second form uses the binary function supplied as argument to order values, but not all compilers support the second form. If the second form is desired and not supported, the more general generic algorithm can be used, although this is slightly less efficient:

```
// merge with explicit compare function
list_eleven.merge(list_six, widgetCompare);

//the following is similar to the above
list<Widget> list_twelve;
merge (list_eleven.begin(), list_eleven.end(),
   list_six.begin(), list_six.end(),
   inserter(list_twelve, list_twelve.begin()), widgetCompare);
list_eleven.swap(list_twelve);
```

Removing Elements

©Copyright 1999 Rogue Wave Software, Inc.

Just as there are a number of different ways to insert an element into a ***list***, there are a variety of ways to remove values from a ***list***. The most common operations used to remove a value are `pop_front()` or `pop_back()`, which delete the single element from the front or the back of the list, respectively. These member functions simply remove the given element, and do not themselves yield any useful result. If a destructor is defined for the element type, it is invoked as the element is removed. To look at the values before deletion, use the member functions `front()` or `back()`.

The `erase()` operation can be used to remove a value denoted by an iterator. For a ***list***, the argument iterator and any other iterators that denote the same location become invalid after the removal, but iterators denoting other locations are unaffected. We can also use `erase()` to remove an entire sub-sequence denoted by a pair of iterators. The values beginning at the initial iterator and up to, but not including, the final iterator are removed from the ***list***. Erasing elements from the middle of a ***list*** is an efficient operation, unlike erasing elements from the middle of a ***vector*** or a ***deque***:

```
   list_nine.erase (location);

   // erase values between the first occurrence of 5
   // and the following occurrence of 7
list<int>::iterator
   location = find(list_nine.begin(), list_nine.end(), 5);
   list<int>::iterator location2 =
       find(location, list_nine.end(), 7);
   list_nine.erase (location, location2);
```

The `remove()` member function removes all occurrences of a given value from a ***list***. A variation, `remove_if()`, removes all values that satisfy a given predicate. An alternative to either of these are the `remove()` or `remove_if()` generic algorithms (Section 13.5.1). The generic algorithms do not reduce the size of the ***list***; instead they move the elements to be retained to the front of the ***list***, leave the remainder of the ***list*** unchanged, and return an iterator denoting the location of the first unmodified element. This value can be used in conjunction with the `erase()` member function to remove the remaining values.

```
list_nine.remove(4);                          // remove all fours
list_nine.remove_if(divisibleByThree);     //remove any div by 3

// the following is equivalent to the above
list<int>::iterator location3 =
remove_if(list_nine.begin(), list_nine.end(),
           divisibleByThree);
list_nine.erase(location3, list_nine.end());
```

The operation `unique()` erases all but the first element from every consecutive group of equal elements in a ***list***. The ***list*** need not be ordered. An alternative version takes a binary function and compares adjacent elements using the function, removing the second value in those situations where the function yields a true value. As with `remove_if()`, not all compilers support the second form of `unique()`. In this case the more general `unique()` generic algorithm can be used (Section 13.5.2). In the following example, the binary function is the greater-than operator, which removes all elements smaller than a preceding element:

```
// remove first from consecutive equal elements
list_nine.unique();

// explicitly give comparison function
list_nine.unique(greater<int>());

// the following is equivalent to the above
location3 =
     unique(list_nine.begin(), list_nine.end(), greater<int>());
list_nine.erase(location3, list_nine.end());
```

Extent and Size-Changing Operations

The member function `size()` returns the number of elements being held by a container. The function `empty()` returns `true` if the container is empty, and is more efficient than comparing the size against the value `zero`.

```
cout << "Number of elements: " << list_nine.size () << endl;
if ( list_nine.empty () )
   cout << "list is empty " << endl;
else
   cout << "list is not empty " << endl;
```

The member function `resize()` changes the size of the ***list*** to the value specified by the argument. Values are either added or erased from the end of the collection as necessary. An optional second argument can be used to provide the initial value for any new elements added to the collection:

©Copyright 1999 Rogue Wave Software, Inc.

```
// become size 12, adding values of 17 if necessary
list_nine.resize (12, 17);
```

Access and Iteration

The member functions `front()` and `back()` return, but do not remove, the first and last items in the container, respectively. For a ***list***, access to other elements is possible only by removing elements until the desired element becomes the front or back element, or by using iterators.

There are three types of iterators that can be constructed for lists. The functions `begin()` and `end()` construct iterators that traverse the list in forward order. For the ***list*** datatype, `begin()` and `end()` create bidirectional iterators. The alternative functions `rbegin()` and `rend()` construct iterators that traverse in reverse order, moving from the end of the ***list*** to the front.

Test for Inclusion

The ***list*** datatypes do not directly provide any method that can be used to determine if a specific value is contained in the collection. However, either of the generic algorithms `find()` or `count()` can be used for this purpose (Section 13.3.1 and Section 13.3.1). The following statement, for example, tests whether an integer list contains the element 17:

```
count (vec_five.begin(), vec_five.end(), 17);
```

If your compiler does not support partial specialization, then you must use the following interface instead:

```
count(vec_five.begin)
int num = 0;
count(list_five.begin(), list_five.end(), 17, num);
if (num > 0)
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;

if (find(list_five.begin(), list_five.end(), 17) != list_five.end())
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" << endl;
```

Sorting and Sorted list Operations

The member function `sort()` places elements into ascending order. If a comparison operator other than < is desired, it can be supplied as an argument.

```
list_ten.sort ( );                  // place elements into sequence
list_twelve.sort (widgetCompare);       // sort with widget compare
                                        // function
```

Once a ***list*** has been sorted, a number of the generic algorithms for ordered collections can be used with ***list***s (Chapter 14).

Searching Operations

The various forms of searching functions described in Chapter 13.3, namely `find()`, `find_if()`, `adjacent find()`, `mismatch()`, `max_element()`, `min_element()`, or `search()`, can be applied to ***list***. In all cases the result is an iterator, which can be dereferenced to discover the denoted element, or used as an argument in a subsequent operation.

NOTE: The searching algorithms in the Standard C++ Library always return the end of range iterator if no element matching the search condition is found. Unless the result is guaranteed to be valid, it is a good idea to check for the end of range condition.

In-Place Transformations

A number of operations can be applied to ***list***s in order to transform them in place. Some of these are provided as member functions. Others make use of some of the generic functions described inChapter 13.

For a list, the member function `reverse()` reverses the order of elements in the list:

```
list_ten.reverse();                 // elements are now reversed
```

©Copyright 1999 Rogue Wave Software, Inc.

The generic algorithm `transform()` can be used to modify every value in a container by simply using the same container as both input and result for the operation (Section 13.7.1). For example, the following code increments each element of a list by one:

```
transform(list_ten.begin(), list_ten.end(),
   list_ten.begin(), bind1st(plus<int>(), 1));
```

To construct the necessary unary function, the first argument of the binary integer addition function is bound to the value one. The version of `transform()` that manipulates two parallel sequences can be used like this.

Similarly, the functions `replace()` and `replace_if()` can be used to replace elements of a ***list*** with specific values (Section 13.4.2). Rotations and partitions can also be performed with ***list***s (Section 13.4.3 and Section 13.4.4):

```
// find the location of the value 5, and rotate around it
location = find(list_ten.begin(), list_ten.end(), 5);
rotate(list_ten.begin(), location, list_ten.end());
// now partition using values greater than 7 partition(list_ten.begin(), list_ten.end(),
      bind2nd(greater<int>(), 7));
```

The functions `next_permutation()` and `prev_permutation()` can be used to generate the next permutation (or previous permutation) of a collection of values (Section 13.4.5):

```
next_permutation (list_ten.begin(), list_ten.end());
```

Other Operations

The algorithm `for_each()` applies a function to every element of a collection (Section 13.8). An illustration of this use is given in the radix sort example program in the section on the ***deque*** data structure (Section 7.3).

The `accumulate()` generic algorithm reduces a collection to a scalar value (Chapter 13.6.2). This can be used, for example, to compute the sum of a list of numbers. A more unusual use of `accumulate()` is illustrated in the radix sort example from Section 7.3:

```
cout << "Sum of list is: " <<
      accumulate(list_ten.begin(), list_ten.end(), 0) << endl;
```

Two ***list***s can be compared against each other. They are equal if they are the same size and all corresponding elements are equal. A ***list*** is less than another list if it is lexicographically smaller (Section 13.6.5).

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example Program: An Inventory System

Let's use a simple inventory management system to illustrate the use of several *list* operations. Assume a business, named *WorldWideWidgetWorks*, requires a software system to manage their supply of widgets.

NOTE: The executable version of the widget works program is in file widwork.cpp.

Widgets are simple devices, distinguished by different identification numbers:

```
class  Widget {
public:
   Widget(int a = 0) : id(a) { }
   void operator = (const Widget& rhs) { id = rhs.id; }
   int id;
   friend ostream & operator << (ostream & out,const Widget & w)
      { return out << "Widget " << w.id; }
   friend bool operator == (const Widget& lhs, const Widget& rhs)
      { return lhs.id == rhs.id; }
   friend bool operator< (const Widget& lhs, const Widget& rhs)
      { return lhs.id < rhs.id; }
};
```

The state of the inventory is represented by two *list*s: a ***list*** of widgets represents the stock of widgets on hand, and a ***list*** of widget identification types represents the type of widgets that customers have backordered. To handle our inventory we have two commands: `order()` processes orders, and `receive()` processes the shipment of a new widget.

```
class inventory {
public:
   void order (int wid);     // process order for widget type wid
   void receive (int wid);   // receive widget of type wid in shipment
private:
   list<Widget> on_hand;
   list<int> on_order;
};
```

When a new widget arrives in shipment, we compare the widget identification number with the list of widget types on backorder. We use `find()` to search the backorder list, immediately shipping the widget if necessary. Otherwise it is added to the stock on hand:

```
void inventory::receive (int wid)
{
   cout << "Received shipment of widget type " << wid << endl;
   list<int>::iterator weneed =
      find (on_order.begin(), on_order.end(), wid);
   if (weneed != on_order.end())
   {
      cout << "Ship " << Widget(wid)
           << " to fill back order" << endl;
      on_order.erase(weneed);
   }
   else
      on_hand.push_front(Widget(wid));
}
```

When a customer orders a new widget, we scan the list of widgets in stock to determine if the order can be processed immediately. We can use the function `find_if()` to search the list. To do so, we need a binary function that takes as its argument a widget and determines whether the widget matches the type requested. We create this function by taking a general binary widget-testing function, and binding the second argument to the specific widget type. To use the function `bind2nd()`, however, requires that the binary function be an instance of the class *binary_function*. The general widget-testing function is written as follows:

```
class WidgetTester : public binary_function<Widget, int, bool> {
public:
   bool operator () (const Widget & wid, int testid) const
```

©Copyright 1999 Rogue Wave Software, Inc.

```
      { return wid.id == testid; }
};
```

The widget order function is then written as follows:

```
void inventory::order (int wid)
{
   cout << "Received order for widget type " << wid << endl;
   list<Widget>::iterator wehave =
         find_if (on_hand.begin(), on_hand.end(),
            bind2nd(WidgetTester(), wid));
   if (wehave != on_hand.end())
   {
      cout << "Ship " << *wehave << endl;
      on_hand.erase(wehave);
   }
   else
   {
      cout << "Back order widget of type "  << wid  << endl;
      on_order.push_front(wid);
   }
}
```

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 7: deque

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The deque Data Abstraction

The name *deque* is short for *double-ended queue*, and is pronounced like *deck*. Traditionally, the term is used to describe any data structure that permits both insertions and removals from either the front or the back of a collection. The ***deque*** container class permits this and much more. In fact, the capabilities of the ***deque*** data structure are almost a union of those provided by the ***vector*** and ***list*** classes.

Like a ***vector***, the ***deque*** is an indexed collection. Values can be accessed by subscript, using the position within the collection as a key. This capability is not provided by the ***list*** class.

Like a ***list***, however, values can be efficiently added either to the front or to the back of a ***deque***. This capability is provided only in part by the vector class.

As with both the ***list*** and ***vector*** classes, insertions can be made into the middle of the sequence held by a ***deque***. Such insertion operations are not as efficient as with a ***list***, but slightly more efficient that they are in a ***vector***.

In short, a ***deque*** can often be used in situations that require a ***vector*** and in situations that require a ***list***. Often, using a ***deque*** in place of either a ***vector*** or a ***list*** results in faster programs. To determine which data structure should be used, you can refer to the set of questions described in Section 4.2

Include Files

The `deque` header file must appear in all programs that use the ***deque*** datatype:

```
# include <deque>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

deque Operations

A ***deque*** is declared the same way as a ***vector***, and includes within the class the same type definitions as ***vector***. The `begin()` and `end()` member functions return random access iterators, rather than the bidirectional iterators returned for ***list***s.

An insertion--either `insert()`, `push_front()`, or `push_back()`--can potentially invalidate all outstanding iterators and references to elements in the ***deque***. As with the ***vector*** datatype, this is a much more restrictive condition than insertions into a ***list***.

If the underlying element type provides a destructor, the destructor is invoked when a value is erased from a ***deque***.

Since the ***deque*** datatype provides random access iterators, all the generic algorithms that operate with ***vector***s can also be used with ***deque***s.

A ***vector*** holds elements in a single large block of memory. A ***deque***, on the other hand, uses a number of smaller blocks. This may be important on systems that restrict the size of memory blocks, as it permits a ***deque*** to hold many more elements than a ***vector***.

As values are inserted, the index associated with any particular element in the collection changes. For example, if a value is inserted into position 3, the value formerly indexed by 3 is now found at index location 4; the value formerly at 4 is now found at index location 5, and so on.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example Program: Radix Sort

The radix sort algorithm is a good illustration of how ***list***s and ***deque***s can be combined with other containers. In the radix sort, a ***vector*** of ***deque***s is manipulated much like a hash table.

NOTE: The complete radix sort program is in the file radix.cpp.

Radix sorting is a technique for ordering a list of positive integer values. The values are successively ordered on digit positions, from right to left. This is accomplished by copying the values into *buckets*, where the index for the bucket is given by the position of the digit being sorted. Once all digit positions are examined, the list must be sorted.

Table 12 shows the sequences of values found in each bucket during the four steps involved in sorting the list `624 852 426 987 269 146 415 301 730 78 593`. During pass 1, the ones place digits are ordered. During pass 2, the tens place digits are ordered, retaining the relative positions of values set by the earlier pass. On pass 3 the hundreds place digits are ordered, again retaining the previous relative ordering. After three passes the result is an ordered list.

Table 12 -- Sequence of values in each bucket during radix sort

Bucket	Pass 1	Pass 2	Pass 3
0	730	301	78
1	301	415	146
2	852	624, 426	269
3	593	730	301
4	624	146	415, 426
5	415	852	593
6	426, 146	269	624
7	987	78	730
8	78	987	852
9	269	593	987

The radix sorting algorithm is simple. A `while` loop is used to cycle through the various passes. The value of the variable `divisor` indicates which digit is currently being examined. A boolean flag is used to determine when execution should halt. Each time the `while` loop is executed, a ***vector*** of ***deque***s is declared. By placing the declaration of this structure inside the `while` loop, it is reinitialized to empty each step. Each time the loop is executed, the values in the ***list*** are copied into the appropriate bucket by executing the function `copyIntoBuckets()` on each value. Once distributed into the buckets, the values are gathered back into the ***list*** by means of an accumulation.

```
void radixSort(list<unsigned int> & values)
{
   bool flag = true;
   int divisor = 1;

   while (flag) {
      vector< deque<unsigned int> > buckets(10);
      flag = false;
      for_each(values.begin(), values.end(),
            copyIntoBuckets(...));
      accumulate(buckets.begin(), buckets.end(),
            values.begin(), listCopy);
      divisor *= 10;
      }
}
```

The use of the function `accumulate()` here is slightly unusual. The *scalar* value being constructed is the ***list*** itself. The initial value for the accumulation is the iterator denoting the beginning of the ***list***. Each bucket is processed by the following binary function:

©Copyright 1999 Rogue Wave Software, Inc.

```
list<unsigned int>::iterator
      listCopy(list<unsigned int>::iterator c,
         deque<unsigned int> & lst)
{
   // copy list back into original list, returning end
   return copy(lst.begin(), lst.end(), c);
}
```

The only difficulty remaining is defining the function `copyIntoBuckets()`. The problem here is that the function must take as its argument only the element being inserted, but it must also have access to the three values `buckets`, `divisor`, and `flag`. In languages that permit functions to be defined within other functions, the solution is to define `copyIntoBuckets()` as a local function within the `while` loop. But C++ has no such facilities. Instead, we must create a class definition, which can be initialized with references to the appropriate values. The parenthesis operator for this class is then used as the function for the `for_each()` invocation in the radix sort program.

```
class copyIntoBuckets {
public:
   copyIntoBuckets
      (int d, vector< deque<unsigned int> > & b, bool & f)
         : divisor(d), buckets(b), flag(f) {}

   int divisor;
   vector<deque<unsigned int> > & buckets;
   bool & flag;

   void operator () (unsigned int v)
   {   int index = (v / divisor) % 10;
       // flag is set to true if any bucket
       // other than zeroth is used
       if (index) flag = true;
       buckets[index].push_back(v);
   }
};
```

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 8: set, multiset, and bit set

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

The set Data Abstraction

A ***set*** is a collection of values. Although the abstract concept of a ***set*** does not necessarily imply an ordered collection, the ***set*** datatype is always ordered.

NOTE: If necessary, a collection of values that cannot be ordered can be maintained in an alternative data structure, such as a list.

Because the container used to implement the ***set*** data structure maintains values in an ordered representation, ***set***s are optimized for insertion and removal of elements, and for testing to see whether a particular value is contained in the collection. Each of these operations can be performed in a logarithmic number of steps, whereas for a ***list***, ***vector***, or ***deque***, each operation requires in the worst case an examination of every element held by the container. For this reason, ***set***s should be the data structure of choice in any problem that emphasizes insertion, removal, and test for inclusion of values. Like a ***list***, a ***set*** is not limited in size, but rather expands and contracts as elements are added to or removed from the collection.

There are two varieties of ***set***s provided by the Standard C++ Library. In the ***set*** container, every element is unique, and insertions of values that are already contained in the ***set*** are ignored. In the ***multiset*** container, on the other hand, multiple occurrences of the same value are permitted.

NOTE: In other programming languages, a multiset is sometimes referred to as a bag.

Include Files

Whenever you use a ***set*** or a ***multiset***, you must include the set header file:

```
# include <set>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

set and multiset Operations

The member functions provided by the *set* and *multiset* datatypes are described in the following sections. Note that while member functions provide basic operations, the utility of these data structures is greatly extended through the use of the generic algorithms described in Part IV.

Declaration and Initialization of set

A *set* is a template data structure, specialized by the type of the elements it contains and the operator used to compare keys. The latter argument is optional and, if not provided, the less-than operator for the key type is assumed. The element type can be a primitive language type, such as `integer` or `double`; a pointer type; or a user-defined type. The element type must recognize both the equality testing operator `==` and the less-than comparison operator `<`.

A *set* can be declared with no initial elements, or initialized from another container by providing a pair of iterators. The initialization of containers using a pair of iterators requires a mechanism that is still not widely supported by compilers. If not provided, the equivalent effect can be produced by declaring an empty set and using the `copy()` generic algorithm to copy values into the set.

Whether a set is declared with no initial elements or initialized from another container, an optional argument is an alternative comparison function; this value overrides the value provided by the template parameter. This mechanism is useful if a program contains two or more sets with the same values but different orderings, as it prevents more than one copy of the set member function from being instantiated. However, the contained template parameter type and the type passed to the container constructor must be the same. The copy constructor can be used to form a new set that is a clone, or copy, of an existing set.

```
set <int> set_one;

set <int, greater<int> > set_two;
set <int, greater<int> > set_three(greater<int>());

set <gadget, less<gadget> > gset;
set <gadget> gset(less<gadget>());

set <int> set_four (aList.begin(), aList.end());
set <int, greater<int>> set_five
   (aList.begin(), aList.end(), greater<int>());

   set <int> set_six (set_four);                // copy constructor
```

A set can be assigned to another set, and two sets can exchange their values using the `swap()` operation in a manner analogous to other Standard C++ Library containers.

```
set_one = set_five;
set_six.swap(set_two);
```

Type Definitions

The classes *set* and *multiset* include a number of type definitions, most commonly used in declaration statements. For example, an iterator for a set of `integers` can be declared in the following fashion:

```
set<int>::iterator location;
```

In addition to `iterator`, Table 13 defines the following types:

Table 13 -- Type definitions for class set and class multiset

Type	Definition
`value_type`	The type associated with the elements the set maintains.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.

©Copyright 1999 Rogue Wave Software, Inc.

`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying element.
`const_reference`	A reference to an underlying element that will not permit modification.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`value_compare`	A function that can be used to compare two elements.
`difference_type`	A signed integer type, used to describe the distance between iterators.
`allocator_type`	An allocator used by the container or all storage management.

Insertion

Unlike a ***list*** or ***vector***, there is only one way to add a new element to a ***set***. A value can be inserted into a ***set*** or a ***multiset*** using the `insert()` member function. With a ***multiset***, the function returns an iterator that denotes the value just inserted. Insert operations into a ***set*** return a ***pair*** of values, in which the first field contains an iterator, and the second field contains a boolean value that is true if the element was inserted, and false otherwise.

NOTE: If you want to use the pair datatype without using maps, you should include the header file named utility.

Recall that in a ***set***, an element will not be inserted if it matches an element already contained in the collection.

```
set_one.insert (18);

if (set_one.insert(18).second)
   cout << "element was inserted" << endl;
else
   cout << "element was not inserted " << endl;
```

Insertions of several elements from another container can also be performed using an iterator pair:

```
set_one.insert (set_three.begin(), set_three.end());
```

The ***pair*** data structure is a tuple of values. The first value is accessed through the field name `first`, while the second is, naturally, named `second`. A function named `make_pair()` simplifies the task of producing an instance of class ***pair***.

```
template <class T1, class T2>
struct pair {
   T1 first;
   T2 second;
   pair (const T1 & x, const T2 & y) : first(x), second(y) { }
};

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y)
   { return pair<T1, T2>(x, y); }
```

In determining the equivalence of keys, for example, to determine if the key portion of a new element matches any existing key, the comparison function for keys is used, and not the equivalence operator `==`. Two keys are deemed equivalent if the comparison function used to order key values yields false in both directions; that is, if `Compare(key1, key2)` is false, and if `Compare(key2, key1)` is false, then `key1` and `key2` are considered equivalent.

Removal of Elements from a set

Values are removed from a ***set*** using the member function `erase()`. The argument can be either a specific value, an iterator that denotes a single value, or a pair of iterators that denote a range of values. When the first form is used on a ***multiset***, all arguments matching the argument value are removed, and the return value indicates the number of elements that have been erased:

```
// erase element equal to 4
set_three.erase(4);

// erase element five    set<int>::iterator five = set_three.find(5);
set_three.erase(five);

// erase all values between seven and eleven
set<int>::iterator seven = set_three.find(7);
set<int>::iterator eleven = set_three.find(11);
set_three.erase (seven, eleven);
```

©Copyright 1999 Rogue Wave Software, Inc.

If the underlying element type provides a destructor, then the destructor is invoked prior to removing the element from the collection.

Searching and Counting

The member function `size()` yields the number of elements held by a container. The member function `empty()` returns a boolean true value if the container is empty, and is generally faster than testing the size against zero.

The member function `find()` takes an element value, and returns an iterator denoting the location of the value in the ***set*** if it is present, or a value matching the end-of-set value yielded by the function `end()` if it is not. If a ***multiset*** contains more than one matching element, the value returned can be any appropriate value.

```
set<int>::iterator five = set_three.find(5);
if (five != set_three.end())
    cout << "set contains a five" << endl;
```

The member functions `lower_bound()` and `upper_bound()` are most useful with ***multiset***s, since with ***set***s they simply mimic the function `find()`. The member function `lower_bound()` yields the first entry that matches the argument key, while the member function `upper_bound()` returns the first value past the last entry matching the argument. Finally, the member function `equal_range()` returns a ***pair*** of iterators holding the lower and upper bounds.

The member function `count()` returns the number of elements that match the argument. For a ***set***, this value is either zero or one, whereas for a ***multiset*** it can be any nonnegative value. Since a non-zero integer value is treated as true, the `count()` function can be used to test for inclusion of an element, if all that is desired is to determine whether or not the element is present in the set. The alternative, using `find()`, requires testing the result returned by `find()` against the end-of-collection iterator.

```
if (set_three.count(5))
   cout << "set contains a five" << endl;
```

Iterators

The member functions `begin()` and `end()` produce iterators for both ***set***s and ***multiset***s. The iterators produced by these functions are constant to ensure that the ordering relation for the ***set*** is not inadvertently or intentionally destroyed by assigning a new value to a ***set*** element. The iterators generate elements in sequence, ordered by the comparison operator provided when the ***set*** was declared. The member functions `rbegin()` and `rend()` produce iterators that yield the elements in reverse order.

NOTE: Unlike a vector or deque, the insertion or removal of values from a set does not invalidate iterators or references to other elements in the collection.

set Operations

The traditional set operations of `subset test`, `set union`, `set intersection`, and `set difference` are not provided as member functions, but are instead implemented as generic algorithms that work with any ordered structure. These functions are described in more detail in Section 14.6. The following sections describe how these functions can be used with the ***set*** and ***multiset*** container classes.

Subset test

The function `includes()` can be used to determine if one ***set*** is a subset of another; that is, if all elements from the first are contained in the second. For ***multiset***s the number of matching elements in the second ***set*** must exceed the number of elements in the first. The four arguments are a pair of iterators representing the (presumably) smaller ***set***, and a pair of iterators representing the (potentially) larger ***set***.:

```
if (includes(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end()))
      cout << "set_one is a subset of set_two" << endl;
```

The less-than operator `<` is used for the comparison of elements, regardless of the operator used in the declaration of the ***set***. Where this is inappropriate, an alternative version of the `includes()` function is provided. This form takes a fifth argument, which is the comparison function used to order the elements in the two ***set***s.

Set Union or Intersection

©Copyright 1999 Rogue Wave Software, Inc.

The function `set_union()` can be used to construct a union of two ***set***s. The two ***set***s are specified by iterator pairs, and the union is copied into an output iterator that is supplied as a fifth argument. To form the result as a ***set***, an *insert iterator* must be used to form the output iterator (Section 2.4). If the desired outcome is a union of one ***set*** with another, then a temporary ***set*** can be constructed, and the results swapped with the argument ***set*** prior to deletion of the temporary ***set***:

```
// union two sets, copying result into a vector
vector<int> v_one (set_one.size() + set_two.size());

set_union(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end(), v_one.begin());

// form union in place
set<int> temp_set;
set_union(set_one.begin(), set_one.end(),
   set_two.begin(), set_two.end(),
   inserter(temp_set, temp_set.begin()));
set_one.swap(temp_set);  // temp_set will be deleted
```

The function `set_intersection()` is similar, and forms the intersection of the two ***set***s.

As with the `includes()` function, the less-than operator `<` is used to compare elements in the two argument ***set***s, regardless of the operator provided in the declaration of the ***set***s. Should this be inappropriate, alternative versions of both the `set_union()` or `set_intersection()` functions permit the comparison operator used to form the ***set*** to be given as a sixth argument.

The operation of taking the union of two ***multiset***s should be distinguished from the operation of merging two ***set***s. Imagine that one argument ***set*** contains three instances of the element 7, and the second ***set*** contains two instances of the same value. The union will contain only three such values, while the merge will contain five. To form the merge, the function `merge()` can be used (Section 14.5). The arguments to this function exactly match those of the `set_union()` function.

Set Difference

There are two forms of ***set*** difference. A simple ***set*** difference represents the elements in the first ***set*** that are not contained in the second. A symmetric ***set*** difference is the union of the elements in the first ***set*** that are not contained in the second, with the elements in the second that are not contained in the first. These two values are constructed by the functions `set_difference()` and `set_symmetric_difference()`, respectively. The use of these functions is similar to the use of the `set_union()` function described earlier.

Other Generic Algorithms

Because ***set***s are ordered and have constant iterators, a number of the generic functions described in Chapter 13 and Chapter 14 are either not applicable to ***set***s or not particularly useful. However, Table 14 gives a few of the functions that can be used in conjunction with the ***set*** datatype.

Table 14 -- Functions useful for the set datatype

Purpose	Name	Where to find
Copy one sequence into another	`copy`	Section 13.2.2
Find an element that matches a condition	`find_if`	Section 13.3.1
Find a sub-sequence within a set	`search`	Section 13.3.3
Count number of elements that satisfy condition	`count_if`	Section 13.6.1
Reduce set to a single value	`accumulate`	Section 13.6.2
Execute function on each element	`for_each`	Section 13.8

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Example Program: A Spelling Checker

A simple example program that uses a ***set*** is a spelling checker.

NOTE: This program can be found in the file spell.cpp.

The checker takes as arguments two input streams: the first represents a stream of correctly spelled words (that is, a dictionary), and the second a text file. To begin, the dictionary is read into a set. This is performed with a `copy()` function and an input stream iterator, copying the values into an inserter for the dictionary. Next, words from the text are examined one by one, to see if they are in the dictionary. If they are not, they are added to a set of misspelled words. After the entire text has been examined, the program outputs the list of misspelled words.

```
void spellCheck (istream & dictionary, istream & text)
{
   typedef set <string, less<string> > stringset;
   stringset words, misspellings;
   string word;
   istream_iterator<string, ptrdiff_t> dstream(dictionary), eof;

   // first read the dictionary
   copy (dstream, eof, inserter(words, words.begin()));

   // next read the text
   while (text >> word)
      if (! words.count(word))
         misspellings.insert(word);

   // finally, output all misspellings
   cout << "Misspelled words:" << endl;
   copy (misspellings.begin(), misspellings.end(),
      ostream_iterator<string>(cout, "\n"));
}
```

An improvement would be to suggest alternative words for each misspelling. There are various heuristics that can be used to discover alternatives. The technique we use here is to simply exchange adjacent letters. To find these, a call on the following function is inserted into the loop that displays the misspellings:

```
{
   for (int I = 1; I < word.length(); I++) {
      swap(word[I-1], word[I]);
      if (words.count(word))
         cout << "Suggestion: " << word << endl;
      // put word back as before
      swap(word[I-1], word[I]);
      }
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The bitset Abstraction

A ***bitset*** is really a cross between a ***set*** and a ***vector***. Like the vector abstraction ***vector<bool>***, the abstraction represents a set of binary (0/1 bit) values. However, set operations can be performed on bitsets using the logical bit-wise operators. The class ***bitset*** does not provide any iterators for accessing elements.

Include Files

The ***bitset*** header file must appear in al programs that use the bitset datatype:

```
#include <bitset>
```

Declaration and Initialization of bitset

A ***bitset*** is a template class abstraction. However, the template argument is not a type, but an integer value. The value represents the number of bits the set contains.

```
bitset<126> bset_one;        // create a set of 126 bits
```

An alternative technique permits the size of the set to be specified as an argument to the constructor. The actual size will be the smaller of the value used as the template argument and the constructor argument. This technique is useful when a program contains two or more bit vectors of differing sizes. Consistently using the larger size for the template argument means that only one set of methods for the class is generated. The actual size, however, is determined by the constructor.

```
bitset<126> bset_two(100);   // this set has only 100 elements
```

A third form of constructor takes as argument a string of 0 and 1 characters. A ***bitset*** is created that has as many elements as there are characters in the string, and is initialized with the values from the string.

```
bitset<126> small_set("10101010");   // this set has 8 elements
```

Accessing and Testing Elements

An individual bit in the ***bitset*** can be accessed using the subscript operation. Whether the bit is one or not can be determined using the member function `test()`. Whether any bit in the bitset is *on* is tested using the member function `any()`, which yields a boolean value. The inverse of `any()` is returned by the member function `none()`:

```
bset_one[3] = 1;
if (bset_one.test(4))
   cout << "bit position 4 is set" << endl;
if (bset_one.any())
   cout << "some bit position is set" << endl;
if (bset_one.none()) cout << "no bit position is set" << endl;
```

The function `set()` can be used to set a specific bit. The function `bset_one.set(I)` is equivalent to `bset_one[I] = true`. Invoking the function without any arguments sets all bit positions to true. The function `reset()` is similar, and sets the indicated positions to false, or all positions to false if invoked with no argument. The function `flip()` flips either the indicated position, or all positions if no argument is provided. The function `flip()` is also provided as a member function for the individual bit references.

```
bset_one.flip();         // flip the entire set
bset_one.flip(12);       // flip only bit 12
bset_one[12].flip();     // reflip bit 12
```

The member function `size()` returns the size of the bitset, while the member function `count()` yields the number of bits that are set.

set Operations

©Copyright 1999 Rogue Wave Software, Inc.

The set operations on ***bitset***s are implemented using the bit-wise operators, analogous to the way the same operators act on integer arguments.

The negation operator `~` applied to a ***bitset*** returns a new ***bitset*** containing the inverse of the elements in the argument set.

The intersection of two ***bitset***s is formed using the *and* operator `&`. The assignment form of the operator can also be used. In the assignment form, the target becomes the disjunction of the two sets:

```
bset_three = bset_two & bset_four;
   bset_five &= bset_three;
```

The union of two sets is formed in a similar manner using the *or* operator `|`. The *exclusive-or* is formed using the bit-wise exclusive or operator `^`.

The left and right shift operators, `<<` and `>>`, can be used to shift a ***bitset*** left or right as they are used on integer arguments. If a bit is shifted left by an integer value `n`, then the new bit position `I` is the value of the former `I-n`. Zeros are shifted into the new positions.

Conversions

The member function `to_ulong()` converts a ***bitset*** into an `unsigned long`. It is an error to perform this operation on a ***bitset*** containing more elements than can fit into this representation.

The member function `to_string()` converts a ***bitset*** into an object of type ***string***. The string has as many characters as the bitset. Each zero bit corresponds to the character `0`, while each one bit is represented by the character `1`.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 9: map and multimap

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

The map Data Abstraction

A ***map*** is an indexed data structure, similar to a ***vector*** or a ***deque***. However, a ***map*** differs from a ***vector*** or ***deque*** in two important respects:

- First, in a ***map*** the index values or *key values* need not be `integer`, but can be any ordered datatype. For example, a ***map*** can be indexed by real numbers, or by strings. Any datatype for which a comparison operator can be defined can be used as a key. As with a ***vector*** or ***deque***, elements can be accessed through the subscript operator or other techniques.

- Second, a ***map*** is an ordered data structure. This means that elements are maintained in sequence, the ordering determined by key values. Because ***map***s maintain values in order, they can very rapidly find the element specified by any given key. Searching is performed in logarithmic time. Like a ***list***, maps are not limited in size, but expand or contract as necessary as new elements are added or removed. In large part, a ***map*** can simply be considered a ***set*** that maintains a collection of pairs.

In other programming languages, a map-like data structure is sometimes referred to as a dictionary, a table, or an associative array. In the Standard C++ Library, there are two varieties of ***map***s:

- The ***map*** data structure demands unique keys; that is, there is a one-to-one association between key elements and their corresponding values. In a ***map***, the insertion of a new value that uses an existing key is ignored.

- The ***multimap*** permits multiple different entries to be indexed by the same key.

Both data structures provide relatively fast insertion, deletion, and access operations in logarithmic time. For a description of the ***pair*** datatype, see the discussion of insertion in Section 8.2.3.

Include files

Whenever you use a ***map*** or a ***multimap***, you must include the `map` header file.

```
# include <map>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

map and multimap Operations

The member functions provided by the ***map*** and ***multimap*** datatypes will shortly be described in more detail. Note that while member functions provide basic operations, the utility of the data structure is greatly extended through the use of the generic algorithms described in Part IV.

Declaration and Initialization of map

The declaration of a ***map*** follows the pattern we have seen repeatedly in the Standard C++ Library. A ***map*** is a template data structure, specialized by the type of the key elements, the type of the associated values, and the operator to be used in comparing keys. The last of these is optional. You don't need to supply it if your compiler supports default template types, a relatively new feature in C++ not yet supported by all vendors; if not, the less-than operator for the key type is assumed.

A ***map*** can be declared with no initial elements, or initialized from another container by providing a pair of iterators. In the latter case, the iterators must denote values of type ***pair***; the first field in each pair is taken to be a key, while the second field is a value. A copy constructor also permits ***map***s to be created as copies of other ***map***s:

```
// map indexed by doubles containing strings
 map<double, string, less<double> > map_one;
// map indexed by integers, containing integers
map<int, int> map_two(aContainer.begin(), aContainer.end());
// create a new map, initializing it from map two
map<int, int> map_three (map_two);   // copy constructor
```

A ***map*** can be assigned to another ***map***, and two ***map***s can exchange their values using the `swap()` operation, like the other Standard C++ Library containers.

Type Definitions

The classes ***map*** and ***multimap*** include a number of type definitions, most commonly used in declaration statements. For example, an iterator for a ***map*** of `strings` to `integers` can be declared as follows:

```
map<string, int>::iterator location;
```

In addition to `iterator`, Table 15 defines the following types:

Table 15 -- Type definitions for the classes map and multimap

Type	Definition
`key_type`	The type associated with the keys used to index the map.
`value_type`	The type held by the container, a key/value pair.
`mapped_type`	The type associated with values.
`const_iterator`	An iterator that does not allow modification of the underlying sequence.
`reverse_iterator`	An iterator that moves in a backward direction.
`const_reverse_iterator`	A combination constant and reverse iterator.
`reference`	A reference to an underlying value.
`const_reference`	A reference to an underlying value that will not permit the element to be modified.
`size_type`	An unsigned integer type, used to refer to the size of containers.
`key_compare`	A function object that can be used to compare two keys.
`value_compare`	A function object that can be used to compare two elements.
`difference_type`	A signed integer type, used to describe the distances between iterators.
`allocator_type`	An allocator used by the container for all storage management.

Insertion and Access

©Copyright 1999 Rogue Wave Software, Inc.

Values can be inserted into a ***map*** or a ***multimap*** using the `insert()` operation. Note that the argument must be a key-value ***pair***. This ***pair*** is often constructed using the datatype `value_type` associated with the map:

```
map_three.insert (map<int>::value_type(5, 7));
```

Insertions can also be performed using an iterator pair, for example, as generated by another ***map***.

```
map_two.insert (map_three.begin(), map_three.end());
```

With a ***map***, but not a ***multimap***, values can be accessed and inserted using the subscript operator. Simply using a key as a subscript creates an entry--the default element is used as the associated value. Assigning to the result of the subscript changes the associated binding.

```
cout << "Index value 7 is " << map_three[7] << endl;
   // now change the associated value
map_three[7] = 5;
cout << "Index value 7 is " << map_three[7] << endl;
```

Removal of Values

Values can be removed from a ***map*** or a ***multimap*** by naming the key value. In a ***multimap***, the erasure removes all elements with the associated key. An element to be removed can also be denoted by an iterator, like the iterator yielded by a `find()` operation. A pair of iterators can be used to erase an entire range of elements:

```
// erase the 4th element 4
map_three.erase(4);
// erase the 5th element
mtesttype::iterator five = map_three.find(5);
map_three.erase(five);

// erase all values between the 7th and 11th elements
mtesttype::iterator seven = map_three.find(7);
mtesttype::iterator eleven = map_three.find(11);
map_three.erase (seven, eleven);
```

If the underlying element type provides a destructor, the destructor is invoked prior to removing the key and value pair from the collection.

Iterators

The member functions `begin()` and `end()` produce bidirectional iterators for both ***map***s and ***multimap***s. Dereferencing an iterator for either a ***map*** or a ***multimap*** yields a *pair* of key/value elements. The field names `first` and `second` can be applied to these values to access the individual fields. The first field is constant, and cannot be modified. The second field, however, can be used to change the value being held in association with a given key. Elements are generated in sequence, based on the ordering of the key fields.The member functions `rbegin()` and `rend()` produce iterators that yield the elements in reverse order.

NOTE: Unlike a vector or deque, the insertion or removal of elements from a map does not invalidate iterators which may be referencing other portions of the container.

Searching and Counting

The member function `size()` yields the number of elements held by a container. The member function `empty()` returns a boolean true value if the container is empty, and is generally faster than testing the size against zero.

The member function `find()` takes a key argument, and returns an iterator denoting the associated key/value pair. For ***multimap***s, the first such value is returned. In both cases, the past-the-end iterator is returned if no such value is found:

```
if (map_one.find(4) != map_one.end())
   cout << "contains a 4th element" << endl;
```

The member function `lower_bound()` yields the first entry that matches the argument key, while the member function `upper_bound()` returns the first value past the last entry matching the argument. Finally, the member function `equal_range()` returns a pair of iterators, holding the lower and upper bounds. An example showing the use of these procedures is presented later in this chapter.

©Copyright 1999 Rogue Wave Software, Inc.

The member function `count()` returns the number of elements that match the key value supplied as the argument. For a ***map***, this value is always either zero or one, whereas for a ***multimap*** it can be any nonnegative value. If you simply want to determine whether or not a collection contains an element indexed by a given key, using `count()` is often easier than using the `find()` function and testing the result against the end-of-sequence iterator:

```
if (map_one.count(4))
   cout << "contains a 4th element" << endl;
```

Element Comparisons

The member functions `key_comp()` and `value_comp()`, which take no arguments, return function objects that can be used to compare elements of the key or value types. Values used in these comparisons need not be contained in the collection, and neither function has any effect on the container.

```
if (map_two.key_comp (i, j))
   cout << "element i is less than j" << endl;
```

Other map Operations

Because ***map***s and ***multimap***s are ordered collections, and because the iterators for maps return ***pair***s, many of the functions described in Part IV are meaningless or difficult to use. However, there are a few notable exceptions. The functions `for_each()`, `adjacent_find()`, and `accumulate()` each have their own uses. In all cases it is important to remember that the functions supplied as arguments should take a key/value pair as arguments.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example Programs

In this section, we present three example programs that illustrate the use of ***map***s and ***multimap***s. These examples deal with a telephone database, graphs, and a concordance.

Example: A Telephone Database

NOTE: The complete example program is in the file tutorial tele.cpp.

A maintenance program for a simple telephone database is a good application for a ***map***. The database is simply an indexed structure, where the name of the person or business (a `string`) is the key value, and the telephone number (a `long`) is the associated entry. We might write such a class as follows:

```
typedef map<string, long, less<string> > friendMap;
typedef friendMap::value_type entry_type;

class telephoneDirectory {
public:
   void addEntry (string name, long number)   // add new entry to
                                              // database
      { database[name] = number; }

   void remove (string name)   // remove entry from database
      { database.erase(name); }

   void update (string name, long number)   // update entry
      { remove(name); addEntry(name, number); }

   void displayDatabase()   // display entire database
      { for_each(database.begin(), database.end(), printEntry); }

   void displayPrefix(int);   // display entries that match prefix

   void displayByPrefix();   // display database sorted by prefix

private:
   friendMap database;
};
```

Simple operations on our database are directly implemented by map commands. Adding an element to the database is simply an `insert`, removing an element is an `erase`, and updating is a combination of the two. To print all the entries in the database we can use the `for_each()` algorithm, and apply the following simple utility routine to each entry:

```
void printEntry(const entry_type & entry)
   { cout << entry.first << ":" << entry.second << endl; }
```

We now use a pair of slightly more complex operations to illustrate how a few of the algorithms described in Chapter 13 can be used with maps. Suppose we want to display all the phone numbers with a certain three-digit initial prefix[1] we use the `find_if()` function, which is not to be confused with the `find()` member function in class ***map***, to locate the first entry. Starting from this location, subsequent calls on `find_if()` uncover each successive entry:

```
void telephoneDirectory::displayPrefix(int prefix)
{
   cout << "Listing for prefix " << prefix << endl;
   friendMap::iterator where;
   where =
      find_if (database.begin(), database.end(),
             checkPrefix(prefix));
   while (where != database.end()) {
      printEntry(*where);
      where = find_if (++where, database.end(),
             checkPrefix(prefix));
      }
   cout << "end of prefix listing" << endl;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

For the predicate to this operation, we require a boolean function that takes only a single argument, the ***pair*** representing a database entry, and tells us whether or not it is in the given prefix. There is no obvious candidate function, and in any case the test prefix is not being passed as an argument to the comparison function. The solution to this problem is to employ a common Standard C++ Library technique: define the predicate function as an instance of a class, and store the test predicate as an instance variable in the class, initialized when the class is constructed. The desired function is defined as the function call operator for the class:

```
int prefix(const entry_type & entry)
   { return entry.second / 10000; }

class checkPrefix {
public:
   checkPrefix (int p) : testPrefix(p) { }
   int testPrefix;
   bool operator () (const entry_type & entry)
      { return prefix(entry) == testPrefix; }
};
```

Our final example displays the directory sorted by prefix. It is not possible to alter the order of the ***map***s themselves. Instead, we create a new ***map*** with the element types reversed and copy the values into the new ***map***, which orders the values by prefix. Once the new ***map*** is created, it is printed:

```
typedef map<long, string, less<long> > sortedMap;
typedef sortedMap::value_type sorted_entry_type;

void telephoneDirectory::displayByPrefix()
{
   cout << "Display by prefix" << endl;
   sortedMap sortedData;
   friendMap::iterator itr;
   for (itr = database.begin(); itr != database.end(); itr++)
      sortedData.insert(sortedMap::value_type((*itr).second,
            (*itr).first));
   for_each(sortedData.begin(), sortedData.end(),
            printSortedEntry);
}
```

Here is the function used to print the sorted entries:

```
void printSortedEntry (const sorted_entry_type & entry)
      { cout << entry.first << ":" << entry.second << endl; }
```

An Example: Graphs

NOTE: The executable version of this program is in the file graph.cpp.

A ***map*** whose elements are themselves ***map***s is a natural representation for a directed graph. For example, suppose we use strings to encode the names of cities, and we wish to construct a map where the value associated with an edge is the distance between two connected cities. We could create such a graph as follows:

```
typedef map<string, int> stringVector;
typedef map<string, stringVector> graph;

const string pendleton("Pendleton");   // define strings for
                                       // city names
const string pensacola("Pensacola");
const string peoria("Peoria");
const string phoenix("Phoenix");
const string pierre("Pierre");
const string pittsburgh("Pittsburgh");
const string princeton("Princeton");
const string pueblo("Pueblo");

graph cityMap;      // declare the graph that holds the map

cityMap[pendleton][phoenix] = 4;   // add edges to the graph
cityMap[pendleton][pueblo] = 8;
cityMap[pensacola][phoenix] = 5;
cityMap[peoria][pittsburgh] = 5;
cityMap[peoria][pueblo] = 3;
cityMap[phoenix][peoria] = 4;
cityMap[phoenix][pittsburgh] = 10;
cityMap[phoenix][pueblo] = 3;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
cityMap[pierre][pendleton] = 2;
cityMap[pittsburgh][pensacola] = 4;
cityMap[princeton][pittsburgh] = 2;
cityMap[pueblo][pierre] = 3;
```

The type ***stringVector*** is a map of `integers` indexed by `strings`. The type ***graph*** is, in effect, a two-dimensional sparse array, indexed by `strings` and holding `integer` values. A sequence of assignment statements initializes the graph.

A number of classic algorithms can be used to manipulate graphs represented in this form. One example is Dijkstra's shortest-path algorithm. Dijkstra's algorithm begins from a specific city given as an initial location. A ***priority_queue*** of distance/city ***pair***s is then constructed, and initialized with the distance from the starting city to itself (namely, zero). The definition for the distance ***pair*** datatype is as follows:

```
struct DistancePair {
   unsigned int first;
   string second;
   DistancePair() : first(0) { }
   DistancePair(unsigned int f, const string & s)
      : first(f), second(s) { }
};

bool operator < (const DistancePair & lhs, const DistancePair & rhs)
   { return lhs.first < rhs.first; }
```

In the algorithm that follows, note how the conditional test is reversed on the ***priority_queue***, because at each step we wish to pull the smallest, and not the largest, value from the collection. On each iteration around the loop we pull a city from the queue. If we have not yet found a shorter path to the city, the current distance is recorded, and we can compute the distance from this city to each of its adjacent cities by examining the graph. This process continues until the priority_queue becomes exhausted:

```
void shortestDistance(graph & cityMap,
      const string & start, stringVector & distances)
{
   // process a priority queue of distances to cities
   priority_queue<DistancePair, vector<DistancePair>,
                  greater<DistancePair> > que;
   que.push(DistancePair(0, start));

   while (! que.empty()) {
      // pull nearest city from queue
      int distance = que.top().first;
      string city = que.top().second;
      que.pop();
              // if we haven't seen it already, process it
      if (0 == distances.count(city)) {
                // then add it to shortest distance map
         distances[city] = distance;
                // and put values into queue
         const stringVector & cities = cityMap[city];
         stringVector::const_iterator start = cities.begin();
         stringVector::const_iterator stop = cities.end();
         for (; start != stop; ++start)
            que.push(DistancePair(distance + (*start).second,
                                  (*start).first));
         }
      }
}
```

Notice that this relatively simple algorithm makes use of ***vector***s, ***map***s, ***string***s, and ***priority_queue***s (Chapter 11).

Example: A Concordance

A concordance is an alphabetical listing of words in a text that shows the line numbers on which each word occurs.

We develop a concordance to illustrate the use of the ***map*** and ***multimap*** container classes. The data values are maintained in the concordance by a ***multimap***, indexed by `strings` (the words) and hold `integers` (the line numbers). A ***multimap*** is employed because the same word often appears on multiple different lines; indeed, discovering such connections is one of the primary purposes of a concordance. Another possibility would be to use a ***map*** and use a ***set*** of integer elements as the associated values.

```
class concordance {
   typedef multimap<string, int less <string> > wordDictType;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
public:
   void addWord (string, int);
   void readText (istream &);
   void printConcordance (ostream &);

private:
   wordDictType wordMap;
};
```

The creation of the concordance is divided into two steps: the program generates the concordance by reading lines from an input stream, then prints the result on the output stream. This is reflected in the two member functions `readText()` and `printConcordance()`. The first of these, `readText()`, is written as follows:

```
void concordance::readText (istream & in)
{
   string line;
   for (int i = 1; getline(in, line, "\n"); i++) {
      allLower(line);
      list<string> words;
      split (line, " ,.;:", words);
      list<string>::iterator wptr;
      for (wptr = words.begin(); wptr != words.end(); ++wptr)
         addWord(*wptr, i);
      }
}
```

Lines are read from the input stream one by one. The text of the line is first converted into lower case, then the line is split into words using the function `split()` described in Section 12.3. Each word is then entered into the concordance. The method used to enter a value into the concordance is as follows:

```
void concordance::addWord (string word, int line)
{
   // see if word occurs in list
   // first get range of entries with same key
   wordDictType::iterator low = wordMap.lower_bound(word);
   wordDictType::iterator high = wordMap.upper_bound(word);
   // loop over entries, see if any match current line
   for ( ; low != high; ++low)
      if ((*low).second == line)
         return;
   // didn't occur, add now
   wordMap.insert(wordDictType::value_type(word, line));
}
```

The major portion of `addWord()` is concerned with ensuring that values are not duplicated in the word *map* if the same word occurs twice on the same line. To assure this, the range of values matching the key is examined, each value is tested, and if any match the line number then no insertion is performed. It is only if the loop terminates without discovering the line number that the new word/line number pair is inserted.

The final step is to print the concordance. This is performed in the following fashion:

```
void concordance::printConcordance (ostream & out)
{
   string lastword("");
   wordDictType::iterator pairPtr;
   wordDictType::iterator stop = wordMap.end();
   for (pairPtr = wordMap.begin(); pairPtr != stop; ++pairPtr)
         // if word is same as previous, just print line number
      if (lastword == (*pairPtr).first)
         out << " " << (*pairPtr).second;
      else {   // first entry of word
         lastword = (*pairPtr).first;
         cout << endl << lastword << ": " << (*pairPtr).second;
         }
   cout << endl; // terminate last line
}
```

An iterator loop is used to cycle over the elements being maintained by the word list. Each new word generates a new line of output; thereafter, line numbers appear separated by spaces. For example, if the input was the text:

```
It was the best of times,
it was the worst of times.
```

The output, from best to worst, would be:

©Copyright 1999 Rogue Wave Software, Inc.

```
best:  1
it:    1 2
of:    1 2
the:   1 2
times: 1 2
was:   1 2
worst: 1
```

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 10: The Container Adaptors stack and queue

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Overview

Most people have a good intuitive understanding of the ***stack*** and ***queue*** data abstractions based on experience with everyday objects. An excellent example of a stack is a pile of papers on a desk, or a stack of dishes in a cupboard. In both cases, the important characteristic is that the item on the top is most easily accessed. The easiest way to add a new item to the collection is to place it above all the current items in the stack. In this manner, an item removed from a stack is the item that has been most recently inserted into the stack; for example, the top piece of paper in the pile, or the top dish in the stack.

An everyday example of a queue, on the other hand, is a bank teller line, or a line of people waiting to enter a theater. Here new additions are made to the back of the queue, as new people enter the line, while items are removed from the front of the structure, as patrons enter the theater. The removal order for a queue is the opposite of that for a stack. In a queue, the item that is removed is the element that has been present in the queue for the longest period of time.

Among some groups of developers, a stack is referred to as a LIFO structure, and a queue is called a FIFO structure. The abbreviation *LIFO* stands for *Last In, First Out*. This means the first entry removed from a stack is the last entry that was inserted. The term *FIFO*, on the other hand, is short for *First In, First Out*. This means the first element removed from a queue is the first element that was inserted into the queue.

In the Standard C++ Library, both ***stack***s and ***queue***s are *adaptors*, built on top of other containers that actually hold the values. A ***stack*** can be built out of a ***vector***, a ***list***, or a ***deque***, while a ***queue*** can be built on top of either a ***list*** or a ***deque***. Elements held by either a ***stack*** or ***queue*** must recognize both the operators < and ==.

Because neither ***stacks*** nor ***queues*** define iterators, it is not possible to examine the elements of the collection except by removing the values one by one. The fact that these structures do not implement iterators also implies that most of the generic algorithms described in Part IV cannot be used with either data structure.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The stack Data Abstraction

As a data abstraction, a ***stack*** is traditionally defined as any object that implements the operations defined in Table 16:

Table 16 -- Stack operations

Function	Implemented operation
`empty()`	Returns true if the collection is empty
`size()`	Returns number of elements in collection
`top()`	Returns (but does not remove) the topmost element in the stack
`push(newElement)`	Pushes a new element onto the stack
`pop()`	Removes (but does not return) the topmost element from the stack

Note that accessing the front element and removing the front element are separate operations.

Include Files

Programs that use the ***stack*** data abstraction should include the file `stack`:

```
# include <stack>
```

Declaration and Initialization of stack

A declaration for a ***stack*** must specify the underlying element type; it can also specify the container that will hold the elements. For a ***stack*** the default container is a ***deque***, but a ***list*** or ***vector*** can also be used. The ***vector*** version is generally smaller, while the ***deque*** version may be slightly faster.

The following are sample declarations for a stack:

```
stack<int> stackOne;                        // stack using deque
stack< double, deque<double> > stackTwo;
stack< Part *, list<Part * > > stackThree;
stack< Customer, list<Customer> > stackFour;
```

The last example creates a ***stack*** from a user-defined type named ***Customer***.

NOTE: On most compilers it is important to leave a space between the two right angle brackets in the declaration of a stack, as shown in the example, or they are interpreted by the compiler as a right shift operator.

Example Program: An RPN Calculator

A classic application of a ***stack*** is in the implementation of this calculator.

NOTE: This program is in the file calc.cpp.

Input to the calculator consists of a text `string` that represents an expression written in reverse polish notation (RPN). *Operands*, called *integer constants*, are pushed on a ***stack*** of values. As operators are encountered, the appropriate number of operands are popped off the stack, the operation is performed, and the result is pushed back on the stack.

We can divide the development of our ***stack*** simulation into two parts, a *calculator engine* and a *calculator program*. A *calculator engine* is concerned with the actual work involved in the simulation, but does not perform any input or output operations. The name is intended to suggest an analogy to a car engine or a computer processor: the mechanism performs the actual work, but the user of the mechanism does not normally directly interact with it. Wrapped around this is the *calculator program*, which interacts with the user and passes appropriate instructions to the calculator engine.

We can use the following class definition for our calculator engine. Inside the class declaration we define an enumerated ***list*** of values to represent each of the possible operators that the calculator is prepared to accept. We have made two

©Copyright 1999 Rogue Wave Software, Inc.

simplifying assumptions: all operands will be integer values, and only binary operators will be handled.

```
class calculatorEngine {
public:
   enum binaryOperator {plus, minus, times, divide};

   int currentMemory ()           // return current top of stack
      { return data.top(); }

   void pushOperand (int value)   // push operand value on to stack
      { data.push (value); }

   void doOperator (binaryOperator);   // pop stack and perform
                                       // operator

protected:
   stack< int > data;
};
```

The member function `doOperator()` performs the actual work. It pops values from the stack2, performs the operation, then pushes the result back onto the stack:

```
void calculatorEngine::doOperator (binaryOperator theOp)
{
   int right = data.top();             // read top element
   data.pop();                         // pop it from stack
   int left = data.top();              // read next top element
   data.pop();   // pop it from stack
   switch (theOp) {
      case plus: data.push(left + right); break;
      case minus: data.push(left - right); break;
      case times: data.push(left * right); break;
      case divide: data.push(left / right); break;
      }
}
```

The main program reads values in reverse polish notation, invoking the calculator engine to do the actual work:

```
void main() {
   int intval;
   calculatorEngine calc;
   char c;

   while (cin >> c)
      switch (c) {
         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':
            cin.putback(c);
            cin >> intval;
            calc.pushOperand(intval);
            break;

         case '+':  calc.doOperator(calculatorEngine::plus);
            break;

         case '-':  calc.doOperator(calculatorEngine::minus);
            break;

         case '*':  calc.doOperator(calculatorEngine::times);
            break;

         case '/':  calc.doOperator(calculatorEngine::divide);
            break;

         case 'p':  cout << calc.currentMemory() << endl;
            break;

         case 'q':  return; // quit program
      }
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The queue Data Abstraction

As a data abstraction, a ***queue*** is traditionally defined as any object that implements the following operations given in Table 17:

Table 17 -- Queue operations

Function	Implemented operation
`empty()`	Returns true if the collection is empty
`size()`	Returns number of elements in collection
`front()`	Returns (but does not remove) the element at the front of the queue
`back()`	Returns the element at the end of the queue
`push(newElement)`	Pushes a new element on to the end of the queue
`pop()`	Removes (but does not return) the element at the front of the queue

Note that the operations of accessing and of removing the front elements are performed separately.

Include Files

Programs that use the ***queue*** data abstraction should include the file `queue`:

```
# include <queue>
```

Declaration and Initialization of queue

A declaration for a ***queue*** must specify the element type, and can also specify the container that will hold the values. For a ***queue*** the default container is a ***deque***, but a list can also be used. The ***list*** version is generally smaller, while the ***deque*** version may be slightly faster. The following are sample declarations for a ***queue***:

```
queue< int, list<int> > queueOne;
queue< double> > queueTwo;                      // uses a deque
queue< Part *, list<Part * > > queueThree;
queue< Customer, list<Customer> > queueFour;
```

The last example creates a ***queue*** of a user-defined type named ***Customer***. As with the ***stack*** container, all objects stored in a ***queue*** must understand the operators `<` and `==`.

Because the ***queue*** does not implement an iterator, none of the generic algorithms described in Part IV apply to ***queue***s.

Example Program: Bank Teller Simulation

NOTE: The complete version of the bank teller simulation program is in teller.cpp.

Queues are often found in businesses, such as supermarkets or banks. Suppose you are the manager of a bank, and you need to determine how many tellers to have working during certain hours. You decide to create a computer simulation, basing your simulation on certain observed behavior. For example, you note that during peak hours there is a ninety percent chance that a customer will arrive every minute.

We create a simulation by first defining objects to represent both customers and tellers. For customers, the information we want to know is the average amount of time they spend waiting in line. Thus, customer objects simply maintain two integer data fields: the time they arrive in line, and the time they spend at the counter. The latter is a value randomly selected between 2 and 8. (See Section 2.2.5 for a discussion of the `randomInteger()` function.)

```
class Customer {
public:
   Customer (int at = 0) : arrival_Time(at),
         processTime(2 + randomInteger(6)) {}
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   int arrival_Time;
   int processTime;

   bool done()      // are we done with our transaction?
      { return --processTime < 0; }

   operator < (const Customer & c)   // order by arrival time
      { return arrival_Time < c.arrival_Time; }

   operator == (const Customer & c)   // no two customers are alike
      { return false; }
};
```

Because objects can only be stored in Standard C++ Library containers if they can be compared for equality and ordering, it is necessary to define the operators < and == for customers. Customers can also tell us when they are done with their transactions.

Tellers are either busy servicing customers, or they are free. Thus, each teller value holds two data fields: a customer, and a boolean flag. Tellers define a member function to answer whether they are free or not, as well as a member function that is invoked when they start servicing a customer.

```
class Teller {
public:
   Teller() { free = true; }

   bool isFree()   // are we free to service new customer?
      { if (free) return true;
        if (customer.done())
           free = true;
        return free;
       }

   void addCustomer(Customer c)   // start serving new customer
      {   customer = c;
         free = false;
      }

private:
   bool free;
   Customer customer;
};
```

The main program, then, is a large loop cycling once each simulated minute. The probability is 0.9 that each minute a new customer is entered into the queue of waiting customers. Each teller is polled, and if any are free they take the next customer from the queue. Counts are maintained of the number of customers serviced and the total time they spent in queue. From these two values we can determine, following the simulation, the average time a customer spent waiting in the line.

```
void main() {
   int numberOfTellers = 5;
   int numberOfMinutes = 60;
   double totalWait = 0;
   int numberOfCustomers = 0;
   vector<Teller> teller(numberOfTellers);
   queue< Customer > line;

   for (int time = 0; time < numberOfMinutes; time++) {
      if (randomInteger(10) < 9)
         line.push(Customer(time));
      for (int i = 0; i < numberOfTellers; i++) {
         if (teller[i].isFree() & ! line.empty()) {
            Customer & frontCustomer = line.front();
            numberOfCustomers++;
            totalWait += (time - frontCustomer.arrival_Time);
            teller[i].addCustomer(frontCustomer);
            line.pop();
            }
         }
      }
   cout << "average wait:" <<
          (totalWait / numberOfCustomers) << endl;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

By executing the program several times, using various values for the number of tellers, the manager can determine the smallest number of tellers that can service the customers while maintaining the average waiting time at an acceptable level.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 11: The Container Adaptor priority queue

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The priority queue Data Abstraction

A ***priority_queue*** is a data structure useful in problems where you need to rapidly and repeatedly find and remove the largest element from a collection of values. An everyday example of a priority queue is the *to do* list of tasks waiting to be performed that most of us maintain to keep ourselves organized. Some jobs, such as *clean desktop*, are not imperative and can be postponed arbitrarily. Other tasks, such as *finish report by Monday* or *buy flowers for anniversary*, are time-crucial and must be addressed more rapidly. Thus, we sort the tasks waiting to be accomplished in order of their importance, or perhaps based on a combination of their critical importance, their long term benefit, and the fun we will have doing them, and choose the most pressing.

A more computer-related example of a priority queue is the list of pending processes maintained by an operating system, where the value associated with each element is the priority of the job. For example, it may be necessary to respond rapidly to a key pressed at a terminal before the data is lost when the next key is pressed. On the other hand, the process of copying a listing to a queue of output waiting to be handled by a printer is something that can be postponed for a short period, as long as it is handled eventually. By maintaining processes in a priority queue, those jobs with urgent priority are executed prior to any jobs with less urgent requirements.

Simulation programs use a priority queue of *future events*. The simulation maintains a virtual *clock*, and each event has an associated time when the event will take place. In such a collection, the element with the smallest time value is the next event that should be simulated. These are only a few instances of the types of problems for which a ***priority_queue*** is a useful tool. You probably have encountered others, or you soon will.

Some developers may feel the term priority *queue* is a misnomer. The data structure is not a ***queue*** in the sense that we used the term in Chapter 10, since it does not return elements in a strict first-in, first-out sequence. Nevertheless, the name is now firmly associated with this particular datatype.

Include Files

Programs that use the priority queue data abstraction should include the file `queue`:

```
# include <queue>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The priority queue Operations

A ***priority_queue*** is a data structure that can hold elements of type `T` and implement the five operations given in Table 18:

Table 18 -- priority_queue operations

Function	Implemented operation
`push(T)`	Adds a new value to the collection being maintained
`top()`	Returns a reference to the smallest element in the collection
`pop()`	Deletes the smallest element from the collection
`size()`	Returns the number of elements in the collection
`empty()`	Returns true if the collection is empty

Elements of type `T` must be comparable to each other, either through the use of the default less-than operator `<`, or through a comparison function passed either as a template argument or as an optional argument on the constructor. The latter form will be illustrated in the example program provided later in this chapter. As with all the containers in the Standard Library, there are several constructors. The *default constructor* requires either no arguments or the optional comparison function. An *alternative constructor* takes an iterator ***pair***, and initializes the values in the container from the argument sequence. Once more, an optional third argument can be used to define the comparison function.

The ***priority_queue*** datatype is built on top of a container class, which is the structure actually used to maintain the values in the collection. There are two containers in the Standard C++ Library that can be used to construct ***priority_queues***: ***vector***s or ***deque***s. By default, a ***priority_queue*** will use ***vector***.

Declaration and Initialization of priority queue

The following illustrates the declaration of several ***priority_queue***s[3]:

```
priority_queue< int > queue_one;      //uses vector and less<int>
priority_queue< int, vector<int>, greater<int> > queue_two;
priority_queue< double, deque<double> >
      queue_three(aList.begin(), aList.end());
priority_queue< eventStruct, vector<eventStruct> >
      queue_four(eventComparison);
priority_queue< eventStruct, deque<eventStruct> >
      queue_five(aVector.begin(), aVector.end(), eventComparison);
```

The ***queue***s constructed out of ***vector***s tend to be somewhat smaller, while the ***queue***s constructed out of ***deque***s can be somewhat faster, particularly if the number of elements in the ***queue*** varies widely over the course of execution. However, these differences are slight, and either form generally works in most circumstances.

Because the ***priority_queue*** data structure does not itself know how to construct iterators, very few of the algorithms noted in Chapter 13 can be used with ***priority_queue***s. Instead of iterating over values, a typical algorithm that uses a ***priority_queue*** constructs a loop, which repeatedly pulls values from the structure (using the `top()` and `pop()` operations) until the collection becomes empty (tested using the `empty()` operation). The example program described in Section 11.3 will illustrate this use.

A ***priority_queue*** is implemented by internally building a data structure called a ***heap***. Abstractly, a ***heap***[4] is a binary tree in which the value associated with every node is smaller than or equal to the value associated with either child node.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example Program: Event-Driven Simulation

An extended example will now illustrate one of the more common uses of a priority_queues, which is to support the construction of a simulation model. A *discrete event-driven simulation* is a popular simulation technique. Objects in the simulation model objects in the real world, and are programmed to react as much as possible as the real objects would react. A ***priority_queue*** is used to store a representation of *events* that are waiting to happen. This queue is stored in order, based on the time the event should occur, so the smallest element will always be the next event to be modeled. As an event occurs, it can spawn other events. These subsequent events are placed into the queue as well. Execution continues until all events have been processed.

Events can be represented as subclasses of a base class, which we call ***event***. The base class simply records the time at which the event will take place. A pure virtual function named `processEvent` is invoked to execute the event:

```
class event {
public:
   event (unsigned int t) : time(t) { }
   const unsigned int time;
   virtual void processEvent() = 0;
};
```

The simulation queue needs to maintain a collection of different types of events, sometimes called a *heterogeneous* collection. Each different form of event is represented by a subclass of class ***event***, but not all ***event***s have the same exact type. For this reason the collection must store *pointers*[5] to ***event***s, instead of the events themselves.

NOTE: Since the containers maintain pointers to values, not the values themselves, the programmer is responsible for managing the memory for the objects being manipulated.

Since comparison of pointers cannot be specialized on the basis of the pointer types, we must instead define a new comparison function for pointers to events. In the Standard C++ Library we do this by defining a new structure whose sole purpose is to define the function invocation operator `()` in the appropriate fashion. Since in this particular example we want to use the ***priority_queue*** to return the smallest[6] element each time, rather than the largest, the order of the comparison is reversed, as follows:

```
struct eventComparison {
   bool operator () (event * left, event * right) const
      { return left->time > right->time; }
};
```

We are now ready to define the class ***simulation***, which provides the structure for the simulation activities. The class ***simulation*** provides two functions: the first is used to insert a new event into the queue, while the second runs the simulation. A data field is also provided to hold the current simulation ***time***:

```
class simulation {
public:
   simulation () : eventQueue(), time(0) { }

   void scheduleEvent (event * newEvent)
      { eventQueue.push (newEvent); }

   void run();

   unsigned int time;

protected:
   priority_queue<event *, vector<event *>, eventComparison> eventQueue;
};
```

Notice the declaration of the ***priority_queue*** used to hold the pending ***event***s. In this case we are using a ***vector*** as the underlying container, but we could just as easily have used a ***deque***.

The heart of the simulation is the member function `run()`, which defines the event loop. This procedure makes use of three of the five ***priority_queue*** operations, namely `top()`, `pop()`, and `empty()`. It is implemented as follows:

©Copyright 1999 Rogue Wave Software, Inc.

```
void simulation::run()
{
   while (! eventQueue.empty()) {
      event * nextEvent = eventQueue.top();
      eventQueue.pop();
      time = nextEvent->time;
      nextEvent->processEvent();
      delete nextEvent;   // free memory used by event
      }
}
```

Example Program: An Ice Cream Store Simulation

To illustrate the use of our simulation framework, this example program gives a simple simulation of an ice cream store. Such a simulation might be used, for example, to determine the optimal number of chairs that should be provided, based on assumptions such as the frequency with which customers arrive, the length of time they stay, and so on.

NOTE: The complete event simulation is in the file icecream.cpp.

Our store simulation is based around a subclass of class ***simulation***, defined as follows:

```
class storeSimulation : public simulation {
public:
   storeSimulation()
      : freeChairs(35), profit(0.0), simulation() { }

   bool canSeat (unsigned int numberOfPeople);
   void order(unsigned int numberOfScoops);
   void leave(unsigned int numberOfPeople);

private:
   unsigned int freeChairs;
   double profit;
} theSimulation;
```

There are three basic activities associated with the store: arrival, ordering and eating, and leaving. This is reflected not only in the three member functions defined in the ***simulation*** class, but in three separate subclasses of ***event***.

The member functions associated with the store simply record the activities taking place, producing a log that can later be studied to evaluate the simulation.

```
bool storeSimulation::canSeat (unsigned int numberOfPeople)
   // if sufficient room, then seat customers
{
   cout << "Time: " << time;
   cout << " group of " << numberOfPeople << " customers arrives";
   if (numberOfPeople < freeChairs) {
      cout << " is seated" << endl;
      freeChairs -= numberOfPeople;
      return true;
      }
   else {
      cout << " no room, they leave" << endl;
      return false;
      }
}

void storeSimulation::order (unsigned int numberOfScoops)
   // serve icecream, compute profits
{
   cout << "Time: " << time;
   cout << " serviced order for " << numberOfScoops << endl;
   profit += 0.35 * numberOfScoops;
}

void storeSimulation::leave (unsigned int numberOfPeople)
   // people leave, free up chairs
{
   cout << "Time: " << time;
   cout << " group of size " << numberOfPeople <<
         " leaves" << endl;
   freeChairs += numberOfPeople;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

As we noted already, each activity is matched by a subclass of ***event***. Each subclass of ***event*** includes an integer data field, which represents the size of a group of customers. The arrival event occurs when a group enters. When executed, the arrival event creates and installs a new instance of the order event. The function `randomInteger()` is used to compute a random integer between 1 and the argument value (see Section 2.2.5).

```
class arriveEvent : public event {
public:
   arriveEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};

void arriveEvent::processEvent()
{               // see if everybody can be seated
   if (theSimulation.canSeat(size))
      theSimulation.scheduleEvent
         (new orderEvent(time + 1 + randomInteger(4), size));
}
```

An order event similarly spawns a leave event:

```
class orderEvent : public event {
public:
   orderEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};

void orderEvent::processEvent()
{               // each person orders some number of scoops
   for (int i = 0; i < size; i++)
      theSimulation.order(1 + rand(3));
   theSimulation.scheduleEvent
      (new leaveEvent(time + 1 + randomInteger(10), size));
};
```

Finally, leave events free up chairs, but do not spawn any new events:

```
class leaveEvent : public event {
public:
   leaveEvent (unsigned int time, unsigned int groupSize)
      : event(time), size(groupSize) { }
   virtual void processEvent ();
private:
   unsigned int size;
};

void leaveEvent::processEvent ()
{               // leave and free up chairs
   theSimulation.leave(size);
}
```

To run the simulation we simply create some number of initial events (say, 30 minutes worth), then invoke the `run()` member function:

```
void main() {
   // load queue with some number of initial events
   unsigned int t = 0;
   while (t < 30) {
      t += rand(6);
      theSimulation.scheduleEvent(
         new arriveEvent(t, 1 + randomInteger(4)));
      }

   // then run simulation and print profits
   theSimulation.run();
   cout << "Total profits " << theSimulation.profit << endl;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 12: string

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The string Abstraction

A *string* is basically a sequence of characters that can be indexed. In fact, although a ***string*** is not declared as a subclass of ***vector***, almost all the ***vector*** operators discussed in Chapter 5 can be applied to ***string*** values. Indeed, a string qualifies as a ***sequence*** container type. However, a ***string*** is also a much more abstract quantity. In addition to simple ***vector*** operators, the ***string*** datatype provides a number of useful and powerful high level operations.

In the Standard C++ Library, a ***string*** is actually a template class, named ***basic_string***. The template argument represents the type of character that will be held by the ***string*** container. By defining strings in this fashion, the Standard C++ Library not only provides facilities for manipulating sequences of normal 8-bit ASCII characters, but also for manipulating other types of character-like sequences, such as 16-bit wide characters. The datatypes ***string*** and ***wstring*** (for wide string) are simply typedefs of ***basic_string***, defined as follows:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

NOTE: In the remainder of this chapter, we refer to the string datatype, but all the operations we introduce are equally applicable to wide strings.

As we have already noted, a ***string*** is similar in many ways to a ***vector*** of characters. Like the ***vector*** datatype, a ***string*** is associated with two sizes. The first represents the *number of characters* currently being stored in the string; the second is the capacity, the *maximum number of characters* that can potentially be stored in a ***string*** without reallocation of a new internal buffer.

As in the ***vector*** datatype, the capacity of a ***string*** is a dynamic quantity. When ***string*** operations cause the number of characters being stored in a ***string*** value to exceed the capacity of the ***string***, a new internal buffer is allocated and initialized with the ***string*** values, and the capacity of the ***string*** is increased. All this occurs behind the scenes, requiring no interaction with the programmer.

Include Files

Programs that use ***string***s must include the `string` header file:

```
# include <string>
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

string Operations

In the following sections, we'll examine the Standard C++ Library operations used to create and manipulate ***string***s.

Declaration and Initialization of string

The simplest form of declaration for a ***string*** simply names a new variable, or names a variable along with the initial value for the ***string***. This form was used extensively in the example graph program given in Section 9.3.2. A copy constructor also permits a ***string*** to be declared that takes its value from a previously defined ***string***:

```
string s1;
string s2 ("a string");
string s3 = "initial value";
string s4 (s3);
```

In these simple cases the capacity is initially exactly the same as the number of characters being stored. An alternative constructor lets you set the capacity and initialize the ***string*** with repeated copies of a single character value:

```
string s7 (10, '\n');          // holds ten newline characters
```

Finally, like all the container classes in the Standard C++ Library, a ***string*** can be initialized using a pair of ***iterator***s[7]. The sequence denoted by the ***iterator***s must have the appropriate type of elements.

```
string s8 (aList.begin(), aList.end());
```

Resetting Size and Capacity

As with the ***vector*** datatype, the current size of a ***string*** is yielded by the `size()` member function, while the current capacity is returned by `capacity()`. The latter can be changed by a call on the `reserve()` member function, which adjusts the capacity if necessary so that the string can hold at least as many elements as specified by the argument. The member function `max_size()` returns the maximum string size that can be allocated:

```
cout << s6.size() << endl;
cout << s6.capacity() << endl;
s6.reserve(200);                    // change capacity to 200
cout << s6.capacity() << endl;
cout << s6.max_size() << endl;
```

The member function `length()` is simply a synonym for `size()`. The member function `resize()` changes the size of a string, either truncating characters from the end or inserting new characters. The optional second argument for `resize()` can be used to specify the character inserted into the newly created character positions.

```
s7.resize(15, '\t');                // add tab characters at end
cout << s7.length() << endl;        // size should now be 15
```

The member function `empty()` returns `true` if the string contains no characters, and is generally faster than testing the length against a zero constant.

```
if (s7.empty())
   cout << "string is empty" << endl;
```

Assignment, Append, and Swap

A string variable can be assigned the value of either another string, a literal C-style character array, or an individual character:

```
s1 = s2;
s2 = "a new value";
s3 = 'x';
```

©Copyright 1999 Rogue Wave Software, Inc.

The operator += can also be used with any of these three forms of argument, and specifies that the value on the right-hand side should be *appended* to the end of the current string value.

```
s3 += "yz";                    // s3 is now xyz
```

The more general `assign()` and `append()` member functions let you specify a subset of the right-hand side to be assigned to or appended to the receiver. Two arguments, `pos` and `n`, indicate that the `n` values following position `pos` should be assigned or appended:

```
s4.assign (s2, 0, 3);         // assign first three characters
s4.append (s5, 2, 3);         // append characters 2, 3 and 4
```

The addition operator + is used to form the catenation of two strings. The operator + creates a copy of the left argument, then appends the right argument to this value:

```
cout << (s2 + s3) << endl;    // output catenation of s2 and s3
```

As with all the containers in the Standard C++ Library, the contents of two ***string***s can be exchanged using the `swap()` member function:

```
s5.swap (s4);                 // exchange s4 and s5
```

Character Access

An individual character from a string can be accessed or assigned using the subscript operator. The member function `at()` is almost a synonym for this operation, except that an `out_of_range` exception is thrown if the requested location is greater than or equal to `size()`.

```
cout << s4[2] << endl;        // output position 2 of s4
s4[2] = 'x';                  // change position 2
cout << s4.at(2) << endl;     // output updated value
```

The member function `c_str()` returns a pointer to a null-terminated character array, whose elements are the same as those contained in the ***string***. This lets you use ***string***s with functions that require a pointer to a conventional C-style character array. The resulting pointer is declared as constant, which means that you cannot use `c_str()` to modify the string. In addition, the value returned by `c_str()` might not be valid after any operation that may cause reallocation, such as `append()` or `insert()`. The member function `data()` returns a pointer to the underlying character buffer:

```
char d[256];
strcpy(d, s4.c_str());                // copy s4 into array d
```

Iterators

The member functions `begin()` and `end()` return beginning and ending random-access ***iterator***s for the string. The values denoted by the ***iterator***s are individual string elements. The functions `rbegin()` and `rend()` return backwards ***iterator***s.

Insertion, Removal, and Replacement

The ***string*** member functions `insert()` and `erase()` are similar to the ***vector*** functions `insert()` and `erase()`. Like the vector versions, they can take ***iterator***s as arguments, and specify the insertion or removal of the ranges specified by the arguments. The function `replace()` is a combination of erase and insert, in effect replacing the specified range with new values.

```
s2.insert(s2.begin()+2, aList.begin(), aList.end());
s2.erase(s2.begin()+3, s2.begin()+5);
s2.replace(s2.begin()+3, s2.begin()+6, s3.begin(), s3.end());
```

NOTE: Note that the contents of an iterator are not guaranteed to be valid after any operation that might force a reallocation of the internal string buffer, such as an append or an insertion.

The functions above also have non-iterator implementations. The `insert()` member function takes as argument a position and a ***string***, and inserts the ***string*** into the given position. The erase function takes two integer arguments, a position and a length, and removes the characters specified. The replace function takes two similar integer arguments, as well as a string and an optional length, and replaces the indicated range with the string or with an initial portion of a string, if the length has been explicitly specified.

©Copyright 1999 Rogue Wave Software, Inc.

```
s3.insert (3, "abc");      // insert abc after position 3
s3.erase (4, 2);           // remove positions 4 and 5
s3.replace (4, 2, "pqr");  // replace positions 4 and 5 with pqr
```

Copy and Substring

The member function `copy()` generates a substring and assigns this substring to the `char*` target given as the first argument. The range of values for the substring is specified either by an initial position, or a position and a length:

```
s3.copy (s4, 2);       // assign to s4 positions 2 to end of s3
s5.copy (s4, 2, 3);    // assign to s4 positions 2 to 4 of s5
```

The member function `substr()` returns a ***string*** that represents a portion of the current ***string***. The range is specified by either an initial position, or a position and a length:

```
cout << s4.substr(3) << endl;       // output 3 to end
cout << s4.substr(3, 2) << endl;    // output positions 3 and 4
```

string Comparisons

The member function `compare()` is used to perform a lexical comparison between the receiver and an argument string. Optional arguments permit the specification of a different starting position, or a starting position and length of the argument string. See Section 13.6.5 for a description of lexicographical ordering. The function returns a negative value if the receiver is lexically smaller than the argument, a zero value if they are equal, and a positive value if the receiver is larger than the argument.

The relational and equality operators, `<`, `<=`, `==`, `!=`, `>=`, and `>`, are all defined using the comparison member function. Comparisons can be made either between two strings, or between strings and ordinary C-style character literals.

Having explained the functionality of `compare()`, we should note that users seldom invoke this member function directly. Instead, comparisons of strings are usually performed using the conventional comparison operators, which in turn make use of the function `compare()`.

Searching Operations

The member function `find()` determines the first occurrence of the argument string in the current string. An optional integer argument lets you specify the starting position for the search. (Remember that string index positions begin at zero.) If the function can locate such a match, it returns the starting index of the match in the current string. Otherwise, it returns a value out of the range of the set of legal subscripts for the string. The function `rfind()` is similar, but scans the string from the end, moving backwards.

```
s1 = "mississippi";
cout << s1.find("ss") << endl;              // returns 2
cout << s1.find("ss", 3) << endl;           // returns 5
cout << s1.rfind("ss") << endl;             // returns 5
cout << s1.rfind("ss", 4) << endl;          // returns 2
```

The functions `find_first_of()`, `find_last_of()`, `find_first_not_of()`, and `find_last_not_of()` treat the argument string as a set of characters. As with many of the other functions, one or two optional integer arguments can be used to specify a subset of the current string. These functions find the first or last character that is either present or absent from the argument set. The position of the given character, if located, is returned. If no such character exists, a value out of the range of any legal subscript is returned.

```
i = s2.find_first_of ("aeiou");            // find first vowel
j = s2.find_first_not_of ("aeiou", i);     // next non-vowel
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example Function: Split a Line into Words

In this section we illustrate the use of some of the ***string*** functions by defining a function to split a line of text into individual words. We have already made use of this function in the concordance example program in Section 9.3.3.

NOTE: The split function is in the concordance program in file concord.cpp.

There are three arguments to this function. The first two are ***string***s, describing the line of text and the separators to be used to differentiate words, respectively. The third argument is a ***list*** of ***string***s, used to return the individual words in the line.

```
void split
   (string & text, string & separators, list<string> & words)
{
   int n = text.length();
   int start, stop;

   start = text.find_first_not_of(separators);
   while ((start >= 0) && (start < n)) {
      stop = text.find_first_of(separators, start);
      if ((stop < 0) || (stop > n)) stop = n;
      words.push_back(text.substr(start, stop - start));
      start = text.find_first_not_of(separators, stop+1);
      }
}
```

The program begins by finding the first character that is *not* a separator. The loop then looks for the next following character that *is* a separator, or uses the end of the string if no such value is found. The difference between these two is then a word, which is copied out of the text using a substring operation and inserted into the ***list*** of words. A search is then made to discover the start of the next word, and the loop continues. When the index value exceeds the limits of the ***string***, execution stops.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part IV: Algorithms

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 13: Generic Algorithms

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Overview

In this chapter and in Chapter 14 we examine and illustrate each of the generic algorithms provided by the Standard C++ Library. The names and a short description of each of the algorithms in this chapter are given in Table 19. We have divided the algorithms into several categories, based on how they are typically used. This division differs from the categories used in the Standard C++ Library definition, which is based upon whether or not the algorithms modify their arguments.

Table 19 -- Generic algorithms of the Standard C++ Library

Algorithm	Purpose
Algorithms initializing a sequence	
`fill()`	Fills a sequence with an initial value
`fill_n()`	Fills n positions with an initial value
`copy()`	Copies a sequence into another sequence
`copy_backward()`	Copies a sequence into another sequence
`generate()`	Initializes a sequence using a generator
`generate_n()`	Initializes n positions using a generator
`swap_ranges()`	Swaps values from two parallel sequences
Searching algorithms	
`find()`	Finds an element matching the argument
`find_if()`	Finds an element satisfying a condition
`adjacent_find()`	Finds consecutive duplicate elements
`find_first_of()`	Finds the first occurrence of one member of a sequence in another sequence
`find_end()`	Finds the last occurrence of a sub-sequence in a sequence
`search()`	Matches a sub-sequence within a sequence
`max_element()`	Finds the maximum value in a sequence
`min_element()`	Finds the minimum value in a sequence
`mismatch()`	Finds first mismatch in parallel sequences
Algorithms for in-place transformations	
`reverse()`	Reverses the elements in a sequence
`replace()`	Replaces specific values with new value
`replace_if()`	Replaces elements matching predicate
`rotate()`	Rotates elements in a sequence around a point
`partition()`	Partitions elements into two groups
`stable_partition()`	Partitions preserving original ordering
`next_permutation()`	Generates permutations in sequence
`prev_permutation()`	Generates permutations in reverse sequence
`inplace_merge()`	Merges two adjacent sequences into one
`random_shuffle()`	Randomly rearranges elements in a sequence
Removal algorithms	
`remove()`	Removes elements that match condition
`unique()`	Removes all but first of duplicate values in sequences
Scalar-producing algorithms	
`count()`	Counts number of elements matching value
`count_if()`	Counts elements matching predicate
`accumulate()`	Reduces sequence to a scalar value
`inner_product()`	Gives inner product of two parallel sequences
`equal()`	Checks two sequences for equality

©Copyright 1999 Rogue Wave Software, Inc.

`lexicographical_compare()`	Compares two sequences

Sequence-generating algorithms

`transform()`	Transforms each element
`partial_sum()`	Generates sequence of partial sums
`adjacent_difference()`	Generates sequence of adjacent differences

Miscellaneous operations

`for_each()`	Applies a function to each element of a collection

In this chapter, we illustrate the use of each algorithm with a series of short examples. Many of the algorithms are also used in the sample programs provided in the chapters on the various container classes. These cross references have been noted where appropriate.

All the short example programs described in this section are contained in a number of files, named `alg1.cpp` through `alg6.cpp`. In the files, the example programs are augmented with output statements describing the test programs and illustrating the results of executing the algorithms. So as not confuse the reader with unnecessary detail, we have generally omitted these output statements from the descriptions here. If you wish to see the text programs complete with output statements, you can compile and execute the test files. The expected output from these programs is also included in the distribution.

Include Files

To use any of the generic algorithms, you must first include the appropriate header file. The majority of the functions are defined in the header file `algorithm`. The functions `accumulate()`, `inner_product()`, `partial_sum()`, and `adjacent_difference()` are defined in the header file `numeric`.

```
# include <algorithm>
# include <numeric>
```

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Initialization Algorithms

The first set of algorithms we cover are used chiefly, although not exclusively, to initialize a newly created sequence with certain values. The Standard C++ Library provides several initialization algorithms. The initialization algorithms all overwrite every element in a container. The difference between the algorithms is the source for the values used in initialization. The `fill()` algorithm repeats a single value, the `copy()` algorithm reads values from a second container, and the `generate()` algorithm invokes a function for each new value. In our discussion we provide examples of how to apply these algorithms, and suggest how to choose one algorithm over another.

NOTE: The sample programs described in the following sections are in the file alg1.cpp. We generally omit output statements from the descriptions of the programs provided here, although they are included in the executable versions.

Fill a Sequence with An Initial Value

The `fill()` and `fill_n()` algorithms are used to initialize or reinitialize a sequence with a fixed value. Their declarations are as follows:

```
void fill (ForwardIterator first, ForwardIterator last, const T&);
void fill_n (OutputIterator, Size, const T&);
```

The example program illustrates several uses of the algorithm:

```
void fill_example ()
   // illustrates the use of the fill algorithm
   // see alg1.cpp for complete source code
{

      // example 1, fill an array with initial values
   char buffer[100], * bufferp = buffer;
   fill (bufferp, bufferp + 100, '\0');
   fill_n (bufferp, 10, 'x');

      // example 2, use fill to initialize a list
   list<string> aList(5, "nothing");
   fill_n (inserter(aList, aList.begin()), 10, "empty");

      // example 3, use fill to overwrite values in list
   fill (aList.begin(), aList.end(), "full");

      // example 4, fill in a portion of a collection
   vector<int> iVec(10);
   generate (iVec.begin(), iVec.end(), iotaGen(1));
   vector<int>::iterator & seven =
         find(iVec.begin(), iVec.end(), 7);
   fill (iVec.begin(), seven, 0);
}
```

In example 1, an array of character values is declared. The `fill()` algorithm is invoked to initialize each location in this array with a null character value. The first 10 positions are then replaced with the character `'x'` by using the algorithm `fill_n()`. Note that the `fill()` algorithm requires both starting and past-end ***iterator***s as arguments, whereas the `fill_n()` algorithm uses a starting ***iterator*** and a count.

By applying an *insert iterator* (Section 2.4), example 2 illustrates how the `fill_n()` algorithm can be used to initialize a variable length container, such as a ***list***. In this case the ***list*** initially contains five elements, all holding the text `"nothing"`. The call on `fill_n()` then inserts ten instances of the string `"empty"`. The resulting ***list*** contains fifteen elements.

Examples 3 and 4 illustrate how `fill()` can be used to change the values in an existing container. In example 3 each of the fifteen elements in the ***list*** created in example 2 is replaced by the string `"full"`.

Example 4 overwrites only a portion of a ***list***. Using the algorithm `generate()` and the function object ***iotaGen*** (see Section 3.3 and Section 13.2.3), a vector is initialized to the values `1 2 3 ... 10`. The `find()` algorithm (see Section 13.3.1) is used to locate the position of the element 7, saving the location in an iterator appropriate for the ***vector***

©Copyright 1999 Rogue Wave Software, Inc.

datatype. The `fill()` call then replaces all values up to, but not including, the `7` entry with the value `0`. The resulting vector has six zero fields, followed by the values `7`, `8`, `9` and `10`.

The `fill()` and `fill_n()` algorithm can be used with all the container classes contained in the Standard C++ Library, although insert iterators must be used with ordered containers, such as ***set***s.

Copy One Sequence Into Another Sequence

The algorithms `copy()` and `copy_backward()` are versatile functions that can be used for a number of different purposes, and are probably the most commonly executed algorithms in the Standard C++ Library. The declarations for these algorithms are as follows:

```
OutputIterator copy (InputIterator first, InputIterator last,
         OutputIterator result);

BidirectionalIterator copy_backward
   (BidirectionalIterator first, BidirectionalIterator last,
    BidirectionalIterator result);
```

The result returned by the `copy()` function is a pointer to the end of the copied sequence. However, the result of one `copy()` operation can be used as a starting iterator in a subsequent `copy()` to make a *catenation* of values.

Uses of the copy algorithm include:

- Duplicating an entire sequence by copying into a new sequence
- Creating sub-sequences of an existing sequence
- Adding elements into a sequence
- Copying a sequence from input or to output
- Converting a sequence from one form into another

The uses of the copy algorithm are illustrated in the following sample program:

```
void copy_example()

   // illustrates the use of the copy algorithm
   // see alg1.cpp for complete source code

{
   char * source = "reprise";
   char * surpass = "surpass";
   char buffer[120], * bufferp = buffer;

      // example 1, a simple copy
   copy (source, source + strlen(source) + 1, bufferp);

      // example 2, self copies
   copy (bufferp + 2, bufferp + strlen(buffer) + 1, bufferp);
   int buflen = strlen(buffer) + 1;
   copy_backward (bufferp, bufferp + buflen, bufferp + buflen + 3);
   copy (surpass, surpass + 3, bufferp);

      // example 3, copy to output
   copy (bufferp, bufferp + strlen(buffer),
            ostream_iterator<char,char>(cout));
   cout << endl;

      // example 4, use copy to convert type
   list<char> char_list;
   copy (bufferp, bufferp + strlen(buffer),
            inserter(char_list, char_list.end()));
   char * big = "big ";
   copy (big, big + 4, inserter(char_list, char_list.begin()));

   char buffer2 [120], * buffer2p = buffer2;
   * copy (char_list.begin(), char_list.end(), buffer2p) = '\0';
   cout << buffer2 << endl;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

In example 1, the first call on `copy()` simply copies the string pointed to by the variable `source` into a buffer, resulting in the buffer containing the text `"reprise"`. Note that the ending position for the copy is one past the terminating null character, thus ensuring the null character is included in the copy operation.

The `copy()` operation is specifically designed to permit *self-copies*, which are copies of a sequence onto itself, as long as the destination ***iterator*** does not fall within the range formed by the source ***iterator***s. This is illustrated by example 2. Here the copy begins at position 2 of the buffer and extends to the end, copying characters into the beginning of the buffer. This results in the buffer holding the value `"prise"`.

The second half of example 2 illustrates the use of the `copy_backward()` algorithm. This function performs the same task as the `copy()` algorithm, but moves elements from the end of the sequence first, progressing to the front of the sequence. If you think of the argument as a string, characters are moved starting from the right and progressing to the left. In this case the result is that `buffer` is assigned the value `"priprise"`. The first three characters are then modified by another `copy()` operation to the values `"sur"`, resulting in `buffer` holding the `value "surprise"`.

NOTE: In the copy_backwards algorithm, note that it is the order of transfer that is backwards, not the elements themselves; the relative placement of moved values in the target is the same as in the source.

Example 3 illustrates `copy()` being used to move values to an output stream (see Section 2.3.2). The target in this case is an `ostream_iterator` generated for the output stream `cout`. A similar mechanism can be used for input values. For example, a simple mechanism to copy every word in the input stream into a ***list*** is the following call on `copy()`:

```
list<string> words;
istream_iterator<string, char> in_stream(cin), eof;

copy(in_stream, eof, inserter(words, words.begin()));
```

This technique is used in the spell checking program described in Section 8.3.

Copy can also be used to convert from one type of stream to another. For example, the call in example 4 of the sample program copies the characters held in the buffer one by one into a ***list*** of characters. The call on `inserter()` creates an insert iterator, used to insert values into the ***list***. The first call on `copy()` places the string `surprise`, created in example 2, into the ***list***. The second call on `copy()` inserts the values from the string `"big"` onto the front of the ***list***, resulting in the ***list*** containing the characters `big surprise`. The final call on `copy()` illustrates the reverse process, copying characters from a ***list*** back into a character buffer.

Initialize a Sequence with Generated Values

A *generator* is a function that returns a series of values on successive invocations. Probably the generator you are most familiar with is a random number generator. However, generators can be constructed for a variety of different purposes, including initializing sequences.

Like `fill()` and `fill_n()`, the algorithms `generate()` and `generate_n()` are used to initialize or reinitialize a sequence. However, instead of a fixed argument, these algorithms draw their values from a generator. The declarations of these algorithms are as follows:

```
void generate (ForwardIterator, ForwardIterator, Generator);
void generate_n (OutputIterator, Size, Generator);
```

Our example program shows several uses of the generate algorithm to initialize a sequence:

```
string generateLabel () {
   // generate a unique label string of the form L_ddd
   // see alg1.cpp for complete source code
   static int lastLabel = 0;
   char labelBuffer[80];
   ostrstream ost(labelBuffer, 80);
   ost << "L_" << lastLabel++ << '\0';
   return string(labelBuffer);
}

void generate_example ()
   // illustrate the use of the generate and generate_n algorithms
{

      // example 1, generate a list of label values
   list<string> labelList;
   generate_n (inserter(labelList, labelList.begin()),
         4, generateLabel);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
      // example 2, generate an arithmetic progression
   vector<int> iVec(10);
   generate (iVec.begin(), iVec.end(), iotaGen(2));
   generate_n (iVec.begin(), 5, iotaGen(7));
   }
```

A generator can be constructed as a simple function that *remembers* information about its previous history in one or more static variables. An example is shown in the beginning of the example program, where the function `generateLabel()` is described. This function creates a sequence of unique string labels, such as might be needed by a compiler. Each invocation on the function `generateLabel()` results in a new string of the form `L_ddd`, each with a unique digit value. Because the variable named `lastLabel` is declared as `static`, its value is remembered from one invocation to the next. The first example of the sample program illustrates how this function might be used in combination with the `generate_n()` algorithm to initialize a list of four label values.

As we described in Chapter 3, a function is any object that will respond to the function call operator. Using this definition, classes can easily be constructed as functions. The class ***iotaGen***, is an example (see Section 3.3). The ***iotaGen*** function object creates a generator for an integer arithmetic sequence. In the second example in the sample program, this sequence is used to initialize a ***vector*** with the integer values 2 through 11. A call on `generate_n()` is then used to overwrite the first 5 positions of the ***vector*** with the values 7 through 11, resulting in the ***vector*** `7 8 9 10 11 7 8 9 10 11`.

Swap Values from Two Parallel Ranges

The template function `swap()` can be used to exchange the values of two objects of the same type. It has the following definition:

```
template <class T> void swap (T& a, T& b)
{
   T temp(a);
   a = b;
   b = temp;
}
```

The function is generalized to ***iterator***s in the function named `iter_swap()`. The algorithm `swap_ranges()` extends this to entire sequences. The values denoted by the first sequence are exchanged with the values denoted by a second, parallel sequence. The description of the `swap_ranges()` algorithm is as follows:

```
ForwardIterator swap_ranges
   (ForwardIterator first, ForwardIterator last,
         ForwardIterator first2);
```

The second range is described only by a starting ***iterator***. It is assumed but not verified that the second range has at least as many elements as the first range.

NOTE: A number of algorithms operate on two parallel sequences. In most cases, the second sequence is identified using only a starting iterator, not a starting and ending iterator pair. It is assumed, but never verified, that the second sequence is at least as large as the first. Errors will occur if this condition is not satisfied.

In the example program, both `swap()` and `iter_swap()` are used separately and in combination:

```
void swap_example ()

       // illustrates the use of the algorithm swap_ranges
       // see alg1.cpp for complete source code
{
      // first make two parallel sequences
   int data[] = {12, 27, 14, 64}, *datap = data;
   vector<int> aVec(4);
   generate(aVec.begin(), aVec.end(), iotaGen(1));

      // illustrate swap and iter_swap
   swap(data[0], data[2]);
   vector<int>::iterator last = aVec.end(); last--;
   iter_swap(aVec.begin(), last);

      // now swap the entire sequence
   swap_ranges (aVec.begin(), aVec.end(), datap);
}
```


©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Searching Operations

The next category of algorithms we describe are used to locate elements within a sequence that satisfy certain properties. Most commonly the result of a search is then used as an argument to a further operation, such as a `copy` (Section 13.2.2), a `partition` (Section 13.4.4), or an `in-place merge` (Section 13.4.6.)

NOTE: The example functions described in the following sections are in the file alg2.cpp.

The searching routines described in this section return an iterator that identifies the first element that satisfies the search condition. It is common to store this value in an ***iterator*** variable, as follows:

```
list<int>::iterator where;
where = find(aList.begin(), aList.end(), 7);
```

If you want to locate *all* the elements that satisfy the search conditions, you must write a loop. In that loop, the value yielded by a previous search is first advanced, since otherwise the value yielded by the previous search would once again be returned. The resulting value is then used as a starting point for the new search. For example, the following loop from the `adjacent_find()` example program (Section 13.3.2) will print the value of all repeated characters in a string argument:

```
while ((where = adjacent_find(where, stop)) != stop) {
   cout << "double " << *where << " in position "
      << where - start << endl;
   ++where;
   }
```

NOTE: The searching algorithms in the Standard C++ Library all return the end-of-sequence iterator if no value is found that matches the search condition. As it is generally illegal to dereference the end-of-sequence value, it is important to check for this condition before proceeding to use the result of a search.

Many of the searching algorithms have an optional argument that can specify a function for comparing elements in place of the equality operator `==` for the container element type. In the descriptions of the algorithms, we write these optional arguments inside a square bracket to indicate they need not be specified if the standard equality operator is acceptable.

Find an Element Satisfying a Condition

There are two algorithms, `find()` and `find_if()`, that are used to find the first element that satisfies a condition. The declarations of these two algorithms are as follows:

```
InputIterator find_if (InputIterator first, InputIterator last,
      Predicate);

InputIterator find (InputIterator first, InputIterator last,
      const T&);
```

The algorithm `find_if()` takes as argument a predicate function, which can be any function that returns a boolean value (see Section 3.2). The `find_if()` algorithm returns a new ***iterator*** that designates the first element in the sequence that satisfies the predicate. The second argument, the past-the-end iterator, is returned if no element is found that matches the requirement. Because the resulting value is an ***iterator***, the dereference operator `*` must be used to obtain the matching value. This is illustrated in the example program.

The second form of the algorithm, `find()`, replaces the predicate function with a specific value, and returns the first element in the sequence that tests equal to this value, using the appropriate equality operator `==` for the given datatype.

NOTE: The algorithms find() and find_if() perform a linear sequential search through the associated structures. The set and map data structures, which are ordered, provide their own find() member functions, which are more efficient. Because of this, the generic find() algorithm should not be used with set and map.

The following example program illustrates the use of `find()` and `find_if()`:

```
void find_test ()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // illustrates the use of the find algorithm
   // see alg2.cpp for complete source code
{
   int vintageYears[] = {1967, 1972, 1974, 1980, 1995};
   int * start = vintageYears;
   int * stop = start + 5;
   int * where = find_if (start, stop, isLeapYear);

   if (where != stop)
      cout << "first vintage leap year is " << *where << endl;
   else
      cout << "no vintage leap years" << endl;

   where = find(start, stop, 1995);

   if (where != stop)
      cout << "1995 is position " << where - start
         << " in sequence" << endl;
   else
      cout "1995 does not occur in sequence" << endl;
```

Find Consecutive Duplicate Elements

The `adjacent_find()` algorithm is used to discover the first element in a sequence equal to the next immediately following element. For example, if a sequence contained the values `1 4 2 5 6 6 7 5`, the algorithm would return an ***iterator*** corresponding to the first `6` value. If no value satisfying the condition is found, then the end-of-sequence iterator is returned. The declaration of the algorithm is as follows:

```
ForwardIterator adjacent_find (ForwardIterator first,
   ForwardIterator last [, BinaryPredicate ] );
```

The first two arguments specify the sequence to be examined. The optional third argument must be a *binary predicate*, a binary function returning a boolean value. If present, the binary function is used to test adjacent elements, otherwise the equality operator `==` is used.

The example program searches a text string for adjacent letters. In the example text these are found in positions `5`, `7`, `9`, `21`, and `37`. The increment is necessary inside the loop in order to avoid the same position being discovered repeatedly.

```
void adjacent_find_example ()

   // illustrates the use of the adjacent_find instruction
   // see alg2.cpp for complete source code
{
   char * text = "The bookkeeper carefully opened the door.";

   char * start = text;
   char * stop = text + strlen(text);
   char * where = start;

   cout << "In the text: " << text << endl;
   while ((where = adjacent_find(where, stop)) != stop) {
      cout << "double " << *where
         << " in position " << where - start << endl;
      ++where;
   }
}
```

Find the First Occurrence of Any Value from a Sequence

The algorithm `find_first_of()` is used to find the first occurrence of some element from one sequence that is also contained in another sequence:

```
ForwardIterator1 find_first_of
        (ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2
         [, BinaryPredicate pred ] );
```

The algorithm returns a new ***iterator*** that designates the first element found in `[first1,last1)` that is also contained in `[first2,last2)`. If no match is found, the second argument is returned. Because the resulting value is an ***iterator***, the dereference operator `*` must be used to obtain the matching value. This is illustrated in the example program:

```
void find_test ()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // illustrates the use of the find algorithm
   // see alg2.cpp for complete source code
{
   int vintageYears[] = {1967, 1972, 1974, 1980, 1995};
   int requestedYears[] = [1923, 1970, 1980, 1974 };
   int * start = vintageYears;
   int * stop = start + 5;
   int * where = find_first_of (start, stop,
                           requestedyears,requestedyears+4 );

   if (where != stop)
      cout << "first requested vintage year is " << *where << endl;
   else
      cout << "no requested vintage years" << endl;

}

// The output would indicate 1974.
```

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending ***iterator*** pair for both sequences, not just the first sequence.

Like the algorithms `equal()` and `mismatch()`, an alternative version of `find_first_of()` takes an optional binary predicate that is used to compare elements from the two sequences.

The ***basic_string*** class provides its own versions of the `find_first_of` and `find_end` algorithms, including several convenience overloads of the basic pattern indicated here.

Find a Sub-Sequence within a Sequence

The algorithms `search()` and `search_n()` are used to locate the beginning of a particular sub-sequence within a larger sequence. The easiest example to understand is the problem of looking for a particular substring within a larger string, although the algorithm can be generalized to other uses. The arguments are assumed to have at least the capabilities of forward iterators.

```
ForwardIterator search
   (ForwardIterator first1, ForwardIterator last1,
   ForwardIterator first2, ForwardIterator last2
   [, BinaryPredicate ]);
```

Suppose, for example, that we want to discover the location of the string `"ration"` in the string `"dreams and aspirations"`. The solution to this problem is shown in the example program. If no appropriate match is found, the value returned is the past-the-end iterator for the first sequence.

```
void search_example ()

      // illustrate the use of the search algorithm
      // see alg2.cpp for complete source code
{
   char * base = "dreams and aspirations";
   char * text = "ration";

   char * where = search(base, base + strlen(base),
         text, text + strlen(text));

   if (*where != '\0')
      cout << "substring position: " << where - base << endl;
   else
      cout << "substring does not occur in text" << endl;
}
```

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending ***iterator*** pair for both sequences, not just the first sequence.

Like the algorithms `equal()` and `mismatch()`, an alternative version of `search()` takes an optional binary predicate that is used to compare elements from the two sequences.

What determines the speed of the search? In the worst case, the number of comparisons performed by the algorithm `search()` is the product of the number of elements in the two sequences. Except in rare cases, however, this worst case behavior is highly unlikely.

©Copyright 1999 Rogue Wave Software, Inc.

Find the Last Occurrence of a Sub-Sequence

The algorithm `find_end()` is used to locate the beginning of the last occurrence of a particular sub-sequence within a larger sequence. The easiest example to understand is the problem of looking for a particular substring within a larger string, although the algorithm can be generalized to other uses. The arguments are assumed to have at least the capabilities of forward iterators.

```
ForwardIterator find_end
   (ForwardIterator first1, ForwardIterator last1,
   ForwardIterator first2, ForwardIterator last2
   [, BinaryPredicate ]);
```

Suppose, for example, that we want to discover the location of the last occurrence of the string `"le"` in the string `"The road less traveled"`. The solution to this problem is shown in the example program. If no appropriate match is found, the value returned is the past-the-end iterator for the first sequence.

```
void find_end_example ()

      // illustrate the use of the find_end algorithm
      // see alg2.cpp for complete source code
{
   char * base = "The road less traveled";
   char * text = "le";

   char * where = find(base, base + strlen(base),
         text, text + strlen(text));

   if (*where != '\0')
      cout << "substring position: " << where - base << endl;
   else
      cout << "substring does not occur in text" << endl;
}
```

Note that this algorithm, unlike many that manipulate two sequences, uses a starting and ending ***iterator*** pair for both sequences, not just the first sequence.

Like the algorithms `find_first_of()` and `search()`, an alternative version of `find_end()` takes an optional binary predicate that is used to compare elements from the two sequences.

What determines the speed of the search? As with `search()`, in the worst case, the number of comparisons performed by the algorithm `find_end()` is the product of the number of elements in the two sequences.

Locate Maximum or Minimum Element

The functions `max()` and `min()` can be used to find the maximum and minimum of a pair of values. These can optionally take a third argument that defines the comparison function to use in place of the less-than operator `<`. The arguments are values, not ***iterators***:

```
template <class T>
   const T& max(const T& a, const T& b [, Compare ] );
template <class T>
   const T& min(const T& a, const T& b [, Compare ] );
```

The maximum and minimum functions are generalized to entire sequences by the generic algorithms `max_element()` and `min_element()`. For these functions the arguments are input iterators.

```
ForwardIterator max_element (ForwardIterator first,
      ForwardIterator last [, Compare ] );
ForwardIterator min_element (ForwardIterator first,
      ForwardIterator last [, Compare ] );
```

These algorithms return an ***iterator*** that denotes the largest or smallest of the values in a sequence, respectively. Should more than one value satisfy the requirement, the result yielded is the first satisfactory value. Both algorithms can optionally take a third argument, which is the function to be used as the comparison operator in place of the default operator.

The example program illustrates several uses of these algorithms. The function named `split()` that was used to divide a string into words in the ***string*** example is described in Section 12.3. The function `randomInteger()` is described in Section 2.5.

©Copyright 1999 Rogue Wave Software, Inc.

```
void max_min_example ()

   // illustrates use of max_element and min_element algorithms
   // see alg2.cpp for complete source code
{
   // make a vector of random numbers between 0 and 99
   vector<int> numbers(25);
   for (int i = 0; i < 25; i++)
      numbers[i] = randomInteger(100);

   // print the maximum
   vector<int>::iterator max =
      max_element(numbers.begin(), numbers.end());
   cout << "largest value was " << * max << endl;

   // example using strings
   string text =
      "It was the best of times, it was the worst of times.";
   list<string> words;
   split (text, " .,!:;", words);
   cout << "The smallest word is "
         << * min_element(words.begin(), words.end())
         << " and the largest word is "
         << * max_element(words.begin(), words.end())
         << endl;
}
```

The maximum and minimum algorithms can be used with all the datatypes provided by the Standard C++ Library. However, for the ordered datatypes ***set*** and ***map***, the maximum or minimum values are more easily accessed as the first or last elements in the structure.

Locate the First Mismatched Elements in Parallel Sequences

The name `mismatch()` might lead you to think this algorithm is the inverse of the `equal()` algorithm, which determines if two sequences are equal (Section 13.6.4). Instead, the `mismatch()` algorithm returns a ***pair*** of iterators that together indicate the first positions where two parallel sequences have differing elements. The structure ***pair*** is described in Section 9.1 and Section 9.2.1.

The second sequence is denoted only by a starting position, without an ending position. It is assumed, but not checked, that the second sequence contains at least as many elements as the first. The arguments and return type for `mismatch()` can be described as follows:

```
pair<InputIterator, InputIterator> mismatch
   (InputIterator first1, InputIterator last1,
      InputIterator first2 [, BinaryPredicate ] );
```

The elements of the two sequences are examined in parallel, element by element. When a mismatch is found, that is, a point where the two sequences differ, then a ***pair*** containing ***iterator***s denoting the locations of the two differing elements is constructed and returned. If the first sequence is exhausted before discovering any mismatched elements, then the resulting ***pair*** contains the ending value for the first sequence, and the last value examined in the second sequence. The second sequence need not yet be exhausted.

The example program illustrates the use of this procedure. The function `mismatch_test()` takes as arguments two ***string*** values. These are lexicographically compared and a message printed indicating their relative ordering. This is similar to the analysis performed by the `lexicographic_compare()` algorithm, although that function simply returns a boolean value.

Because the `mismatch()` algorithm assumes the second sequence is at least as long as the first, a comparison of the two string lengths is performed first, and the arguments are reversed if the second string is shorter than the first. After the call on `mismatch()`, the elements of the resulting ***pair*** are separated into their component parts. These parts are then tested to determine the appropriate ordering.

```
void mismatch_test (char * a, char * b)

       // illustrates the use of the mismatch algorithm
       // see alg2.cpp for complete source code
{
   pair<char *, char *> differPositions(0, 0);
   char * aDiffPosition;
   char * bDiffPosition;

   if (strlen(a) < strlen(b)) {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
      // make sure longer string is second
      differPositions = mismatch(a, a + strlen(a), b);
      aDiffPosition = differPositions.first;
      bDiffPosition = differPositions.second;
      }
   else {
      differPositions = mismatch(b, b + strlen(b), a);
      // note following reverse ordering
      aDiffPosition = differPositions.second;
      bDiffPosition = differPositions.first;
      }

   // compare resulting values
   cout << "string " << a;
   if (*aDiffPosition == *bDiffPosition)
      cout << " is equal to ";
   else if (*aDiffPosition < *bDiffPosition)
      cout << " is less than ";
   else
      cout << " is greater than ";
   cout << b << endl;
}
```

A second form of the `mismatch()` algorithm is similar to this one, except it accepts a binary predicate as a fourth argument. This binary function is used to compare elements in place of the == operator.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

In-Place Transformations

The next category of algorithms that we examine are used to modify and transform sequences without moving them from their original storage locations.

NOTE: The example functions described in the following sections can be found in the file alg3.cpp.

A few of these routines, such as `replace()`, include a *copy* version as well as the original in-place transformation algorithms. For the others, should it be necessary to preserve the original, a copy of the sequence should be created before the transformations are applied. For example, the following code illustrates how one can place the reversal of one ***vector*** into another newly allocated ***vector***:

```
vector<int> newVec(aVec.size());
copy (aVec.begin(), aVec.end(), newVec.begin()); // first copy
reverse (newVec.begin(), newVec.end());          // then reverse
```

Many of the algorithms described as sequence generating operations, such as `transform()` or `partial_sum()`, can also modify a value in place by simply using the same iterator as both input and output specification (see Section 13.7.1 and Section 13.7.2).

Reverse Elements in a Sequence

The algorithm `reverse()` reverses the elements in a sequence so that the last element becomes the new first, and the first element the new last. The arguments are assumed to be bidirectional iterators, and no value is returned.

```
void reverse (BidirectionalIterator first,
   BidirectionalIterator last);
```

The example program illustrates two uses of this algorithm. In example 1, an array of character values is reversed. The algorithm `reverse()` can also be used with ***list*** values, as shown in example 2. In this example, a ***list*** is initialized with the values `2` to `11` in increasing order. (This is accomplished using the ***iotaGen*** function object introduced in Section 3.3.2.3). The ***list*** is then reversed, which results in the list holding the values `11` to `2` in decreasing order. Note, however, that the ***list*** data structure also provides its own `reverse()` member function.

```
void reverse_example ()

   // illustrates the use of the reverse algorithm
   // see alg3.cpp for complete source code
{
   // example 1, reversing a string
char * text = "Rats live on no evil star";
reverse (text, text + strlen(text));
cout << text << endl;


   // example 2, reversing a list
list<int> iList;
generate_n (inserter(iList, iList.begin()), 10, iotaGen(2));
reverse (iList.begin(), iList.end());
}
```

Replace Certain Elements With Fixed Value

The algorithms `replace()` and `replace_if()` are used to replace occurrences of certain elements with a new value. For both algorithms, the new value is the same, no matter how many replacements are performed. Using the algorithm `replace()`, all occurrences of a particular test value are replaced with the new value. In the case of `replace_if()`, all elements that satisfy a predicate function are replaced by a new value. The ***iterator*** arguments must be forward iterators.

The algorithms `replace_copy()` and `replace_copy_if()` are similar to `replace()` and `replace_if()`, however, they leave the original sequence intact and place the revised values into a new sequence, which may be a different type.

©Copyright 1999 Rogue Wave Software, Inc.

```
void replace (ForwardIterator first, ForwardIterator last,
         const T&, const T&);

void replace_if (ForwardIterator first, ForwardIterator last,
         Predicate, const T&);

OutputIterator replace_copy (InputIterator, InputIterator,
         OutputIterator, const T&, const T&);

OutputIterator replace_copy (InputIterator, InputIterator,
         OutputIterator, Predicate, const T&);
```

In the example program, a vector is initially assigned the values `0 1 2 3 4 5 4 3 2 1 0`. A call on `replace()` replaces the value `3` with the value `7`, resulting in the vector `0 1 2 7 4 5 4 7 2 1 0`. The invocation of `replace_if()` replaces all even numbers with the value `9`, resulting in the vector `9 1 9 7 9 5 9 7 9 1 9`.

```
void replace_example ()

       // illustrates the use of the replace algorithm
       // see alg3.cpp for complete source code
{
      // make vector 0 1 2 3 4 5 4 3 2 1 0
   vector<int> numbers(11);
   for (int i = 0; i < 11; i++)
      numbers[i] = i < 5 ? i : 10 - i;

      // replace 3 by 7
   replace (numbers.begin(), numbers.end(), 3, 7);

      // replace even numbers by 9
   replace_if (numbers.begin(), numbers.end(), isEven, 9);

      // illustrate copy versions of replace
   int aList[] = {2, 1, 4, 3, 2, 5};
   int bList[6], cList[6], j;
   replace_copy (aList, aList+6, &bList[0], 2, 7);
   replace_copy_if (bList, bList+6, &cList[0],
         bind2nd(greater<int>(), 3), 8);
}
```

The example program also illustrates the use of the `replace_copy` algorithms. First, an array containing the values `2 1 4 3 2 5` is created. This is modified by replacing the `2` values with `7`, resulting in the array `7 1 4 3 7 5`. Next, all values larger than `3` are replaced with the value `8`, resulting in the array values `8 1 8 3 8 8`. In the latter case the `bind2nd()` adaptor is used to modify the binary greater-than function by binding the second argument to the constant value `3`, thereby creating the unary function `x > 3`.

Rotate Elements Around a Midpoint

A *rotation* of a sequence divides the sequence into two sections and swaps the order of the sections, while maintaining the order of elements within each section. Suppose, for example, that we have the values `1` to `10` in sequence:

```
1 2 3 4 5 6 7 8 9 10
```

If we were to rotate around the element `7`, the values `7` to `10` would be moved to the beginning, while the elements `1` to `6` would be moved to the end. This would result in the following sequence:

```
7 8 9 10 1 2 3 4 5 6
```

When you invoke the algorithm `rotate()`, the starting point, midpoint, and past-the-end location are all denoted by forward iterators:

```
void rotate (ForwardIterator first, ForwardIterator middle,
   ForwardIterator last);
```

The *prefix* portion, the set of elements following the start and not including the midpoint, is swapped with the *suffix*, the set of elements between the midpoint and the past-the-end location. Note, as in the previous example, that these two segments need not be the same length.

```
void rotate_example()

       // illustrates the use of the rotate algorithm
       // see alg3.cpp for complete source code
```

©Copyright 1999 Rogue Wave Software, Inc.

```
{
       // create the list 1 2 3 ... 10
   list<int> iList;
   generate_n(inserter(iList, iList.begin()), 10, iotaGen(1));

      // find the location of the seven
   list<int>::iterator & middle =
         find(iList.begin(), iList.end(), 7);

      // now rotate around that location
   rotate (iList.begin(), middle, iList.end());

      // rotate again around the same location
   list<int> cList;
   rotate_copy (iList.begin(), middle, iList.end(),
      inserter(cList, cList.begin()));
}
```

The example program first creates a list of the integers in order from `1` to `10`. Next, the `find()` algorithm (Section 13.3.1) is used to find the location of the element `7`. This is used as the midpoint for the rotation.

A second form of `rotate()` copies the elements into a new sequence, rather than rotating the values in place. This is also shown in the example program, which once again rotates around the middle position that now contains a `3`. The resulting list is `3 4 5 6 7 8 9 10 1 2`. The values held in `iList` remain unchanged.

a Sequence into Two Groups

A *partition* is formed by moving all the elements that satisfy a predicate to one end of a sequence, and all the elements that fail to satisfy the predicate to the other end. Partitioning elements is a fundamental step in certain sorting algorithms, such as `quicksort`.

```
BidirectionalIterator partition
   (BidirectionalIterator, BidirectionalIterator, Predicate);

BidirectionalIterator stable_partition
   (BidirectionalIterator, BidirectionalIterator, Predicate);
```

There are two forms of partition supported in the Standard C++ Library. The first, provided by the algorithm `partition()`, guarantees only that the elements are divided into two groups. The result value is an ***iterator*** that describes the final midpoint between the two groups; it is one past the end of the first group.

In the example program, the initial ***vector*** contains the values `1` to `10` in order. The partition moves the even elements to the front, and the odd elements to the end. This results in the ***vector*** holding the values `10 2 8 4 6 5 7 3 9 1`, and the midpoint ***iterator*** pointing to the element `5`.

```
void partition_example ()

       // illustrates the use of the partition algorithm
       // see alg3.cpp for complete source code
{
       // first make the vector 1 2 3 ... 10
   vector<int> numbers(10);
   generate(numbers.begin(), numbers.end(), iotaGen(1));

      // now put the even values low, odd high
   vector<int>::iterator result =
      partition(numbers.begin(), numbers.end(), isEven);
   cout << "middle location " << result - numbers.begin() << endl;

      // now do a stable partition
   generate (numbers.begin(), numbers.end(), iotaGen(1));
   stable_partition (numbers.begin(), numbers.end(), isEven);
}
```

The relative order of the elements within a partition in the resulting ***vector*** may not be the same as the values in the original ***vector***. For example, the value `4` preceded the element `8` in the original, but may follow the element `8` in the result. A second version of partition, named `stable_partition()`, guarantees the ordering of the resulting values. For the input shown in the example, the stable partition would result in the sequence `2 4 6 8 10 1 3 5 7 9`. The `stable_partition()` algorithm is slightly slower and uses more memory than the `partition()` algorithm, so when the order of elements is not important you should use `partition()`.

©Copyright 1999 Rogue Wave Software, Inc.

While there is a unique `stable_ partition()` for any sequence, the `partition()` algorithm can return any number of values. For example, the following are all legal partitions of the sample problem:

```
2 4 6 8 10 1 3 5 7 9

10 8 6 4 2 5 7 9 3 1

2 6 4 8 10 3 5 7 9 1

6 4 2 10 8 5 3 7 9 1
```

Generate Permutations in Sequence

A *permutation* is a rearrangement of values. If values such as integers, characters, or words can be compared against each other, then it is possible to systematically construct all permutations of a sequence. For example, there are two permutations of two values, six permutations of three values, and 24 permutations of four values.

The permutation generating algorithms have the following definition:

```
bool next_permutation (BidirectionalIterator first,
      BidirectionalIterator last, [ Compare ] );

bool prev_permutation (BidirectionalIterator first,
      BidirectionalIterator last, [ Compare ] );
```

The second example in the sample program illustrates the same idea, using pointers to character arrays instead of integers. In this case, a different comparison function must be supplied, since the default operator would simply compare pointer addresses.

```
bool nameCompare (char * a, char * b) { return strcmp(a, b) <= 0; }

void permutation_example ()

   // illustrates the use of the next_permutation algorithm
   // see alg3.cpp for complete source code
{

      // example 1, permute the values 1 2 3
   int start [] = { 1, 2, 3};
   do
      copy (start, start + 3,
            ostream_iterator<int,char> (cout, " ")), cout << endl;
   while (next_permutation(start, start + 3));

      // example 2, permute words
   char * words = {"Alpha", "Beta", "Gamma"};
   do
      copy (words, words + 3,
         ostream_iterator<char *,char> (cout, " ")), cout << endl;
   while (next_permutation(words, words + 3, nameCompare));

      // example 3, permute characters backwards
   char * word = "bela";
   do
      cout << word << ' ';
   while (prev_permutation (word, &word[4]));
   cout << endl;
}
```

Example 3 in the sample program illustrates the use of the reverse permutation algorithm, which generates values in reverse sequence. This example also begins in the middle of a sequence, rather than at the beginning. The remaining permutations of the word *bela* are: `beal`, `bale`, `bael`, `aleb`, `albe`, `aelb`, `aebl`, `able`, and `abel`.

Permutations can also be ordered so that the smallest permutation lists values smallest to largest, and the largest sequence lists values largest to smallest. For example, consider the permutations of the integers `1 2 3`. Taken in order, the six permutations of these values are:

```
123

132
```

©Copyright 1999 Rogue Wave Software, Inc.

```
      213

      231

      312

      321
```

Notice that in the first permutation the values are all ascending, while in the last permutation they are all descending.

Merge Two Adjacent Sequences into One

A *merge* takes two ordered sequences and combines them into a single ordered sequence, interleaving elements from each collection as necessary to generate the new list. The `inplace_merge()` algorithm assumes a sequence is divided into two adjacent sections, each of which is ordered. The merge combines the two sections into one, moving elements as necessary. The alternative `merge()` algorithm (Section 6.2.3.1 and Section 14.5) can be used to merge two separate sequences into one.

The arguments to `inplace_merge()` must be bidirectional iterators.

```
void inplace_merge (BidirectionalIterator first,
   BidirectionalIterator middle,
   BidirectionalIterator last [, BinaryFunction ] );
```

The example program illustrates the use of the `inplace_merge()` algorithm with a ***vector*** and with a ***list***. The sequence `0 2 4 6 8 1 3 5 7 9` is placed into a ***vector***. A `find()` call (Section 13.3.1) is used to locate the beginning of the odd number sequence. The two calls on `inplace_merge()` then combine the two sequences into one.

```
void inplace_merge_example ()

      // illustrate the use of the inplace_merge algorithm
      // see alg3.cpp for complete source code
{
      // first generate the sequence 0 2 4 6 8 1 3 5 7 9
   vector<int> numbers(10);
   for (int i = 0; i < 10; i++)
      numbers[i] = i < 5 ? 2 * i : 2 * (i - 5) + 1;

      // then find the middle location
   vector<int>::iterator midvec =
      find(numbers.begin(), numbers.end(), 1);

      // copy them into a list
   list<int> numList;
   copy (numbers.begin(), numbers.end(),
         inserter (numList, numList.begin()));
   list<int>::iterator midList =
         find(numList.begin(), numList.end, 1);

      // now merge the lists into one
   inplace_merge (numbers.begin(), midvec, numbers.end());
   inplace_merge (numList.begin(), midList, numList.end());
}
```

Randomly Rearrange Elements in a Sequence

The algorithm `random_shuffle()` randomly rearranges the elements in a sequence. Exactly `n` swaps are performed, where `n` represents the number of elements in the sequence. Of course, the results are unpredictable. Because the arguments must be random access ***iterator***s, this algorithm can only be used with ***vector***s, ***deque***s, or ordinary pointers; it cannot be used with ***list***s, ***set***s, or ***map***s.

```
void random_shuffle (RandomAccessIterator first,
   RandomAccessIterator last [, Generator ] );
```

An alternative version of the algorithm uses the optional third argument. This value must be a random number generator. This generator must take as an argument a positive value `m` and return a value between `0` and `m-1`. As with the `generate()` algorithm, this random number function can be any type of object that responds to the function invocation operator.

```
void random_shuffle_example ()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // illustrates the use of the random_shuffle algorithm
   // see alg3.cpp fr complete source code
{
   // first make the vector containing 1 2 3 ... 10
   vector<int> numbers;
   generate(numbers.begin(), numbers.end(), iotaGen(1));

   // then randomly shuffle the elements
   random_shuffle (numbers.begin(), numbers.end());

   // do it again, with explicit random number generator
   struct RandomInteger {
   {
      operator() (int m) { return rand() % m; }
   } random;

   random_shuffle (numbers.begin(), numbers.end(), random);
}
```

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Removal Algorithms

The `remove()` algorithm and the `unique()` algorithm can be somewhat confusing the first time you encounter them. Both claim to remove certain values from a sequence, yet neither reduces the size of the sequence. Both operate by moving the values that are to be *retained* to the front of the sequence, and returning an ***iterator*** that describes where this sequence ends. Elements after this ***iterator*** are simply the original sequence values, left unchanged. This is necessary because the generic algorithm has no knowledge of the container it is working on. It only has a generic iterator. This is part of the price we pay for generic algorithms. In most cases you will want to use this ***iterator*** result as an argument to the `erase()` member function for the container, removing the values from the ***iterator*** to the end of the sequence.

Let us illustrate this with a simple example. Suppose we want to remove the even numbers from the sequence `1 2 3 4 5 6 7 8 9 10`, something we could do with the `remove_if()` algorithm. Applying this algorithm would leave us with the following sequence:

```
1 3 5 7 9 | 6 7 8 9 10
```

The vertical bar here represents the position of the iterator returned by the `remove_if()` algorithm. Notice that the five elements before the bar represent the result we want, while the five values after the bar are simply the original contents of those locations. Using this ***iterator*** value along with the end-of-sequence iterator as arguments to `erase()`, we can eliminate the unwanted values, and obtain the desired result.

Both the algorithms described here have an alternative *copy* version. The copy version of the algorithms leaves the original unchanged, and places the preserved elements into an output sequence.

NOTE: The example functions described in this section can be found in the file alg4.cpp.

Remove Unwanted Elements

The algorithm `remove()` eliminates unwanted values from a sequence. As with the `find()` algorithm, these can either be values that match a specific constant, or values that satisfy a given predicate. The declaration of the argument types is as follows:

```
ForwardIterator remove
   (ForwardIterator first, ForwardIterator last, const T &);
ForwardIterator remove_if
   (ForwardIterator first, ForwardIterator last, Predicate);
```

The algorithm `remove()` copies values to the front of the sequence, overwriting the location of the removed elements. All elements not removed remain in their relative order. Once all values have been examined, the remainder of the sequence is left unchanged. The iterator returned as the result of the operation provides the end of the new sequence. For example, eliminating the element `2` from the sequence `1 2 4 3 2` results in the sequence `1 4 3 3 2`, with the ***iterator*** returned as the result pointing at the second `3`. This value can be used as argument to `erase()` in order to eliminate the remaining elements `3` and `2`, as illustrated in the example program.

A copy version of the algorithms copies values to an output sequence, rather than making transformations in place.

```
OutputIterator remove_copy
         (InputIterator first, InputIterator last,
         OutputIterator result, const T &);

OutputIterator remove_copy_if
         (InputIterator first, InputIterator last,
         OutputIterator result, Predicate);
```

The use of `remove()` is shown in the following program.

```
void remove_example ()

   // illustrates the use of the remove algorithm
   // see alg4.cpp for complete source code
{
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   // create a list of numbers
   int data[] = {1, 2, 4, 3, 1, 4, 2};
   list<int> aList;
   copy (data, data+7, inserter(aList, aList.begin()));

      // remove 2's, copy into new list
   list<int> newList;
   remove_copy (aList.begin(), aList.end(),
      back_inserter(newList), 2);

      // remove 2's in place
   list<int>::iterator where;
   where = remove (aList.begin(), aList.end(), 2);
   aList.erase(where, aList.end());

      // remove all even values
   where = remove_if (aList.begin(), aList.end(), isEven);
   aList.erase(where, aList.end());
}
```

Remove Runs of Similar Values

The algorithm `unique()` moves through a linear sequence, eliminating all but the first element from every consecutive group of equal elements. The argument sequence is described by forward iterators:

```
ForwardIterator unique (ForwardIterator first,
   ForwardIterator last [, BinaryPredicate ] );
```

As the algorithm moves through the collection, elements are moved to the front of the sequence, overwriting the existing elements. Once all unique values are identified, the remainder of the sequence is left unchanged. For example, a sequence such as `1 3 3 2 2 2 4` is changed into `1 3 2 4 | 2 2 4`. We use a vertical bar to indicate the location returned by the iterator result value. This location marks the end of the unique sequence, and the beginning of the left-over elements. With most containers, the value returned by the algorithm can be used as an argument in a subsequent call on `erase()` to remove the undesired elements from the collection. This is illustrated in the example program.

A copy version of the algorithm moves the unique values to an output iterator, rather than making modifications in place. In transforming a ***list*** or ***multiset***, an insert ***iterator*** can be used to change the copy operations of the output ***iterator*** into insertions.

```
OutputIterator unique_copy
      (InputIterator first, InputIterator last,
       OutputIterator result [, BinaryPredicate ] );
```

These are illustrated in the sample program:

```
void unique_example ()

   // illustrates the use of the unique algorithm
   // see alg4.cpp for complete source code
{
      // first make a list of values
   int data[] = {1, 3, 3, 2, 2, 4};
   list<int> aList;
   set<int> aSet;
   copy (data, data+6, inserter(aList, aList.begin()));

      // copy unique elements into a set
   unique_copy (aList.begin(), aList.end(),
      inserter(aSet, aSet.begin()));

      // copy unique elements in place
   list<int>::iterator where;
   where = unique(aList.begin(), aList.end());

      // remove trailing values
   aList.erase(where, aList.end());
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Scalar-Producing Algorithms

The next category of algorithms reduces an entire sequence to a single scalar value. Remember that two of these algorithms, `accumulate()` and `inner_product()`, are declared in the `numeric` header file, not the `algorithm` header file as are the other generic algorithms.

NOTE: The example functions described in the following sections are in the file alg5.cpp.

Count the Number of Elements That Satisfy a Condition

The algorithms `count()` and `count_if()` are used to discover the number of elements that match a given value or that satisfy a given predicate, respectively. Each algorithm comes in two forms. The newer form returns the number of matches found, while the older one takes as argument a reference to a counting value, typically an integer, and increments this value. In the latter case the `count()` function itself yields no value.

The newer form of these functions is currently mandated by the standard. The older form is retained for two reasons: first, for backward compatibility, since older versions of the standard contained it; and second, because the newer form requires that a compiler support partial specialization, which at this writing is still rare.

```
iterator_traits<InputIterator>::distance_type
count (InputIterator first, InputIterator last, const T& value);

iterator_traits<InputIterator>::distance_type
count_if (InputIterator first, InputIterator last, Predicate pred);

void count (InputIterator first, InputIterator last,
            const T&, Size &);
void count_if (InputIterator first, InputIterator last,
         Predicate, Size &);
```

The example code fragment illustrates the use of the older form of these algorithms. The call on `count()` counts the number of occurrences of the letter `e` in a sample string, while the invocation of `count_if()` counts the number of vowels.

```
void count_example ()

      // illustrates the use of the count algorithm
      // see alg5.cpp for complete source code
{
   int eCount = 0;
   int vowelCount = 0;

   char * text = "Now is the time to begin";

   count (text, text + strlen(text), 'e', eCount);
   count_if (text, text + strlen(text), isVowel, vowelCount);

   cout << "There are " << eCount << " letter e's " << endl
        << "and " << vowelCount << " vowels in the text:"
        << text << endl;
}
```

Note that if your compiler does not support partial specialization, you don't have the form of the `count()` algorithms that return the sum as a function result, but only the form that adds to the last argument in its parameter list, which is passed by reference. This means successive calls on these functions can be used to produce a cumulative sum. This also means that you must initialize the variable passed to this last argument location prior to calling one of these algorithms.

Reduce Sequence to a Single Value

The result generated by the `accumulate()` algorithm is the value produced by placing a binary operator between each element of a sequence, and evaluating the result. By default the operator is the addition operator, `+`, but this can be replaced by any binary function. An initial value or *identity* must be provided. This value is returned for empty sequences, and is otherwise used as the left argument for the first calculation.

©Copyright 1999 Rogue Wave Software, Inc.

```
ContainerType accumulate (InputIterator first, InputIterator last,
         ContainerType initial [, BinaryFunction ] );
```

The example program illustrates the use of `accumulate()` to produce the sum and product of a ***vector*** of `integer` values. In the first case the identity is zero, and the default operator `+` is used. In the second invocation, the identity is `1`, and the multiplication operator named *times* is explicitly passed as the fourth argument:

```
void accumulate_example ()

// illustrates the use of the accumulate algorithm
// see alg5.cpp for complete source code
{
   int numbers[] = {1, 2, 3, 4, 5};

// first example, simple accumulation
   int sum = accumulate (numbers, numbers + 5, 0);
   int product =
         accumulate (numbers, numbers + 5, 1, times<int>());

   cout << "The sum of the first five integers is " << sum << endl;
   cout << "The product is " << product << endl;

// second example, with different types for initial value
   list<int> nums;
   nums = accumulate (numbers, numbers+5, nums, intReplicate);
}

list<int>& intReplicate (list<int>& nums, int n)
      // add sequence n to 1 to end of list
{
   while (n) nums.push_back(n--);
   return nums;
}
```

Neither the identity value nor the result of the binary function is required to match the container type. This is illustrated in the example program by the invocation of `accumulate()` shown in the second example above. Here the identity is an empty list. The function, shown after the example program, takes as argument a ***list*** and an `integer` value, and repeatedly inserts values into the ***list***. The values inserted represent a decreasing sequence from the argument down to `1`. For the example input, the same ***vector*** as in the first example, the resulting ***list*** contains the `15` values `1 2 1 3 2 1 4 3 2 1 5 4 3 2 1`.

Generalized Inner Product

Assume we have two sequences of `n` elements each: `a1, a2, ... an` and `b1, b2, ... bn`. The *inner product* of the sequences is the sum of the parallel products, that is, the value `a1 * b1 + a2 * b2 + ... + an * bn`. Inner products occur in a number of scientific calculations. For example, the inner product of a row times a column is the heart of the traditional matrix multiplication algorithm. A generalized inner product uses the same structure, but permits the addition and multiplication operators to be replaced by arbitrary binary functions. The Standard C++ Library includes the following algorithm for computing an inner product:

```
ContainerType inner_product
   (InputIterator first1, InputIterator last1,
   InputIterator first2, ContainerType initialValue
      [ , BinaryFunction add, BinaryFunction times ] );
```

The first three arguments to the `inner_product()` algorithm define the two input sequences. The second sequence is specified only by the beginning iterator, and is assumed to contain at least as many elements as the first sequence. The next argument is an initial value, or *identity*, used for the summation operator. This is similar to the identity used in the `accumulate()` algorithm. In the generalized inner product function, the last two arguments are the binary functions that are used in place of the addition operator and the multiplication operator, respectively.

In the example program, the second invocation illustrates the use of alternative functions as arguments. The multiplication is replaced by an equality test, while the addition is replaced by a logical *or*. The result is true if any of the pairs are equal, and false otherwise. Using an *and* in place of the *or* would have resulted in a test which was true only if *all* pairs were equal; in effect, the same as the `equal()` algorithm described in the next section.

```
void inner_product_example ()

      // illustrates the use of the inner_product algorithm
      // see alg5.cpp for complete source code
{
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   int a[] = {4, 3, -2};
   int b[] = {7, 3, 2};

      // example 1, a simple inner product
   int in1 = inner_product(a, a+3, b, 0);
   cout << "Inner product is " << in1 << endl;

         // example 2, user defined operations
   bool anyequal = inner_product(a, a+3, b, true,
         logical_or<bool>(), equal_to<int>());
   cout << "any equal? " << anyequal << endl;
}
```

Test Two Sequences for Pairwise Equality

The `equal()` algorithm tests two sequences for pairwise equality. By using an alternative binary predicate, it can also be used for a wide variety of other pair-wise tests of parallel sequences, such as a pairwise test that returns a boolean result. The `mismatch()` algorithm gives the location of elements that fail that test (Section 13.3.7).

The arguments of the `equal()` algorithm are simple input ***iterator***s:

```
bool equal (InputIterator first, InputIterator last,
         InputIterator first2 [, BinaryPredicate] );
```

The `equal()` algorithm assumes, but does not verify, that the second sequence contains at least as many elements as the first. A `true` result is generated if all values test equal to their corresponding element. The alternative version of the algorithm substitutes an arbitrary boolean function for the equality test, and returns `true` if all pair-wise elements satisfy the predicate. In the sample program, this is illustrated by replacing the predicate with the `greater_equal()` function; in this fashion, `true` is returned only if all values in the first sequence are greater than or equal to their corresponding value in the second sequence.

```
void equal_example ()

   // illustrates the use of the equal algorithm
   // see alg5.cpp for complete source code
{
   int a[] = {4, 5, 3};
   int b[] = {4, 3, 3};
   int c[] = {4, 5, 3};

   cout << "a = b is: " << equal(a, a+3, b) << endl;
   cout << "a = c is: " << equal(a, a+3, c) << endl;
   cout << "a pair-wise greater-equal b is: "
      << equal(a, a+3, b, greater_equal<int>()) << endl;
}
```

Lexical Comparison

A *lexical comparison* is commonly used to determine the dictionary order of words. In this procedure, the elements or *characters* of two sequences are compared in pair-wise fashion. As long as characters within a pair match, the algorithm advances to the next pair. When characters within a pair fail to match, the earlier character determines the smaller word. For example, `everybody` is smaller than `everything`, since the `b` in the former word alphabetically precedes the `t` in the latter. Should one or the other sequence terminate before the other, the terminated sequence is considered the smaller. For example, `every` precedes both `everybody` and `everything`, but comes after `eve`. Finally, if both sequences terminate at the same time and pair-wise characters match in all cases, the two words are considered equal.

The `lexicographical_compare()` algorithm implements the concept of lexical comparison, returning `true` if the first sequence is smaller than the second, and `false` otherwise. The algorithm is generalized to any sequence. Thus, the `lexicographical_compare()` algorithm can be used with arrays, ***string***s, ***vector***s, ***list***s, or any other data structures of the Standard C++ Library.

```
bool lexicographical_compare
   (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2 [, BinaryFunction ] );
```

Unlike most other algorithms that take two sequences as argument, the `lexicographical_compare()` algorithm uses a first and a past-end ***iterator*** for *both* sequences. A variation on the algorithm also takes a fifth argument, which is the binary function used to compare corresponding elements from the two sequences.

©Copyright 1999 Rogue Wave Software, Inc.

The example program illustrates the use of the `lexicographical_compare()` algorithm with character sequences, and with arrays of integer values.

```
void lexicographical_compare_example()

   // illustrates the use of the lexicographical_compare algorithm
   // see alg5.cpp for complete source code
{
   char * wordOne = "everything";
   char * wordTwo = "everybody";

   cout << "compare everybody to everything " <<
      lexicographical_compare(wordTwo, wordTwo + strlen(wordTwo),
         wordOne, wordOne + strlen(wordOne)) << endl;

   int a[] = {3, 4, 5, 2};
   int b[] = {3, 4, 5};
   int c[] = {3, 5};

   cout << "compare a to b:" <<
      lexicographical_compare(a, a+4, b, b+3) << endl;
   cout << "compare a to c:" <<
      lexicographical_compare(a, a+4, c, c+2) << endl;
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Sequence-Generating Algorithms

All of the algorithms described in this section are used to generate a new sequence from an existing sequence by performing some type of transformation. In most cases, the output sequence is described by an output ***iterator***. This means these algorithms can be used to overwrite an existing structure, such as a ***vector***. Alternatively, by using an insert ***iterator*** (see Section 2.4), the algorithms can insert the new elements into a variable length structure, such as a ***set*** or ***list***. Finally, in some cases that we will discuss, the output iterator can be the same as one of the sequences specified by an input ***iterator***, thereby providing the ability to make an in-place transformation.

The functions `partial_sum()` and `adjacent_difference()` are declared in the header file `numeric`, while the other functions are described in the header file `algorithm`.

NOTE: The example functions described in the following sections can be found in the file alg6.cpp.

Transform One or Two Sequences

The algorithm `transform()` is used either to make a general transformation of a single sequence, or to produce a new sequence by applying a binary function in a pair-wise fashion to corresponding elements from two different sequences. The general definition of the argument and result types are as follows:

```
OutputIterator transform (InputIterator first, InputIterator last,
   OutputIterator result, UnaryFunction);

OutputIterator transform
   (InputIterator first1, InputIterator last1,
   InputIterator first2,  OutputIterator result, BinaryFunction);
```

The first form applies a unary function to each element of a sequence. In the example program given below, this is used to produce a ***vector*** of `integer` values that hold the arithmetic negation of the values in a linked ***list***. The input and output ***iterator***s can be the same, in which case the transformation is applied in-place, as shown in the example program.

The second form takes two sequences and applies the binary function in a pair-wise fashion to corresponding elements. The transaction assumes, but does not verify, that the second sequence has at least as many elements as the first sequence. Once more, the result can either be a third sequence, or one of the two input sequences.

```
int square(int n) { return n * n; }

void transform_example ()

// illustrates the use of the transform algorithm
// see alg6.cpp for complete source code
{
// generate a list of value 1 to 6
   list<int> aList;
   generate_n (inserter(aList, aList.begin()), 6, iotaGen(1));

// transform elements by squaring, copy into vector
   vector<int> aVec(6);
   transform (aList.begin(), aList.end(), aVec.begin(), square);

// transform vector again, in place, yielding 4th powers
   transform (aVec.begin(), aVec.end(), aVec.begin(), square);

// transform in parallel, yielding cubes
   vector<int> cubes(6);
   transform (aVec.begin(), aVec.end(), aList.begin(),
      cubes.begin(), divides<int>());
}
```

Sums

A *partial sum* of a sequence is a new sequence in which every element is formed by adding the values of all prior elements. For example, the partial sum of the vector `1 3 2 4 5` is the new vector `1 4 6 10 15`. The element `4` is formed

©Copyright 1999 Rogue Wave Software, Inc.

from the sum `1 + 3`, the element `6` from the sum `1 + 3 + 2`, and so on. Although the term *sum* is used in describing the operation, the binary function can be any arbitrary function. The example program illustrates this by computing partial products.

The arguments to the partial sum function are described as follows:

```
OutputIterator partial_sum
      (InputIterator first, InputIterator last,
       OutputIterator result [, BinaryFunction] );
```

By using the same value for both the input iterator and the result, the partial sum can be changed into an in-place transformation.

```
void partial_sum_example ()

// illustrates the use of the partial sum algorithm
//see alg6.cpp for complete source code
{
// generate values 1 to 5
   vector<int> aVec(5);
   generate (aVec.begin(), aVec.end(), iotaGen(1));

// output partial sums
   partial_sum (aVec.begin(), aVec.end(),
      ostream_iterator<int> (cout, " ")), cout << endl;

// output partial products
   partial_sum (aVec.begin(), aVec.end(),
      ostream_iterator<int> (cout, " "),
      times<int>() );
}
```

Adjacent Differences

An *adjacent difference* of a sequence is a new sequence formed by replacing every element with the difference between the element and the immediately preceding element. The first value in the new sequence remains unchanged. For example, a sequence such as `(1, 3, 2, 4, 5)` is transformed into `(1, 3-1, 2-3, 4-2, 5-4)`, and in this manner becomes the sequence `(1, 2, -1, 2, 1)`.

As with the algorithm `partial_sum()`, the term *difference* is not necessarily accurate, as an arbitrary binary function can be employed. The adjacent sums for this sequence are `(1, 4, 5, 6, 9)`, for example. The adjacent difference algorithm has the following declaration:

```
OutputIterator adjacent_difference (InputIterator first,
   InputIterator last, OutputIterator result [, BinaryFunction ]);
```

By using the same ***iterator*** as both input and output ***iterator***, the adjacent difference operation can be performed in place.

```
void adjacent_difference_example ()

// illustrates the use of the adjacent difference algorithm
//see alg6.cpp for complete source code
{
// generate values 1 to 5
   vector<int> aVec(5);
   generate (aVec.begin(), aVec.end(), iotaGen(1));

// output adjacent differences
   adjacent_difference (aVec.begin(), aVec.end(),
      ostream_iterator<int,char> (cout, " ")), cout << endl;

// output adjacent sums
   adjacent_difference (aVec.begin(), aVec.end(),
      ostream_iterator<int,char> (cout, " "),
      plus<int>() );
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The for_each Algorithm

The algorithm `for_each` applies a function to all elements in a collection. This algorithm takes three arguments: the first two provide the ***iterator***s that describe the sequence to be evaluated, and the third is a one-argument function. The algorithm `for_each()` applies the function to each value of the sequence, passing the value as an argument:

```
Function for_each
   (InputIterator first, InputIterator last, Function);
```

For example, the following code fragment, which uses the `print_if_leap()` function, prints a list of the leap years that occur between 1900 and 1997:

```
cout << "leap years between 1990 and 1997 are: ";
for_each (1990, 1997, print_if_leap);
cout << endl;
```

The argument function is guaranteed to be invoked only once for each element in the sequence. The `for_each()` algorithm itself returns the value of the third argument, although this is usually ignored.

NOTE: The function passed as the third argument is not permitted to make any modifications to the sequence, so it can only achieve a result by means of a side effect, such as printing, assigning a value to a global or static variable, or invoking another function that produces a side effect. If the argument function returns any result, it is ignored.

The following example searches an array of `integer` values representing dates, to determine which vintage wine years were also leap years:

```
int vintageYears[] = {1947, 1955, 1960, 1967, 1994};
...

cout << "vintage years which were also leap years are: ";
for_each (vintageYears, vintageYears + 5, print_if_leap);
cout << endl;
```

Side effects need not be restricted to printing. Assume we have a function `countCaps()` that counts the occurrence of capital letters:

```
int capCount = 0;

void countCaps(char c)  { if (isupper(c)) capCount++; }
```

The following example counts the number of capital letters in a string value:

```
string advice = "Never Trust Anybody Over 30!";
for_each(advice.begin(), advice.end(),countCaps);
cout << "upper-case letter count is " << capCount << endl;
```

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 14: Ordered Collection Algorithms

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Overview

In this section we describe the generic algorithms in the Standard C++ Library that are specific to ordered collections. These algorithms are summarized in Table 20:

Table 20 -- Generic algorithms specific to ordered collections

Algorithm	Purpose
Sorting algorithms	
`sort()`	Rearranges sequence, places in order
`stable_sort()`	Sorts, retaining original order of equal elements
`partial_sort()`	Sorts only part of sequence
`partial_sort_copy()`	Partial sorts into copy
Nth largest element algorithm	
`nth_element()`	Locates nth largest element
Binary search algorithms	
`binary_search()`	Searches, returning boolean
`lower_bound()`	Searches, returning first position
`upper_bound()`	Searches, returning last position
`equal_range()`	Searches, returning both positions
Merge ordered sequences algorithm	
`merge()`	Combines two ordered sequences
Set operations algoithms	
`set_union()`	Forms union of two sets
`set_intersection()`	Forms intersection of two sets
`set_difference()`	Forms difference of two sets
`set_symmetric_difference()`	Forms symmetric difference of two sets
`includes()`	Sees if one set is a subset of another
Heap operations algorithms	
`make_heap()`	Turns a sequence into a heap
`push_heap()`	Adds a new value to the heap
`pop_heap()`	Removes largest value from the heap
`sort_heap()`	Turns heap into sorted collection

Ordered collections can be created using the Standard C++ Library in a variety of ways. For example:

- The containers ***set***, ***multiset***, ***map***, and ***multimap*** are ordered collections by definition.
- A ***list*** can be ordered by invoking the `sort()` member function.
- A ***vector***, ***deque***, or ordinary C++ array can be ordered by using one of the sorting algorithms described in Section 14.2.

Like the generic algorithms described in Section 13, the algorithms described here are not specific to any particular container class. This means that they can be used with a wide variety of types. However, many of them do require the use of random-access ***iterator***s. For this reason they are most easily used with ***vector***s, ***deque***s, or ordinary arrays.

Almost all the algorithms described in this section have two versions. The first version uses the `less than` operator `<` for comparisons appropriate to the container element type. The second, and more general, version uses an explicit comparison function object, which we will write as `Compare`. This function object must be a binary predicate (see Section 3.2). Since this argument is optional, we will write it within square brackets in the description of the argument types.

©Copyright 1999 Rogue Wave Software, Inc.

A sequence is considered *ordered* if for every valid or *denotable* iterator `i` with a denotable successor `j`, the comparison `Compare(*j, *i)` is `false`. Note that this does not necessarily imply that `Compare(*i, *j)` is `true`. It is assumed that the relation imposed by `Compare` is transitive, and induces a total ordering on the values.

In the descriptions that follow, two values `x` and `y` are said to be equivalent if both `Compare(x, y)` and `Compare(y, x)` are `false`. Note that this need not imply that `x == y`.

Include Files

As with the algorithms described in Chapter 13, before you can use any of these algorithms in a program you must include the `algorithm` header file:

```
# include <algorithm>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Sorting Algorithms

NOTE: The example programs described in the following sections have been combined and are included in the file alg7.cpp. As in Chapter13, we generally omit output statements from the descriptions of the programs provided here, although they are included in the executable versions.

The Standard C++ Library provides two fundamental sorting algorithms, described as follows:

```
void sort (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ] );

void stable_sort (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ] );
```

The `sort()` algorithm is slightly faster, but it does not guarantee that equal elements in the original sequence retain their relative orderings in the final result. If order is important, the `stable_sort()` version should be used.

Because these algorithms require random access ***iterator***s, they can be used only with ***vector***s, ***deque***s, and ordinary C pointers. Note, however, that the ***list*** container provides its own `sort()` member function.

The comparison operator can be explicitly provided when the default operator `<` is not appropriate. This is used in the example program to sort a ***list*** into descending, rather than ascending order. An alternative technique for sorting an entire collection in the inverse direction is to describe the sequence using reverse ***iterator***s.

NOTE: Another sorting algorithm is provided by the heap operations described in Section 14.7.

The following example program illustrates the `sort()` algorithm being applied to a ***vector***, and the `sort()` algorithm with an explicit comparison operator being used with a ***deque***.

```
void sort_example ()

   // illustrates the use of the sort algorithm
   // see alg7.cpp for complete source code
{
      // fill both a vector and a deque
      // with random integers
   vector<int> aVec(15);
   deque<int> aDec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   generate (aDec.begin(), aDec.end(), randomValue);

      // sort the vector ascending
   sort (aVec.begin(), aVec.end());

      // sort the deque descending
   sort (aDec.begin(), aDec.end(), greater<int>() );

      // alternative way to sort descending
   sort (aVec.rbegin(), aVec.rend());
}
```

Sort

The generic algorithm `partial_sort()` sorts only a portion of a sequence. In the first version of the algorithm, three iterators are used to describe the beginning, middle, and end of a sequence. If `n` represents the number of elements between the start and middle, then the smallest `n` elements are moved into this range in order. The remaining elements are moved into the second region. The order of the elements in this second region is undefined.

```
void partial_sort (RandomAccessIterator first,
   RandomAccessIterator middle,
   RandomAccessIterator last [ , Compare ]);
```

©Copyright 1999 Rogue Wave Software, Inc.

A second version of the algorithm leaves the input unchanged. The output area is described by a pair of random access *iterator*s. If `n` represents the size of this area, the smallest `n` elements in the input are moved into the output in order. If `n` is larger than the input, the entire input is sorted and placed in the first `n` locations in the output. In either case, the end of the output sequence is returned as the result of the operation.

```
RandomAccessIterator partial_sort_copy
   (InputIterator first, InputIterator last,
   RandomAccessIterator result_first,
   RandomAccessIterator result_last [, Compare ] );
```

Because the input to this version of the algorithm is specified only as a pair of input iterators, the `partial_sort_copy()` algorithm can be used with any of the containers in the Standard C++ Library. In the example program, it is used with a *list*:

```
void partial_sort_example ()

      // illustrates the use of the partial sort algorithm
      // see alg7.cpp for complete source code
{
      // make a vector of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);

      // partial sort the first seven positions
   partial_sort (aVec.begin(), aVec.begin() + 7, aVec.end());

      // make a list of random integers
   list<int> aList(15, 0);
   generate (aList.begin(), aList.end(), randomValue);

      // sort only the first seven elements
   vector<int> start(7);
   partial_sort_copy (aList.begin(), aList.end(),
      start.begin(), start.end(), greater<int>());
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

nth Element

Imagine we have the sequence `2 5 3 4 7`, and we want to discover the median or middle element. If we do this with the function `nth_element()`, one result might be the following sequence:

```
3 2 | 4 | 7 5
```

The vertical bars are used to describe the separation of the result into three parts: the elements before the requested value, the requested value, and the values after the requested value. Note that the values in the first and third sequences are unordered; in fact, they can appear in the result in any order. The only requirement is that the values in the first part are no larger than the value we are seeking, and the elements in the third part are no smaller than this value.

The three iterators provided as arguments to the algorithm `nth_element()` divide the argument sequence into the three sections we just described. These are the section prior to the middle iterator, the single value denoted by the middle iterator, and the region between the middle iterator and the end. Either the first or third of these may be empty.

The arguments to the algorithm can be described as follows:

```
void nth_element (RandomAccessIterator first,
   RandomAccessIterator nth,
   RandomAccessIterator last [, Compare ] );
```

Following the call on `nth_element()`, the nth largest value is copied into the position denoted by the middle ***iterator***. The region between the first ***iterator*** and the middle iterator will have values no larger than the nth element, while the region between the middle ***iterator*** and the end will hold values no smaller than the nth element.

The example program illustrates finding the fifth largest value in a ***vector*** of random numbers.

```
void nth_element_example ()

      // illustrates the use of the nth_element algorithm
      // see alg7.cpp for complete source code
{
      // make a vector of random integers
   vector<int> aVec(10);
   generate (aVec.begin(), aVec.end(), randomValue);

      // now find the 5th largest
   vector<int>::iterator nth = aVec.begin() + 4;
   nth_element (aVec.begin(), nth, aVec.end());

   cout << "fifth largest is " << *nth << endl;
}
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Binary Search

The Standard C++ Library provides a number of different variations on binary search algorithms. All perform only approximately `log n` comparisons, where `n` is the number of elements in the range described by the arguments. The algorithms work best with random access ***iterator***s, like those generated by ***vector***s or ***deque***s, when they also perform approximately `log n` operations in total. However, they also work with non-random access ***iterator***s, like those generated by ***list***s, in which case they perform a linear number of steps. Although legal, it is not necessary to perform a binary search on a ***set*** or ***multiset*** data structure, since those container classes provide their own search methods, which are more efficient.

The generic algorithm `binary_search()` returns `true` if the sequence contains a value that is equivalent to the argument. Recall that to be equivalent means that both `Compare(value, arg)` and `Compare(arg, value)` are `false`. The algorithm is declared as follows:

```
bool binary_search (ForwardIterator first, ForwardIterator last,
      const T & value [, Compare ] );
```

In other situations it is important to know the position of the matching value. This information is returned by a collection of algorithms, defined as follows:

```
ForwardIterator last, const T& value [, Compare ] );

ForwardIterator upper_bound (ForwardIterator first,
   ForwardIterator last, const T& value [, Compare ] );

pair<ForwardIterator, ForwardIterator> equal_range
   (ForwardIterator first, ForwardIterator last,
      const T& value [, Compare ] );
```

The algorithm `lower_bound()` returns, as an ***iterator***, the first position into which the argument could be inserted without violating the ordering, whereas the algorithm `upper_bound()` finds the last such position. These match only when the element is not currently found in the sequence. Both can be executed together in the algorithm `equal_range()`, which returns a pair of ***iterator***s.

Our example program shows these functions being used with a ***vector*** of random integers.

```
void binary_search_example ()

   // illustrates the use of the binary search algorithm
   // see alg7.cpp for complete source code
{
      // make an ordered vector of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   sort (aVec.begin(), aVec.end());

      // see if it contains an eleven
   if (binary_search (aVec.begin(), aVec.end(), 11))
      cout << "contains an 11" << endl;
   else
      cout << "does not contain an 11" << endl;

      // insert an 11 and a 14
   vector<int>::iterator where;
   where = lower_bound (aVec.begin(), aVec.end(), 11);
   aVec.insert (where, 11);

   where = upper_bound (aVec.begin(), aVec.end(), 14);
   aVec.insert (where, 14);

}
```


©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Merge Ordered Sequences

The algorithm `merge()` combines two ordered sequences to form a new ordered sequence. The size of the result is the sum of the sizes of the two argument sequences. This should be contrasted with the `set_union()` operation, which eliminates elements that are duplicated in both sets. The `set_union()` function is described later in this chapter.

The merge operation is stable. This means, for equal elements in the two ranges, not only is the relative ordering of values from each range preserved, but the values from the first range always precede the elements from the second. The two ranges are described by a pair of ***iterator***s, whereas the result is defined by a single output ***iterator***. The arguments are shown in the following declaration:

```
OutputIterator merge (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2,
   OutputIterator result [, Compare ]);
```

The example program illustrates a simple merge, the use of a merge with an inserter, and the use of a merge with an output stream ***iterator***.

```
void merge_example ()

   // illustrates the use of the merge algorithm
   // see alg7.cpp for complete source code

{
      // make a list and vector of 10 random integers
   vector<int> aVec(10);
   list<int> aList(10, 0);
   generate (aVec.begin(), aVec.end(), randomValue);
   sort (aVec.begin(), aVec.end());
   generate_n (aList.begin(), 10, randomValue);
   aList.sort();

      // merge into a vector
   vector<int> vResult (aVec.size() + aList.size());
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      vResult.begin());

      // merge into a list
   list<int> lResult;
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      inserter(lResult, lResult.begin()));

      // merge into the output
   merge (aVec.begin(), aVec.end(), aList.begin(), aList.end(),
      ostream_iterator<int,char> (cout, " "));
   cout << endl;

}
```

The algorithm `inplace_merge()` (Section 13.4.6) can be used to merge two sections of a single sequence into one sequence.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

set Operations

The operations of `set union`, `set intersection`, and `set difference` were all described in Section 8.2.7 when we discussed the ***set*** container class. However, the algorithms that implement these operations are generic, and applicable to any ordered data structure. The algorithms assume the input ranges are ordered collections that represent ***multiset***s; that is, elements can be repeated. However, if the inputs represent ***set***s, then the result will always be a ***set***. Unlike the `merge()` algorithm, none of the ***set*** algorithms produce repeated elements in the output that are not present in the input ***set***s.

The ***set*** operations all have the same format. The two input ***set***s are specified by pairs of input ***iterator***s. The output ***set*** is specified by an input ***iterator***, and the end of this range is returned as the result value. An optional comparison operator is the final argument. In all cases it is required that the output sequence not overlap in any manner with either of the input sequences.

```
OutputIterator set_union
   (InputIterator first1, InputIterator last1,
   InputIterator first2, InputIterator last2,
   OutputIterator result [, Compare ] );
```

The example program illustrates the use of the four ***set*** algorithms, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference`. It also shows a call on `merge()` in order to contrast the merge and the set union operations. The algorithm `includes()` is slightly different. Again the two input sets are specified by pairs of input ***iterator***s, and the comparison operator is an optional fifth argument. The return value for the algorithm is `true` if the first ***set*** is entirely included in the second, and `false` otherwise.

```
void set_example ()

   // illustrates the use of the generic set algorithms
   // see alg7.cpp for complete source code
{
   ostream_iterator<int> intOut (cout, " ");

      // make a couple of ordered lists
   list<int> listOne, listTwo;
   generate_n (inserter(listOne, listOne.begin()), 5, iotaGen(1));
   generate_n (inserter(listTwo, listTwo.begin()), 5, iotaGen(3));

      // now do the set operations
      // union - 1 2 3 4 5 6 7
   set_union (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // merge - 1 2 3 3 4 4 5 5 6 7
   merge (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // intersection - 3 4 5
   set_intersection (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // difference - 1 2
   set_difference (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;
      // symmetric difference - 1 2 6 7
   set_symmetric_difference (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end(), intOut), cout << endl;

   if (includes (listOne.begin(), listOne.end(),
      listTwo.begin(), listTwo.end()))
         cout << "set is subset" << endl;
   else
      cout << "set is not subset" << endl;
}
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

heap Operations

A *heap* is a binary tree in which every node is larger than the values associated with either child. A heap and a binary tree, for that matter, can be very efficiently stored in a ***vector***, by placing the children of node `i` in positions `2 * i + 1` and `2 * i + 2`.

Using this encoding, the largest value in the heap is always located in the initial position, and can therefore be very efficiently retrieved. In addition, efficient logarithmic algorithms exist that permit a new element to be added to a heap and the largest element removed from a heap. For these reasons, a heap is a natural representation for the ***priority queue*** datatype, described in Chapter 11.

The default operator is the less-than operator `<` appropriate to the element type. If desired, an alternative operator can be specified. For example, by using the greater-than operator `>`, one can construct a heap that locates the smallest element in the first location, instead of the largest.

The algorithm `make_heap()` takes a range, specified by random access ***iterator***s, and converts it into a heap. The number of steps required is a linear function of the number of elements in the range.

```
void make_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

To add a new element to a heap, insert it at the end of a range using the `push_back()` member function of a ***vector*** or ***deque***, for example, and invoke the algorithm `push_heap()`. The `push_heap()` algorithm restores the heap property, performing at most a logarithmic number of operations.

```
void push_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

The algorithm `pop_heap()` swaps the first and final elements in a range, and restores to a heap the collection without the final element. The largest value of the original collection is therefore still available as the last element in the range. It can be accessed using the `back()` member function in a ***vector***, for example, and removed using the `pop_back()` member function. At the same time, the remainder of the collection continues to have the heap property. The `pop_heap()` algorithm performs at most a logarithmic number of operations.

```
void pop_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

Finally, the algorithm `sort_heap()` converts a heap into an ordered or *sorted* collection. Note that a sorted collection is still a heap, although the reverse is not the case.

NOTE: An ordered collection is a heap, but a heap need not necessarily be an ordered collection. In fact, a heap can be constructed in a sequence much more quickly than the sequence can be sorted.

The sort is performed using approximately `n log n` operations, where `n` represents the number of elements in the range. The `sort_heap()` algorithm is not stable.

```
void sort_heap (RandomAccessIterator first,
      RandomAccessIterator last [, Compare ]);
```

Here is an example program that illustrates the use of these functions:

```
void heap_example ()

   // illustrates the use of the heap algorithms
   // see alg7.cpp for complete source code

{
      // make a heap of 15 random integers
   vector<int> aVec(15);
   generate (aVec.begin(), aVec.end(), randomValue);
   make_heap (aVec.begin(), aVec.end());
   cout << "Largest value " << aVec.front() << endl;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
      // remove largest and reheap
   pop_heap (aVec.begin(), aVec.end());
   aVec.pop_back();

      // add a 97 to the heap
   aVec.push_back (97);
   push_heap (aVec.begin(), aVec.end());

      // finally, make into a sorted collection
   sort_heap (aVec.begin(), aVec.end());
}
```

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Part V: Special Techniques

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 15: Using Allocators

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

An Overview

The Standard C++ allocator interface encapsulates the types and functions needed to manage the storage of data in a generic way. The interface provides:

- pointer and reference types
- the type of the difference between pointers
- the type to describe the size of a block of storage
- storage allocation and deallocation primitives
- object construction and destruction primitives.

This allocator interface wraps the mechanism for managing data storage, and separates this mechanism from the classes and functions used to maintain associations between data elements. This eliminates the need to rewrite containers and algorithms to suit different storage mechanisms. The interface lets you encapsulate all the storage mechanism details in an allocator, then provide that allocator to an existing container when appropriate.

The Standard C++ Library provides a default allocator class, ***allocator***, that implements this interface using the standard `new` and `delete` operators for all storage management.

This chapter briefly describes how to use ***allocator***s with existing containers, then discusses what you need to consider when designing your own ***allocator***s. Chapter 16 describes what you must consider when designing containers that use ***allocator***s.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Using Allocators with Existing Standard Library Containers

Using allocators with existing Standard C++ Library container classes is a simple process. You just provide an allocator type when you instantiate a container, and provide an actual allocator object when you construct a container object:

```
my_allocator alloc<int>;                  // Construct an allocator
vector<int,my_allocator<int> > v(alloc);  // Use the allocator
```

All standard containers default the allocator template parameter type to `allocator<T>` and the object to `Allocator()`, where `Allocator` is the template parameter type. This means that the simplest use of allocators is to ignore them entirely. When you do not specify an allocator, the default allocator is used for all storage management.

If you do provide a different allocator type as a template parameter, the type of object that you provide must match the template type. For example, the following code will cause a compiler error because the types in the template signature and the call to the allocator constructor don't match:

```
template <class T> class my_alloc;
list <int, allocator<int> > my_list(my_alloc()); \\ Wrong!
```

The following call to the allocator constructor does match the template signature:

```
list <int, my_alloc<int> > my_list(my_alloc());
```

It's also important that the type used for the allocator template parameter and the type used as the element type in a standard container agree. For instance:

```
list<int, allocator<long>>                        \\ Wrong!
```

won't work. Remember that with a ***map*** the contained type is actually a key-value pair:

```
map<int,long,less<int>,allocator<pair<int,long>>>
```

Note that the container always holds a *copy* of the ***allocator*** object that is passed to the constructor. If you need a single ***allocator*** object to manage all storage for a number of containers, you must provide an ***allocator*** that maintains a reference to some shared implementation.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Building Your Own Allocators

Defining your own ***allocator*** is a relatively simple process. The Standard C++ Library describes a particular interface, consisting of types and functions. An allocator that conforms to the standard must match the syntactic requirements for these member functions and types. The Standard C++ Library also specifies a portion of the semantics for the allocator type.

The Standard C++ Library allocator interface relies heavily on member templates. As of this writing, many compilers do not yet support both member function templates and member class templates. This makes it impossible to implement a standard allocator. Rogue Wave's implementation of the Standard C++ Library provides an alternative allocator interface that provides most of the power of the standard interface, without requiring unavailable compiler features. This interface differs significantly from the standard interface, and will not work with other vendors' versions of the Standard C++ Library.

We recommend that when you define an allocator and implement containers, you provide both the standard interface and the Rogue Wave interface. This allows you to use allocators now, and to take advantage of the standard once it becomes available on your compiler.

The remainder of this chapter describes the requirements for the Standard C++ Library allocator, the requirements for Rogue Wave's alternative allocator, and some techniques that specify how to support both interfaces in the same code base.

Using the Standard Allocator Interface

An ***allocator*** that conforms to the Standard C++ Library allocator specification must have the following interface. The example uses `my_allocator` as a place holder for your own allocator name:

```
template <class T>
class my_allocator
{
  typedef implementation_defined size_type;
  typedef implementation_defined difference_type
  typedef implementation_defined pointer;
  typedef implementation_defined const_pointer;
  typedef implementation_defined reference;
  typedef implementation_defined const_reference;
  typedef implementation_defined value_type;

  template <class U>
  struct rebind { typedef allocator<U> other; };
```

Each of the pointer types in this interface must have a conversion to `void*`. It must be possible to use the resulting `void*` as a `this` value in a constructor or destructor and in conversions to `B<void>::pointer`, for appropriate `B`, for use by `B::deallocate()`.

The rebind member allows a container to construct an allocator for some arbitrary type out of the allocator type provided as a template parameter. For instance, the ***list*** container gets an ***allocator<T>*** by default, but a ***list*** may well need to allocate ***list_node***s as well as ***T***s. The container can construct an allocator for ***list_node***s out of the allocator for ***T***, which is the template parameter ***Allocator*** in this case, as follows:

```
Allocator::rebind<list_node>::other list_node_allocator;
```

Here is a description of the member functions that a Standard C++ Library ***allocator*** must provide:

```
my_allocator();
template <class U>
my_allocator(const my_allocator<U>&);
template <class U>
```

Constructors.

```
~my_allocator();
```

©Copyright 1999 Rogue Wave Software, Inc.

Destructor.

```
operator=(const my_allocator<U>&);
```

Assignment operator.

```
pointer address(reference r) const;
```

Returns the address of `r` as a `pointer` type. This function and the following function are used to convert references to pointers.

```
const_pointer address(const_reference r) const;
```

Returns the address of `r` as a `const_pointer` type.

```
pointer allocate(size_type n, const_pointer hint=0);
```

Allocates storage for `n` values of `T`. Uses the value of `hint` to optimize storage placement, if possible.

```
void
deallocate(pointer);
```

Deallocates storage obtained by a call to `allocate`.

```
size_type
max_size();
```

Returns the largest possible storage available through a call to `allocate`.

```
void
construct(pointer p, const T& val);
```

Constructs an object of type `T` at the location of `p`, using the value of `u` in the call to the constructor for `T`. The effect is:

```
new((void*)p) T(u);
```

```
void
destroy(pointer p);
```

Calls the destructor on the value pointed to by `p`. The effect is:

```
(T*)p)->~T()
```

Here is a description of the non-member functions that a Standard C++ Library ***allocator*** must provide:

```
template <class T>
my_allocator::pointer
operator new(my_allocator::size_type, my_allocator&);
```

Allocates space for a single object of type `T` using `my_allocator::allocate`. The effect is:

```
new((void*)x.template allocate<T>(1)) T;
```

```
template <class T, class U>
bool
operator==(const my_allocator<T>& a,
           const my_allocator<U>& b);
```

Returns `true` if allocators `b` and `a` can be safely interchanged. *Safely interchanged* means that `b` could be used to deallocate storage obtained through `a`, and vice versa.

```
template <class T, class U>
bool
operator!=(const my_allocator<T>& a,
           const my_allocator<U>& b);
```

Returns `!(a == b)`.

Using Rogue Wave's Alternative Interface

©Copyright 1999 Rogue Wave Software, Inc.

Rogue Wave provides an alternative allocator interface for those compilers that do not support both class templates and member function templates.

In this interface, the class ***allocator_interface*** provides all types and typed functions. Memory is allocated as raw bytes using the class provide by the `Allocator` template parameter. Functions within ***allocator_interface*** cast appropriately before returning pointer values. Because multiple ***allocator_interface*** objects can attach to a single ***allocator***, one ***allocator*** can allocate all storage for a container, regardless of how many types are involved. The one real restriction is that pointers and references are hard-coded as type `T*` and `T&`. (Note that in the standard interface they are `implementation_defined`.). If your compiler supports partial specialization instead of member templates, you can use it to get around even this restriction by specializing ***allocator_interface*** on just the allocator type.

To implement an ***allocator*** based on the alternative interface, supply the class labeled `my_allocator` below:

```
//
// Alternative allocator uses an interface class
// (allocator_interface)
// to get type safety.
//
template <class T>
class  my_allocator
{
  public:
    typedef implementation_defined size_type;
    typedef implementation_defined difference_type;
    typedef implementation_defined pointer;
    typedef implementation_defined const_pointer;
    typedef implementation_defined reference;
    typedef implementation_defined const_reference;
    typedef implementation_defined value_type;

    my_allocator();
    ~my_allocator();

    void * allocate (size_type n, void *  = 0);
    void deallocate (void* p);
    size_type max_size (size_type size) const
};
```

We also include a listing of the full implementation of the ***allocator_interface*** class, to show how a standard container uses your class. Chapter 16 provides a full description of how the containers use the alternative interface.

```
template <class Allocator,class T>
class allocator_interface
{
public:
  typedef Allocator allocator_type;
  typedef T*         pointer;
  typedef const T*   const_pointer;
  typedef T&         reference;
  typedef const T&   const_reference;
  typedef T          value_type;
  typedef typename Allocator::size_type    size_type;
  typedef typename Allocator::difference_type difference_type;

protected:
  allocator_type*         alloc_;

public:
  allocator_interface() : alloc_(0) { ; }
  allocator_interface(Allocator* a) : alloc_(a) { ; }

  void alloc(Allocator* a)
  {
    alloc_ = a;
  }

  pointer address (T& x)
  {
    return static_cast<pointer>(&x);
  }

  size_type max_size ()  const
  {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    return alloc_->max_size(sizeof(T));
  }

  pointer allocate(size_type n, pointer  = 0)
  {
    return static_cast<pointer>(alloc_->allocate(n*sizeof(T)));
  }

  void deallocate(pointer p)
  {
    alloc_->deallocate(p);
  }

  void construct(pointer p, const T& val)
  {
    new (p) T(val);
  }

  void destroy(T* p)
  {
    ((T*)p)->~T();
  }

};

class allocator_interface<my_allocator,void>
{
  public:
    typedef void*         pointer;
    typedef const void*   const_pointer;
};

//
// allocator globals
//
void * operator new(size_t N, my_allocator& a);
inline void * operator new[](size_t N, my_allocator& a);
inline bool operator==(const my_allocator&, const my_allocator&);
```

How to Support Both Interfaces

Rogue Wave strongly recommends that you implement containers that support both the Standard C++ Library allocator interface, and our alternative interface. By supporting both interfaces, you can use allocators now, and take advantage of the standard once it becomes available on your compiler.

In order to implement both versions of the allocator interface, your containers must have some mechanism for determining whether the standard interface is available. Rogue Wave provides the macro `_RWSTD_ALLOCATOR` in `stdcomp.h` to define whether or not the standard allocator is available. If `_RWSTD_ALLOCATOR` evaluates to `true`, your compiler is capable of handling Standard C++ Library allocators; otherwise, you must use the alternative.

The first place that you use `_RWSTD_ALLOCATOR` is for determining which type names the container must use to reflect the interface. To do this, place the equivalent of the following code in your container class definition:

```
#ifdef RWSTD_ALLOCATOR
    typedef typename Allocator::rebind<T>::other::reference
         reference;
    typedef typename
         Allocator::rebind<T>::other::const_reference
         const_reference;
    typedef typename Allocator::rebind<node>::other::pointer
         link_type;

    typedef Allocator::rebind<T>::other value_allocator;
    typedef Allocator::rebind<node>::other node_allocator;
#else
    typedef typename
       allocator_interface<Allocator,T>::reference reference;
    typedef typename
       allocator_interface<Allocator,T>::const_reference
         const_reference;
    typedef typename
       allocator_interface<Allocator,node>::pointer  link_type;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  Allocator alloc;
  typedef allocator_interface<Allocator,T>  value_allocator;
  typedef allocator_interface<Allocator,node>
                                          node_allocator;
#endif
```

Notice that we use rebind even for the types associated with `T`. This is safest since it ensures that the container will work even if the allocator is instantiated with a different type for the allocator template parameter, for example, `vector<int, allocator<void> >`. This makes our containers more robust. Note also that we provide two allocator types: `value_allocator` and `node_allocator`. You will need to assemble actual allocators inside your container, probably as they're needed. In our example, the mechanism for calling `allocator::allocate` for `T`'s looks like this, regardless which interface is being used:

```
value_allocator(alloc)::allocate(...);
```

In this call we construct an appropriate allocator using its template copy constructor and then call `allocate` on that ***allocator***. One result of this use of the allocator is that any state held by an allocator had better be passed through the copy constructor by reference, so that it is maintained in the one allocator object that we keep around, which is the one passed into the constructor for the container.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 16: Building Containers and Generic Algorithms

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Extending the Library

The adoption of the Standard C++ Library marks a very important development for users of the C++ programming language. Although the library is written in an OOP language and provides plenty of objects, it also employs an entirely different paradigm. This other approach, called *generic programming*, provides a flexible way to apply generic algorithms to a wide variety of different data structures. The flexibility of C++ in combination with this synthesis of two advanced design paradigms results in an unusual and highly-extensible library.

The clearest example of this synthesis is the ability to extend the library with user-defined containers and algorithms. This extension is possible because the definition of data structures has been separated from the definition of generic operations on those structures (Section 1.2.2). The library defines very specific parameters for these two broad groups, giving users some confidence that containers and algorithms from different sources will work together as long as they all meet the specifications of the standard. At the same time, containers encapsulate data and a limited range of operations on that data in classic OOP fashion.

Each standard container is categorized as one of two types: a *sequence* or an *associative container*. A user-defined container need not fit into either of these two groups since the standard also defines rudimentary requirements for a container, but the categorization can be very useful for determining which algorithms will work with a particular container and how efficiently those algorithms will work. In determining the category of a container, the most important characteristics are the *iterator category* and *element ordering*. (See the chapter on each container type, and the listing for each container and iterator in the *Class Reference*.)

Standard C++ Library algorithms can be grouped into categories using a number of different criteria. The most important of these are:

- whether or not the algorithm modifies the contents of a container
- the type of iterator required by the algorithm
- whether or not the algorithm requires a container to be sorted.

An algorithm may also require further state conditions from any container it's applied to. For instance, all the standard ***set*** algorithms require not only that a container is in sorted order, but also that the order of elements is determined using the same compare function or object that will be used by the algorithm.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Building on the Standard Containers

Let's examine a few of the ways you can use existing Standard C++ Library containers to create your own containers. For example, say you want to implement a ***set*** container that enforces unique values that are not inherently sorted. You also want a group of algorithms to operate on that ***set***. The container is certainly a *sequence* but not an a*ssociative container*, since an associative container is sorted by definition. The algorithms will presumably work on other sequences, assuming those sequences provide appropriate iterator types, since the iterator required by a set of algorithms determines the range of containers those algorithms can be applied to. The algorithms will be universally available if they require only forward iterators. On the other hand, they'll be most restrictive if they require random access iterators.

Simple implementations of this ***set*** container could make use of existing Standard C++ Library containers for much of their mechanics. Three possible ways of achieving this code reuse are:

- inheritance
- generic inheritance
- generic composition.

Let's take a look at each of these approaches in the next sections.

Inheritance

As you recall, *inheritance* is the powerful feature of object-oriented programming that allows objects to derive attributes and behavior from other objects. To create your own container, you could derive from an existing Standard C++ container, then override certain functions to get the desired behavior. One approach would be to derive from the ***vector*** container, as shown here:

```
#include <vector>

// note the use of a namespace to avoid conflicts with standard // or global names

namespace my_namespace {

template <class T, class Allocator = std::allocator>
class set : public std::vector<T,Allocator>
{
public:
// override functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(begin(),end(),x) == end())
        return vector<T,Allocator>::insert(position,x);
      else
        return end(); // This value already present!
  }
...

};

} // End of my_namespace
```

Generic Inheritance

A second approach to creating your own container is to create a *generic adaptor*, rather than specifying ***vector***. You do this by providing the underlying container through a template parameter:

```
namespace my_namespace {

template <class T, class Container = std::vector<T> >
class set : public Container
{
public:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
// Provide typedefs (iterator only for illustration)
  typedef  typename Container::iterator iterator;

// override functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(begin(),end(),x) == end())
        return Container::insert(position,x);
      else
        return end();  // This value already present!
  }
...

};

} // End of my_namespace
```

If you use generic inheritance through an adaptor, the adaptor and users of the adaptor cannot expect more than default capabilities and behavior from any container used to instantiate it. If the adaptor or its users expect functionality beyond what is required of a basic container, the documentation must specify precisely what is expected.

Generic Composition

The third approach to building your own container uses *composition* rather than inheritance. You can see the spirit of this approach in the Standard C++ Library adaptors ***queue***, ***priority queue***, and ***stack***. When you use generic composition, you have to implement all of the desired interface. This option is most useful when you want to limit the behavior of an adaptor by providing only a subset of the interface provided by the container.

```
namespace my_namespace {

template <class T, class Container = std::vector<T> >
class set
{
protected:
   Container c;
public:
// Provide needed typedefs
  typedef  typename Container::iterator iterator;

// provide all necessary functions such as insert
  iterator insert (iterator position, const T& x)
  {
      if (find(c.begin(),c.end(),x) == c.end())
        return c.insert(position,x);
      else
        return c.end();  // This value already present!
  }
...

};

} // End of my_namespace
```

The advantages of adapting existing containers are numerous. For instance, you get to reuse the implementation and the specifications of the container that you're adapting.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Creating Your Own Containers

All of the options that build on existing Standard C++ Library containers incur a certain amount of overhead. When performance demands are critical, or the container requirements very specific, there may be no choice but to implement a container from scratch.

When building from scratch, there are three sets of design requirements that you must meet:

- container interface requirements
- allocator interface requirements
- iterator requirements.

We'll talk about each of these in the next sections.

Meeting the Container Requirements

The Standard C++ Library defines general interface requirements for containers, and specific requirements for specialized containers. When you create a container, the first part of your task is making sure that the basic interface requirements for a container are met. In addition, if your container will be a sequence or an associative container, you need to provide all additional pieces specified for those categories. For anything but the simplest container, this is definitely not a task for the faint of heart.

It's very important to meet the requirements so that users of the container will know exactly what capabilities to expect without having to read the code directly. Review the sections on individual containers for information about the container requirements.

Meeting the Allocator Interface Requirements

A user-defined container makes use of the allocator interface for all storage management. An exception to this is a container that exists in a completely self-contained environment where there is no need for substitute allocators.

The basic interface of an ***allocator*** class consists of a set of typedefs, a pair of allocation functions, `allocate` and `deallocate`, and a pair of construction/destruction members, `construct` and `destroy`. The typedefs are used by a container to determine the look of pointers, references, sizes, and differences, where a *difference* means a distance between two pointers. The functions are used to do the actual management of data storage.

To use the allocator interface, a container must meet the following three requirements:

- A container needs to have a set of typedefs that look like the following:

```
typedef Allocator allocator_type;

typedef typename Allocator::size_type size_type;

typedef typename Allocator::difference_type difference_type;

typedef typename Allocator::reference reference;

typedef typename Allocator::const_reference const_reference;

typedef implementation_defined iterator;

typedef implementation_defined iterator;
```

- A container also needs to have an `Allocator` member that contains a copy of the allocator argument provided by the constructors:

©Copyright 1999 Rogue Wave Software, Inc.

```
protected:

Allocator the_allocator;
```

- A container needs to use that `Allocator` member for all storage management. For instance, our ***set*** container might have a naive implementation that simply allocates a large buffer and then constructs values on that buffer. Note that this not a very efficient mechanism, but it serves as a simple example. We're also going to avoid the issue of `Allocator::allocate` throwing an exception, in the interest of brevity.

An abbreviated version of the ***set*** class appears below. The class interface shows the required typedefs and the `Allocator` member for this class:

```
#include <memory>

namespace my_namespace {

template <class T, class Allocator = std::allocator<T> >
class set
{
public:
   // typedefs and allocator member as  above
   typedef Allocator allocator_type;
   typedef typename Allocator::size_type size_type;
   typedef typename Allocator::difference_type
                               difference_type;
   typedef typename Allocator::reference reference;
   typedef typename Allocator::const_reference
                                     const_reference;

   // Our iterator will be a simple pointer
   typedef Allocator::pointer iterator;
   typedef Allocator::const_pointer iterator;

protected:
   Allocator the_allocator;  // copy of the allocator

private:
   size_type buffer_size;
   iterator buffer_start;
   iterator current_end;
   iterator end_of_buffer;

public:
   // A constructor that initializes the set using a range
   // from some other container or array
   template <class Iterator>
   set(Iterator start, Iterator finish,
       Allocator alloc = Allocator());

   iterator begin() { return buffer_start; }
   iterator end() { return current_end; }
};
```

Given this class interface, here's a definition of a possible constructor that uses the allocator. The numbered comments following this code briefly describe the allocator's role. For a fuller treatment of allocators, see Chapter 15 and the *Class Reference* entry for allocators.

```
template <class T, class Allocator>
template <class Iterator>
set<T,Allocator>::set(Iterator start, Iterator finish,
    Allocator alloc)
  : buffer_size(finish-start + DEFAULT_CUSHION),
    buffer_start(0),
    current_end(0), end_of_buffer(0)
{
   // Copy the argument to our internal object
   the_allocator = alloc;                              // 1

   // Create an initial buffer
   buffer_start = the_allocator.allocate(buffer_size);  // 2
   end_of_buffer = buffer_start + buffer_size;

   // Construct new values from iterator range on the buffer
   for (current_end = buffer_start;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
        start != finish;
        current_end++, start++)
      the_allocator.construct(current_end,*start);      // 3

   // Now let's remove duplicates using a standard algorithm
   std::unique(begin(),end());
}


} // End of my_namespace
```

//1 The allocator parameter is copied into a protected member of the container. This private copy can then be used for all subsequent storage management.

//2 An initial buffer is allocated using the allocator's allocate function.

//3 The contents of the buffer are initialized using the values from the iterator range supplied to the constructor by the `start` and `finish` parameters. The `construct` function constructs an object at a particular location. In this case the location is at an index in the container's buffer.

Iterator Requirements

Every container must define an iterator type. Iterators allow algorithms to iterate over the container's contents. Although iterators can range from simple to very complex, it is not the complexity but the *iterator category* that most affects an algorithm. The iterator category describes capabilities of the iterator, such as which direction it can traverse. Section 16.4 and the iterator entries in the *Class Reference* provide additional information about iterator categories.

The example in Section 16.3.2 shows the implementation of a container that uses a simple pointer. A simple pointer is actually an example of the most powerful type of iterator: the *random access iterator*. If an iterator supports random access, we can add to or subtract from it as easily as we can increment it.

Some iterators have much less capability. For example, consider an iterator attached to a singly-linked *list*. Since each node in the ***list*** has links leading forward only, a naive iterator can advance through the container in only one direction. An iterator with this limitation falls into the category of forward iterator.

Certain member functions such as `begin()` and `end()` produce iterators for a container. A container's description should always describe the category of iterator that its member functions produce. That way, a user of the container can see immediately which algorithms can operate successfully on the container.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Tips and Techniques for Building Algorithms

This section describes some techniques that use features of iterators to increase the flexibility and efficiency of your algorithms.

The iterator_traits Template

Sometimes an algorithm that can be implemented most efficiently with a random access iterator can also work with less powerful iterators. The Standard C++ Library includes primitives that allow a single algorithm to provide several different implementations, depending upon the power of the iterator passed into it. The following example demonstrates the usual technique for setting up multiple versions of the same algorithm:

```
// Note, this requires that the iterators be derived from
// Standard base types, unless the iterators are simple pointers.

namespace my_namespace {

template <class Iterator>
Iterator union(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result)
{
  return union_aux(first1,last1,first2,last2,Result,
                   iterator_traits<first1>());
}

template <class Iterator>
Iterator union_aux(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result, forward_iterator_tag)
{
  // General but less efficient implementation
}

template <class Iterator>
Iterator union_aux(Iterator first1, Iterator last1,
               Iterator first2, Iterator last2,
               Iterator Result,
               random_access_iterator_tag)
{
  // More efficient implementation
}

} // End of my_namespace
```

The `iterator_traits` template provides typedefs for value, difference, pointer, reference, and category types that are based on the type used to instantiate the template. In the example above, we use `iterator_traits::iterator_category` to determine the capabilities of the iterator, and then use specializations to get the best available implementation of the algorithm. In order for `iterator_traits` to work, the iterator provided to the algorithm must be a simple pointer type or be derived from the iterator template, or it must itself define the types for `value_type`, `difference_type`, `pointer`, `reference`, and `iterator_category`. The `iterator_category` type must be one of the following: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, or `random_access_iterator_tag`.

Note that when you use the `iterator_traits` template, the default implementation of an algorithm should expect at most a forward iterator. This default version is used if the algorithm encounters an iterator that is not a simple pointer or derived from a basic standard iterator. Note that input and output iterators are less capable than forward iterators, but that the requirements of algorithms generally mandate read/write capabilities.

Not also that `iterator_traits` only works with compilers that support partial specialization, since the specialization of `iterator_traits` for pointer types uses this feature. If your compiler doesn't support partial specialization, you can use the primitive `__iterator_category()`. Calling this function with an iterator argument returns the same tag you would get by using `iterator_traits`. For example, we could substitute the following line for the use of `iterator_traits` in the example above:

©Copyright 1999 Rogue Wave Software, Inc.

```
return union_aux(first1,last1,first2,last2,Result,
                 __iterator_category(first1));
```

Use `iterator_traits::value_type` and `iterator_traits::difference_type` to discover the type of value pointed to by an iterator, or the type that represents a distance between iterators. As with the category type, you must use the alternate functions `__value_type()` or `__distance_type()` when partial specialization is not available. Both of these functions take an iterator as an argument in just the same way as `__iterator_category()`.

The distance and advance Primitives

The `value_type` primitive lets you determine the type of value pointed to by an iterator. Similarly, you can use the `distance_type` primitive to get a type that represents distances between iterators.

In order to efficiently find the distance between two iterators, regardless of their capabilities, you can use the `distance` primitive. The `distance` primitive uses the technique in Section 16.4.1 to send a calling program to one of four different implementations. This offers a considerable gain in efficiency, since an implementation for a forward iterator must step through the range defined by the two iterators:

```
Distance d = 0;
while (start++ != end)
  d++;
```

whereas an implementation for a random access iterator can simply subtract the start iterator from the end iterator:

```
Distance d = end - start;
```

Similar gains are available with the `advance` primitive, which allows you to step forward or backward an arbitrary number of steps as efficiently as possible for a particular iterator.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 17: The Traits Parameter

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Defining the Problem

The traits technique discussed in this chapter is useful for solving the following kind of problem. Consider that you have a matrix that must work for all types of numbers, but the behavior of the matrix depends on the type of `number` in at least some measure. This means your matrix can't handle all numbers the same way.

Except for the behavioral difference, it sounds like the perfect problem for a template. But you can't use a single template, since you can't hang extra information on the number type because it's often just a built-in type. The template will do the same thing for every number type, which is just what you can't do in this case. You could specialize, but then you have to re-implement the entire matrix class for every type of number. It may well be that most of the class is the same. Worse yet, if you want to leave your interface open for use with some unknown future type, you're requiring that future user to reimplement the entire class as well.

What you really want is to put everything that doesn't change in one place, and repeatedly specify only the small part that does change with the type. The technique for doing this is generally referred to as using a *traits parameter*.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Using the Traits Technique

To implement a traits parameter for a class, you add it as an extra template parameter to your class. You then supply a class for this parameter that encapsulates all the specific operations. Usually that class is itself a template.

As an example, let's look at the matrix problem described in Section 17.1. When you want to add a new type to the matrix, you can use the traits technique and simply specialize the traits class, not the entire matrix. You do no more work than you have to, and you retain the ability to use the matrix on any reasonable number.

Here's how the matrix traits template and specializations for `long` and `int` might look. The example also includes a skeleton of the matrix class that uses the traits template.

```
template <class Num>
class matrix_traits
{
  // traits functions and literals
};

template <class Num, class traits>
class matrix
{
  // matrix
}

class matrix_traits<long>
{
  // traits functions and literals specific to long
};

class matrix_traits<int>
{
  // traits functions and literals specific to int
};

... etc.

matrix<int, matrix_traits<int> > int_matrix;
matrix<long, matrix_traits<long> > long_matrix;
```

Of course you don't even have to specialize on `matrix_traits`. You just have to make sure that you provide the interface that `matrix` expects from its traits template parameter.

Most of the time, the operations contained in a traits class are static functions so there's no need to actually instantiate a traits object.

The Standard Library uses this technique to give the ***string*** class maximum flexibility and efficiency across a wide range of types. The ***char_traits*** traits class provides elementary operations on character arrays. In the simplest case, this means providing ***string*** a `wstring` with access to the C library functions for skinny and wide characters, for example `Strcpy` and `wcstrcpy`.

See the ***char_traits*** entry in the *Class Reference* for a complete description of the traits class.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 18: Exception Handling

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Overview

The Standard C++ Library provides a set of classes for reporting errors. These classes use the exception handling facility of the language. The library implements a particular error model, which divides errors in two broad categories: *logic errors* and *runtime errors*.

Logic errors are errors caused by problems in the internal logic of the program. They are generally preventable.

Runtime errors, on the other hand, are generally not preventable, or at least not predictable. These are errors generated by circumstances outside the control of the program, such as peripheral hardware faults.

Include Files

Programs that use the exception handling classes must include the following header file:

```
#include <stdexcept>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The Standard Exception Hierarchy

The Standard C++ Library implements the two-category error model described in Section 18.1 with a set of classes. These classes, which are defined in the `stdexcept` header file, can be used to catch exceptions thrown by the library and to throw exceptions from your own code.

The exception handling classes are related through inheritance. The inheritance hierarchy looks like this:

```
exception
    logic_error
       domain_error
      invalid_argument
      length_error
      out_of_range
   runtime_error
      range_error
      overflow_error
      underflow_error
```

Classes ***logic_error*** and ***runtime_error*** inherit from class ***exception***. All other exception handling classes inherit from either ***logic_error*** or ***runtime_error***.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Using Exceptions

All exceptions thrown explicitly by any element of the library are guaranteed to be part of the Standard C++ Library exception hierarchy. Please review the *Class Reference* entries for these classes to determine which functions throw which exceptions. You can then choose to catch particular exceptions, or catch any that might be thrown by specifying the base class exception.

For instance, if you are going to call the `insert` function on ***string*** with a position value that could at some point be invalid, you should use code like this:

```
string s;
int n;
...
try
{
s.insert(n,"Howdy");
}
catch (const exception& e)
{
   // deal with the exception
}
```

To throw your own exceptions, simply construct an exception of an appropriate type, assign it an appropriate message, and throw it. For example:

```
...
if (n > max)
   throw out_of_range("You're past the end, bud");
```

The class ***exception*** serves as the base class for all other exception classes. As such it defines a standard interface. This interface includes the `what()` member function, which returns a null-terminated string that represents the message that was thrown with the exception. This function is likely to be most useful in a catch clause, as demonstrated in the example program in Section 18.4.

The class ***exception*** does not contain a constructor that takes a message string, although it can be thrown without a message. Calling `what()` on an exception object returns a default message. All classes derived from ***exception*** *do* provide a constructor that allows you to specify a particular message.

To throw a base exception, you could use the following code:

```
throw exception;
```

This is generally not very useful, however, since whatever catches this exception has no idea what kind of error has occurred. Instead of a base exception, you will usually throw a derived class such as ***logic_error*** or one of its derivations, such as ***out_of_range*** in the example above. Better still, you can extend the hierarchy by deriving your own classes. This allows you to provide error reporting specific to your particular problem. For instance:

```
class bad_packet_error : public runtime_error
{
   public:
      bad_packet_error(const string& what);
};

if (bad_packet())
   throw bad_packet_error("Packet size incorrect");
```

This section has demonstrated how the Standard C++ Library exception classes provide you with a basic error model. From this foundation, you can build the right error detection and reporting methods required for your particular application.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Example Program: Exceptions

NOTE: This program is in the file exceptn.cpp.

This following example program demonstrates the use of exceptions:

```
#include <stdexcept>
#include <string>

static void f() { throw runtime_error("a runtime error"); }

int main ()
{
  string s;

  // First we'll try to incite then catch an exception
  // from the Standard C++ Library string class.
  // We'll try to replace at a position that is non-existent.
  //
  // By wrapping the body of main in a try-catch block we can be
  // assured that we'll catch all exceptions in the exception
  // hierarchy. You can simply catch an exception as is done below,
  // or you can catch each of the exceptions in which you have an
  // interest.
  try
  {
    s.replace(100,1,1,'c');
  }
  catch (const exception& e)
  {
    cout << "Got an exception: " << e.what() << endl;
  }

  // Now we'll throw our own exception using the function
  // defined above.
  try
  {
    f();
  }
  catch (const exception& e)
  {
    cout << "Got an exception: " << e.what() << endl;
  }

 return 0;
}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part VI: Special Classes

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 19: auto_ptr

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Overview

The ***auto_ptr*** class wraps any pointer obtained through `new` and provides automatic deletion of that pointer. The pointer wrapped by an ***auto_ptr*** object is deleted when the ***auto_ptr*** itself is destroyed.

Include File

To access the ***auto_ptr*** class, include the memory header file:

```
#include <memory>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Declaration and Initialization of Autopointers

You attach an ***auto_ptr*** object to a pointer either by using one of the constructors for ***auto_ptr***, by assigning one ***auto_ptr*** object to another, or by using the reset member function. Only one ***auto_ptr*** owns a particular pointer at any one time, except for the NULL pointer, which all ***auto_ptr***s own by default. Any use of the ***auto_ptr*** copy constructor or assignment operator transfers ownership from one ***auto_ptr*** object to another. For instance, suppose we create an ***auto_ptr*** `a` like this:

```
auto_ptr<string> a(new string);
```

The ***auto_ptr*** object `a` now owns the newly created pointer. When `a` is destroyed, such as when it goes out of scope, the pointer is deleted. But if we assign `a` to `b` using the assignment operator:

```
auto_ptr<string> b = a;
```

`b` now owns the pointer. Use of the assignment operator causes `a` to release ownership of the pointer. Now if `a` goes out of scope the pointer is not affected. However, the pointer *is* deleted when `b` goes out of scope.

The use of `new` within the constructor for `a` may seem a little odd. Normally we avoid constructs like this since it puts the responsibility for deletion on a different entity than the one responsible for allocation. In this case, however, the sole responsibility of the ***auto_ptr*** is to manage the deletion. This syntax is actually preferable since it prevents us from accidentally deleting the pointer ourselves.

Use `operator*`, `operator->` or the member function `get()` to access the pointer held by an ***auto_ptr***. For instance, we can use any of the three following statements to assign `"What's up Doc"` to the string now pointed to by the ***auto_ptr*** `b`:

```
*b = "What's up Doc";
*(b.get()) = "What's up Doc";
 b->assign("What's up Doc");
```

Class ***auto_ptr*** also provides a release member function that releases ownership of a pointer. Any ***auto_ptr*** that does not own a specific pointer is assumed to point to the NULL pointer, so calling release on an ***auto_ptr*** will set it to the NULL pointer. In the example above, when `a` is assigned to `b`, the pointer held by `a` is released and `a` is set to the NULL pointer.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 20: complex

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Overview

The class ***complex*** is a template class used to create objects for representing and manipulating complex numbers. The operations defined on complex numbers allow them to be freely intermixed with the other numeric types available in the C++ language, thereby permitting numeric software to be easily and naturally expressed.

Include Files

Programs that use complex numbers must include the `complex` header file:

```
# include <complex>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Creating and Using Complex Numbers

In the following sections we describe the operations used to create and manipulate complex numbers.

Declaring Complex Numbers

The template argument is used to define the types associated with the real and imaginary fields. This argument must be one of the floating point number datatypes available in the C++ language, either `float`, `double`, or `long double`.

There are several constructors associated with the class. A constructor with no arguments initializes both the real and imaginary fields to zero. A constructor with a single argument initializes the real field to the given value, and the imaginary value to zero. A constructor with two arguments initializes both real and imaginary fields. Finally, a copy constructor can be used to initialize a complex number with values derived from another complex number.

```
complex<double> com_one;              // value 0 + 0i
complex<double> com_two(3.14);        // value 3.14 + 0i
complex<double> com_three(1.5, 3.14)  // value 1.5 + 3.14i
complex<double> com_four(com_two);    // value is also 3.14 + 0i
```

A complex number can be assigned the value of another complex number. Since the one-argument constructor is also used for a conversion operator, a complex number can also be assigned the value of a real number. The real field is changed to the right-hand side, while the imaginary field is set to zero:

```
com_one = com_three;                   // becomes 1.5 + 3.14i
com_three = 2.17;                      // becomes 2.17 + 0i
```

The function `polar()` can be used to construct a complex number with the given magnitude and phase angle:

```
com_four = polar(5.6, 1.8);
```

The conjugate of a complex number is formed using the function `conj()`. If a complex number represents `x + iy`, then the conjugate is the value `x-iy`.

```
complex<double> com_five = conj(com_four);
```

Accessing Complex Number Values

The member functions `real()` and `imag()` return the real and imaginary fields of a complex number, respectively. These functions can also be invoked as ordinary functions with complex number arguments.

```
// the following should be the same
cout << com_one.real() << "+" << com_one.imag() << "i" << endl;
cout << real(com_one) << "+" << imag(com_one) << "i" << endl;
```

NOTE: With the exception of the member functions `real()` and `imag()`, most operations on complex numbers are performed using ordinary functions, not member functions.

Arithmetic Operations

The arithmetic operators `+`, `-`, `*`, and `/` can be used to perform addition, subtraction, multiplication, and division of complex numbers. All four work either with two complex numbers, or with a complex number and a real value. Assignment operators are also defined for all four.

```
cout << com_one + com_two << endl;
cout << com_one - 3.14 << endl;
cout << 2.75 * com_two << endl;
com_one += com_three / 2.0;
```

The unary operators `+` and `-` can also be applied to complex numbers.

©Copyright 1999 Rogue Wave Software, Inc.

Comparing Complex Values

Two complex numbers can be compared for equality or inequality, using the operators `==` and `!=`. Two values are equal if their corresponding fields are equal. Complex numbers are not well-ordered, and thus cannot be compared using any other relational operator.

Stream Input and Output

Complex numbers can be written to an output stream, or read from an input stream, using the normal stream I/O conventions. A value is written in parentheses, either as `(u)` or `(u,v)`, depending upon whether or not the imaginary value is zero. A value is read as a set of parentheses surrounding two numeric values.

Norm and Absolute Value

The function `norm()` returns the norm of the complex number. This is the sum of the squares of the real and imaginary parts. The function `abs()` returns the absolute value, which is the square root of the norm. Note that both are ordinary functions that take the complex value as an argument, not member functions.

```
cout << norm(com_two) << endl;
cout << abs(com_two) << endl;
```

The directed phase angle of a complex number is yielded by the function `arg()`:

```
cout << com_four << " in polar coordinates is "
   << arg(com_four) << " and " << norm(com_four) << endl;
```

Trigonometric Functions

The trigonometric functions defined for floating point values have all been extended to complex number arguments. These functions are `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()`, and `tanh()`. Each takes a single complex number as argument and returns a complex number as result.

Transcendental Functions

The transcendental functions `exp()`, `log()`, `log10()`, and `sqrt()` have been extended to complex arguments. Each takes a single complex number as argument, and returns a complex number as result.

The Standard C++ Library defines several variations of the exponential function `pow()`. Versions exist to raise a complex number to an integer power, to raise a complex number to a complex power or to a real power, or to raise a real value to a complex power.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Example Program: Roots of a Polynomial

NOTE: This program is in the file complx.cpp.

The roots of a polynomial $ax^2 + bx + c = 0$ are given by the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The following program takes as input three double precision numbers, and returns the complex roots as a pair of values.

```
typedef complex<double> dcomplex;

pair<dcomplex, dcomplex> quadratic
      (dcomplex a, dcomplex b, dcomplex c)
         // return the roots of a quadratic equation
{
   dcomplex root = sqrt(b * b - 4.0 * a * c);
   a *= 2.0;
   return make_pair(
      (-b + root)/a,
      (-b - root)/a);
}
```

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 21: numeric_limits

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Overview

A new feature of the Standard C++ Library is an organized mechanism for describing the characteristics of the fundamental types provided in the execution environment. In older C and C++ libraries, these characteristics were often described by large collections of symbolic constants. For example, the smallest representable value that could be maintained in a character would be found in the constant named `CHAR_MIN`; the similar constant for a `short` would be known as `SHRT_MIN`; a float would be `FLT_MIN`, and so on.

The template class ***numeric_limits*** provides a new and uniform way of representing this information for all numeric types. Instead of using a different symbolic name for each new datatype, the class defines a single static function, named `min()`, which returns the appropriate values. Specializations of this class then provide the exact value for each supported type. In this way, the smallest character value is found as the result of invoking the function `numeric_limits<char>::min()`, while the smallest floating point value is found by invoking `numeric_limits<float>::min()`, and so on.

Using a template class not only greatly reduces the number of symbolic names that need to be defined to describe the operating environment, but it also ensures consistency between the descriptions of the various types.

For the sake of compatibility, the `numeric_limits` mechanism is used as an addition to the symbolic constants used in older C++ libraries, rather than a strict replacement. Thus both mechanisms exist in parallel for the present. However, as the `numeric_limits` technique is more uniform and extensible, it should be expected that over time the older symbolic constants will become outmoded.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Fundamental Datatypes

The Standard C++ Library describes a specific type by providing a specialized implementation of the `numeric_limits` class for the type. Static functions and static constant data members then provide information specific to the type. The Standard C++ Library includes descriptions of the fundamental datatypes given in Table 21.

Table 21 -- Fundamental datatypes of the Standard C++ Library

bool	**char**	**int**	**float**
	signed char	short	double
	unsigned char	long	long double
		wchar_t	
		unsigned short	
		unsigned int	
		unsigned long	

Certain implementations may also provide information on other datatypes. Whether or not an implementation is described can be discovered using the static data member field `is_specialized`. For example, the following is legal, and will indicate that the ***string*** datatype is not described by this mechanism.

```
cout << "are strings described " <<
   numeric_limits<string>::is_specialized << endl;
```

For datatypes that do not have a specialization, the values yielded by the functions and data fields in `numeric_limits` are generally `zero` or `false`.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

numeric_limits Members

Since a number of the fields in the `numeric_limits` structure are meaningful only for floating point values, it is useful to separate the description of the members into *common* fields and *floating-point specific* fields.

Members Common to All Types

Table 22 summarizes the information available through the ***numeric_limits*** static member data fields and functions.

Table 22 -- Information available through numeric_limits

Type	Name	Meaning
`bool`	`is_specialized`	True if a specialization exists, false otherwise
`T`	`min()`	Smallest finite value
`T`	`max()`	Largest finite value
`int`	`radix`	The base of the representation
`int`	`digits`	Number of `radix` digits that can be represented without change
`int`	`digits10`	Number of base-10 digits that can be represented without change
`bool`	`is_signed`	True if the type is signed
`bool`	`is_integer`	True if the type is integer
`bool`	`is_exact`	True if the representation is exact
`bool`	`is_bounded`	True if representation is finite
`bool`	`is_modulo`	True if type is modulo
`bool`	`traps`	True if trapping is implemented for the type

Radix represents the internal base for the representation. For example, most machines use a base 2 radix for integer data values; however, some may also support a representation, such as BCD, that uses a different base. The `digits` field then represents the number of such radix values that can be held in a value. For an integer type, this would be the number of non-sign bits in the representation.

All fundamental types are bounded. However, an implementation might choose to include, for example, an infinite precision integer package that would not be bounded.

A type is *modulo* if the value resulting from the addition of two values can *wrap around*, that is, be smaller than either argument. The fundamental unsigned integer types are all modulo.

Members Specific to Floating Point Values

The members described in Table 23 are either specific to floating point values, or have a meaning slightly different for floating point values than the one described earlier for non-floating datatypes.

Table 23 -- Members specific to floating point values

Type	Name	Meaning
`T`	`min()`	Minimum positive normalized value
`int`	`digits`	Number of digits in the mantissa
`int`	`radix`	Base (or radix) of the exponent representation
`T`	`epsilon()`	Difference between 1 and the least representable value greater than 1
`T`	`round_error()`	A measurement of the rounding error
`int`	`min_exponent`	Minimum negative exponent
`int`	`min_exponent10`	Minimum value such that 10 raised to that power is in range
`int`	`max_exponent`	Maximum positive exponent

©Copyright 1999 Rogue Wave Software, Inc.

`int`	`max_exponent10`	Maximum value such that 10 raised to that power is in range
`bool`	`has_infinity`	True if the type has a representation of positive infinity
`T`	`infinity()`	Representation of infinity, if available
`bool`	`has_quiet_NaN`	True if there is a representation of a quiet \Q\QNot a Number"
`T`	`quiet_NaN()`	Representation of quiet NaN, if available
`bool`	`has_signaling_NaN`	True if there is a representation for a signaling NaN
`T`	`signaling_NaN()`	Representation of signaling NaN, if available
`bool`	`has_denorm`	True if the representation allows denormalized values
`T`	`denorm_min()`	Minimum positive denormalized value
`bool`	`is_iec559`	True if representation adheres to IEC 559 standard.
`bool`	`tinyness_before`	True if `tinyness` is detected before rounding
	`round_style`	Rounding style for type

For the `float` datatype, the value in field `radix`, which represents the base of the exponential representation, is equivalent to the symbolic constant `FLT_RADIX`.

For the types `floatdouble`, and `long double` the value of `epsilon` is also available as `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`.

A `NaN` is a *Not a Number*. It is a representable value that nevertheless does not correspond to any numeric quantity. Many numeric algorithms manipulate such values.

The IEC 559 standard is a standard approved by the International Electrotechnical Commission. It is the same as the IEEE standard 754.

The value returned by the function `round_style()` is one of the following: `round_indeterminate`, `round_toward_zero`, `round_to_nearest`, `round_toward_infinity`, or `round_toward_neg_infinity`.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 22: valarray

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Overview

The C++ language is often used in scientific and engineering work to perform long and difficult operations on arrays of numbers. While the language itself provides the flexibility and efficiency needed for these kind of calculations, the code can become very complex. Naturally, we can use the object-oriented features of the language to encapsulate this complexity, but classes designed to simplify numeric programming without sacrificing performance are notoriously difficult to code correctly. Fortunately, the Standard C++ Library now provides some relief in the form of the ***valarray*** class template.

This class template provides the necessary efficiency in the form of a single-dimensional array. Class ***valarray*** can be used to perform calculations directly on arrays in a single dimension, and to represent higher order arrays with a little more effort. An extended set of subscripting operators provides the basis for building matrices and other more sophisticated classes from the relatively simple and lean ***valarray***.

Performance Issues

Since efficiency is such an important concern in numeric programming, the ***valarray*** class addresses performance in several ways.

First of all, ***valarray*** is designed to give compilers maximum latitude in optimizing operations within the class. This is achieved primarily by avoiding any aliasing of elements within a ***valarray***. This means that every element in a given ***valarray*** is located at its own unique memory address, exactly as it would be in an ordinary array. While this prevents a ***valarray*** implementation from using certain kinds of optimizations, such as aliasing all elements with default values to a single stored value, that loss is more than compensated by a competent optimizer's ability to streamline operations on the array.

Secondly, the ***valarray*** class uses internal optimization templates to ensure that the most efficient method is used to copy data whenever possible. This optimization is possible in part because of the restriction on aliasing.

Lastly, the ***valarray*** class imposes certain requirements on any type used with it. Most of these restrictions are not related to performance; they are simply necessary to allow generalized numeric operations. But one in particular, initialization semantics, exists at least in part to allow optimizations. A summary of these restrictions follows.

Type Restrictions

A ***valarray*** can be instantiated only on a type `T` meeting the following requirements:

- A type `T` must not be an abstract class.
- A type `T` cannot be a reference type; that is, `valarray<int&>` is not allowed.
- A type `T` must not be const-volatile qualified; that is,

  `valarray<const int>` is not allowed.
- A type `T` must not overload unary `operator&`.
- A type `T` must not throw exceptions.
- If a type `T` is a class it must have a public default constructor, a public copy constructor with the signature `T::T(const T&)`, and a public destructor.
- If a type `T` is a class, it must have a public assignment operator with a signature matching one of the following: `T& T::operator=(const T&)` or `T& T::operator=(T)`.
- If a type `T` is a class, then its default constructor, copy constructor, and assignment operator must behave in such a way that there is no difference between default construction followed by assignment, and copy construction. Additionally, destruction of an object followed by copy construction must be equivalent to assignment.

©Copyright 1999 Rogue Wave Software, Inc.

Here is a summary of the necessary relationship between copy construction and assignment, where `T` is the type, `t` and `v` are instances of the type, and `p` is a pointer to an instance of the type:

`T t(), t = v`	*is semantically equivalent to:* `T t(v)`
`new (p) T(), *p = v`	*is semantically equivalent to:* `new (p) T(v)`
`p->~T(), new (p) T(v)`	*is semantically equivalent to:* `*p = v`

This summary demonstrates that there is nothing subtle and clever happening in the process of copy construction or assignment. This consistency is particularly important for portability, since any implementation of ***valarray*** can expect that these operations have the described equivalency even if a particular implementation may not care.

Class That Meets the Type Restrictions

All the built-in numeric types, like `int`, `long`, `float`, and so on, clearly meet the type requirements for ***valarray***. Here is an example of a minimalist class that does, too. This class is simple enough that the compiler generated constructors, destructor, and assignment operator would actually suffice. As you can see, there is nothing very special needed here. Rather, it is the absence of something special that the requirements guard against.

```
class Num
{
  int val_;
public:
  Num() : val_(0)                    // public default constructor
  {}
  Num(const Num& n) : val_(n.val())  // public copy constructor
  {}
  ~Num()                             // public destructor
  {}
  Num& operator=(const Num& n)       // public assignment
  {
    val-_ = n.val();
    return *this;
  }

  int val() const
  { return val_; }
  int val(int v)
  {
    int tmp = val_;
    val_ = v;
    return tmp;
  }
};
```

Class That Doesn't Meet the Type Restrictions

Now let's look at a small variation on the ***Num*** class that does not meet the type requirements for class ***valarray***. Depending on how this class is used, it may or may not be a problem in a ***valarray***, but it most definitely violates the requirements since default construction followed by assignment may result in a different state than copy construction.

```
class Num
{
  bool assigned_;
  int val_;
public:
  Num() : val_(0) , assigned_(false)  // public default constructor
  {}
  Num(const Num& n)                   // public copy constructor
   : val_(n.val()), assigned_(n.assigned())
  {}
  ~Num()                              // public destructor
  {}
  Num& operator=(const Num& n)        // public assignment
  {
    val_ = n.val();
    assigned_ = true;                 // Whoops, violates rule 6 if
                                      // n.assigned() == false.
    return *this;
  }

  int val() const
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  { return val_; }
  int val(int v)
  {
    int tmp = val_;
    val_ = v;
    return tmp;
  }
  bool assigned() const
  { return assigned; }
};
```

Other Unique Features

It is important to note that operations not defined for a particular type are not available for a ***valarray*** of that type. For instance, if a type does not have ordering operations, as is the case with the standard ***complex*** class template, then those ordering operands are not available in a ***valarray*** of that type.

Another important feature is that, along with the ***valarray*** class, the ***valarray*** header defines several auxiliary classes to support extended subset operations. These are `slice`, `gslice`, `slice_array`, `gslice_array`, `indirect_array`, and `mask_array`, which are explained in Section 22.4.2 and its subsections. For now we'll just note that `slice` and `gslice` are used to define the parameters of a BLAS-like slice, or a generalized slice into an array. The remainder of these auxiliary classes define the types returned by subset operations. None of these can be instantiated directly by a program since they lack public constructors.

Header Files

Programs that use ***valarray***s must include the `valarray` header file:

```
#include <valarray>
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Declaring a valarray

A ***valarray*** can be constructed by any of the following means:

- **Default constructor**. The default constructor is provided to allow arrays of ***valarray***s. Use the resize function to adjust the ***valarray*** after construction.
- **Construction and initialization.** A ***valarray*** provides two ways to start with a given size and values:
  - `valarray(const T& v,size_t n)` constructs a ***valarray*** of size `n` with each element initialized to the value `v`.
  - `valarray(const T* p,size_t n)` constructs a ***valarray*** of size `n` with each element initialized to the corresponding element of the array pointed to by `p`. This constructor allows a program to transfer data with maximum efficiency from an ordinary `C' array (resulting from a file operation, for example) into a ***valarray***.
- **Copy constructor**. The copy constructor has value semantics.
- **Conversion constructors**. Class ***valarray*** provides four conversion constructors for converting from auxiliary classes generated by subscript operations. We'll look at these classes in detail and describe the use of the conversion constructors in Section 22.4.2.

The following example shows the use of the first three categories of constructors:

```
#include <valarray>

using std::valarray;

valarray<int> v1;        // construct an empty valarray
valarray<int> v2(1,3);   // construct a valarray of three
                         // elements, all initialized to 1
v1.resize(3,2);          // resize the first valarray to
                         // three elements, all initialized to 2
valarray<int> v3(v1);    // v3 gets a copy of v1's elements.
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Assignment Operators

Class ***valarray*** provides six assignment operations. These are: assignment from another ***valarray***, assignment from a `T` value, and assignment from any of the four auxiliary classes returned by the subset operations.

Assignment from a `T` value assigns that value to all elements in the ***valarray***. Assignment from another ***valarray*** or any of the auxiliary classes assigns corresponding elements from the one to the other. In any case, the left- and right-hand- side arrays must be equal in length.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Element and Subset Access

Ordinary Index Operators

Class ***valarray*** provides `const` and non-`const` versions of `operator[](size_t)`. The const version returns the value of the indicated index, while the non-const version returns a reference to the value.

For example, continuing the example from Section 22.2:

```
v1[2] = 5;            // change the element at position 2 using
                      // the non-const index operator
```

Subset Operators

Along with the ordinary subscript operators, ***valarray*** provides four different subset operations. Each of these also has both `const` and non-`const` versions. The `const` version always returns a new ***valarray*** initialized with data appropriate to the subset operation. The non-`const` version returns an auxiliary array class that references directly the data in the original ***valarray***. Thus each non-`const` version provides a view into the original array by which specific elements may be accessed. Note that an instance of any of these auxiliary classes can only be obtained from the operator. A program cannot instantiate one of these classes directly as they lack a public constructor.

The four subset operations are the `slice` operation, the `gslice` operation, the `mask` operation, and the `indirect` operation. Both `slice` and `gslice` use a special auxiliary class to specify the intent of the operation. These classes are called ***slice*** and ***gslice*** respectively.

The Slice Operation

The `slice` operation provides a view into a ***valarray*** that is like the same operation defined in Basic Linear Algebra Subprograms (BLAS). A *slice* is defined by a starting index, a length, and a stride. To call the `slice` subscript operator, a program passes a ***slice*** object initialized with a constructor taking these three parameters: `slice::slice(size_t start, size_t length, size_t stride)`. The returned array consists only of length elements, starting with the element at the starting index and continuing with each element's stride steps after the previously selected one. The following example illustrates this process:

```
using std::slice;

int a[9] = {1,2,3,4,5,6,7,8,9};
valarray<int> all(a,9);                 //all = {1,2,3,4,5,6,7,8,9}
valarray<int> odd = all[slice(0,5,2)];  //odd = {1,3,5,7,9}
```

In this example, the subscript operation actually returns a ***slice_array***. Class ***valarray*** contains conversion constructors for this class as well as the auxiliary classes returned by other subset operations. Note, however, that the reference semantics are lost in the conversion since odd is an entirely new array. Class ***slice_array*** and the other auxiliary classes exist only to help in the selection of a view and the creation of a new ***valarray*** based on that view.

The gslice Operation

The `gslice` operation differs from the `slice` in that it defines a set of strides and an associated set of lengths. This set of lengths and strides allows a program to treat a single-dimensional ***valarray*** as a multidimensional array and to pull multi-dimensional slices from that array. Note that the `gslice` subset operator takes a `gslice` argument and returns a ***gslice_array*** in the same manner that the `slice` operation takes a ***slice*** and returns a ***slice_array***.

Here is a simple example that uses ***gslice***s to represent a three-dimensional array. In this example, the first three numbers form the top row of a two-dimensional array whose center and bottom rows are completed with the next six numbers. The next nine numbers represent the second or *middle* two-dimensional array, and the last nine the *back* two-dimensional array. Taken together they form a three- dimensional *cube*. See the *Class Reference* for an extended version of this example that uses additional ***gslice***s to pull ***slice***s along several different axes of the three-dimensional cube defined by this first slice.

©Copyright 1999 Rogue Wave Software, Inc.

```
using std::gslice;

int a[27] =   {0,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,9,2,3,4,5,6,7,8,9,10};
size_t lengths[2] = {3,3};
size_t strides[2] = {3,1};

valarray<int>  v(a,27);             // initial valarray
valarray<size_t> len(lengths,2);    // valarray of lengths
valarray<size_t> stride(strides,2); // valarray of strides
valarray<int> v2(0,9);              // valarray to hold the
                                    // generalized slice
v2 = (v[gslice(0,len,stride)]);     //  slice off the front  =
                                    // {0,1,2,3,4,5,6,7,8}
v2 = (v[gslice(9,len,stride)]);     // slice off the middle  =
                                    // {1,2,3,4,5,6,7,8,9}

                          Back two-dimensional array = 2  3  4
                                                       5  6  7
                                                       8  9  10
           Middle two-dimensional array = 1  2  3
                                          4  5  6
                                          7  8  9
 Front two-dimensional array = 0  1  2
                               3  4  5
                               6  7  8
```

Boolean Mask

The third operation is a boolean `mask`, which selects certain elements of a ***valarray*** based on whether or not the corresponding element of a boolean ***valarray*** is set to `true`. The size of the boolean ***valarray*** must equal the size of the subscripted ***valarray***. Like the ***slice*** example, the following code snippet also selects all the odd numbers in ***all***. Note that the operation actually returns a ***mask_array***.

```
int a[9] = {1,2,3,4,5,6,7,8,9};
valarray<int> all(a,9);               //all = {1,2,3,4,5,6,7,8,9}

bool b[9]= {true,false,true,false,true,false,true,false,true};
valarray<bool> vb(b,9);
valarray<int> odd2 = all[vb];         //odd2 = {1,3,5,7,9}
```

Indirect Operation

Finally, the `indirect` operation selects elements of a ***valarray*** dependent on whether or not a given index is present in a ***valarray*** of ***size_t***. For instance, if we use ***valarray<size_t>()*** as our subscripting argument, then we select `0` elements, returning a ***valarray*** or ***indirect_array*** (if the sub-scripted array is `const`) of `0` elements. On the other hand, if we use ***valarray<size_t>(3,1)***, we select only the third element, thus returning a ***valarray*** or indirect array of `1` element. Once again, the following example selects all odd-valued elements from ***all***:

```
int a[9] = {1,2,3,4,5,6,7,8,9};
valarray<int> all(a,9);               // all = {1,2,3,4,5,6,7,8,9}

size_t c[5]  = {0,2,4,6,8};
valarray<size_t> in(c,5);
valarray<int> oddity = all[in];       // oddity = {1,3,5,7}
```

Each of the auxiliary array classes, ***slice_array***, ***gslice_array***, ***mask_array***, and ***indirect_array***, has a set of arithmetic and logical computed assignment operators that when applied affect the selected elements in the original non-`const` array. In each case, these operators take a ***valarray*** as a parameter. Since the ***valarray*** itself contains conversion constructors for each of the auxiliary classes, these arithmetic and logical operators can be applied freely across ***valarray*** objects and auxiliary classes and between the auxiliary classes. Note, however, that reference semantics are lost in the conversion from auxiliary class to ***valarray***. A ***valarray*** never references data in another ***valarray***, reference-counting optimization notwithstanding.

Unary Operators

Class ***valarray*** provides four unary operations: `operator+`, `operator-`, `operator~`, and `operator!`. These operators are not available for types for which they are not defined. Each operator returns a new valarray with the result of applying the operation to each element of the original. For instance, the following negates all values in a ***valarray***.

©Copyright 1999 Rogue Wave Software, Inc.

```
float a[4] = {1.0, -2.3, -4.5, 9.0};
valarray<float> v(a,4);
valarray<float> neg = -v;          // neg = {-1.0, 2.3, 4.5, -9.0}
```

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Computed Assignment Operators

Class ***valarray*** offers two different versions of the following operators: `operator*=`, `operator/=`, `operator%=`, `operator+=`, `operator-=`, `operator^=`, `operator&=`, `operator|=`, `operator<<=`, and `operator>>=`. In each case, one version takes a ***valarray*** reference and the second takes a `T&`. The first version of each operation applies its operation to the corresponding elements of self and the ***valarray*** argument. The second version applies the operation to all elements of self using the `T&` argument. For example:

```
valarray<long> v1(1,3) ;                    // v1 = {1,1,1}
valarray<long> v2(2,3);                     // v2 = {2,2,2}
v1 += v2;                                   // v1 = {3,3,3}
v2 += 2;                                    // v2 = {4,4,4}
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Member Functions

The member functions of class ***valarray*** include the `size()` function, which returns the size of a ***valarray***, and the `sum()` function, which returns the sum of all elements in a ***valarray***. The functions `min()` and `max()` return, respectively, the minimum and maximum values in a ***valarray***.

Class ***valarray*** also contains two shift functions. Both functions shift elements a specified number of steps to the left, where the `0`th element is defined as the leftmost. The `shift()` function fills in at the right with `0`'s or `T()`'s. The `cshift()` function performs a circular shift or *rotation* so that the `i`th element becomes the element at location `self.length() - n - i`, where `i` runs from `0` to `n`, and `n` is the number of steps to shift. Both functions return a new ***valarray***, but the original is not changed. For example:

```
int a[5] = {1,2,3,4,5};
valarray<int> v(a,5);
valarray<int> v2 = v.shift(2);                  // v2 = {3,4,5,0,0}
v2 = v.cshift(2);                               // v2 = {3,4,5,1,2}
```

Class ***valarray*** provides two versions of the `apply()` function. These functions apply a user-specified function to each element in the array. Again, these return a new array with the result leaving the original array unaltered.

Finally, as we noted in the discussion on constructors, ***valarray*** has a resize function that allows a program to change the number of elements contained by that array. The original elements, if any, are lost in the process since the function re-initializes all elements in the newly-sized array.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Non-Member Functions

Binary Operators

Class ***valarray*** provides a large block of binary operators that allow a program to apply these operators to two ***valarray***s or to a ***valarray*** and a `T` value, where the `T` can occur on the left- or right-hand side of the expression. As you'd expect, these operators do not modify either of their arguments, but instead return a new ***valarray*** containing the result of the operation. Each version that takes two ***valarray*** arguments returns the result of applying the operation to the corresponding elements of the arrays. The versions with a `T&` argument return a ***valarray*** whose elements result from applying the operation to each element in the ***valarray*** argument along with the `T&` argument. See the ***valarray*** entry in the *Class Reference* for a complete list of binary operators.

Arithmetic and bitwise operators return a ***valarray*** of `T`. Logical operators return a ***valarray<bool>***, where each element represents the comparison of corresponding elements in two ***valarray***s or the comparison of each element in an array with a particular value. For example:

```
int a[3] = {1,2,3};
int b[3] = {0,1,4};
valarray<int> v1(a,3);
valarray<int> v2(b,3);
valarray<int> v3(0,3);
valarray<bool> v4(false,3);
v3 = v1 + v2;                           // v3 = {1,3,7}
v4 = v1 == v2;                          // v4 = {false,false,false}
v4 = v1 < v2;                           // v4 = {false,false,true}
```

Transcendental Functions

Class ***valarray*** also provides 20 transcendental functions, four of which are overloads. Each function returns a new ***valarray*** that contains elements resulting from the application of the transcendental function to each element in the ***valarray*** passed as an argument. See the ***valarray*** entry in the *Class Reference* for a complete list of transcendental functions.

The following example squares each of the elements of an array:

```
using std::pow;

float a[3] = {2.0, 4.0, 16.0};
valarray<float> v(a,3);               // v = {2.0, 4.0, 16.0}
valarray<float> v2(0.0,3);            // v2 = {0.0, 0.0, 0.0}

v2 = pow(v,2.0);                      // v2 = {4.0, 16.0, 256.0 }
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

Topic Index

Click on one of the letters below to jump immediately to that section of the index. If you get no response, that letter has no entries.

A B C D E F G H I J K L M N O P

Q R S T U V W X Y Z

a

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

b

©Copyright 1999 Rogue Wave Software, Inc.

c

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

d

©Copyright 1999 Rogue Wave Software, Inc.

e

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

f

©Copyright 1999 Rogue Wave Software, Inc.

g

©Copyright 1999 Rogue Wave Software, Inc.

h

i

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

k

l

©Copyright 1999 Rogue Wave Software, Inc.

m

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

n

o

p

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

q

r

©Copyright 1999 Rogue Wave Software, Inc.

s

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

t

©Copyright 1999 Rogue Wave Software, Inc.

u

v

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.
Documentation Tips

If you are accessing this for the first time, please read the licensing statement.

Standard C++ Library General User's Guide - OEM Edition

Welcome to the Standard C++ Library General User's Guide. The Standard C++ Library documentation includes these companion volumes:

Locales and Iostreams User's Guide
Class Reference

In this document, you can view a

Comprehensive Table of Contents

showing all chapters, first- and second-level headings, or click on one of the chapter names below to go directly to that chapter. Each chapter begins with a chapter-level table of contents.

There is also a topic index.

Chapters

©Copyright 1999 Rogue Wave Software, Inc.

Part IV: Algorithms

Part V: Special Techniques

Part VI: Special Classes

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

Endnotes

1

We apologize to international readers for this obviously North American example.

Return

2

A more robust program would also check to see if the stack were empty before attempting to perform the pop() operation.

Return

3

As noted in a previous footnote, support for initializing containers using a pair of iterators requires a feature that is not yet widely supported by compilers, so it may not yet be available on your system.

Return

4

Details of the algorithms used in manipulating heaps will not be discussed here, but such information is readily available in almost any textbook on data structures.

Return

5

In theory one could store references instead of pointers, but the Standard C++ Library containers cannot hold references.

Return

6

We describe the priority queue as a structure for quickly discovering the smallest element in a sequence. If, instead, your problem requires the discovery of the largest element, there are various possibilities. One is to supply the inverse operator as either a template argument or the optional comparison function argument to the constructor. If you are defining the comparison argument as a function, as in the example problem, another solution is to simply invert the comparison test.

Return

7

Remember, the ability to initialize a container using a pair of iterators requires the ability to declare a template member function using template arguments independent of those used to declare the container. At present not all compilers support this feature.

Return

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.
Documentation Tips

If you are accessing this for the first time, please read the licensing statement.

Standard C++ Library Locales and Iostreams User's Guide - OEM Edition

Welcome to the Standard C++ Library Locales and Iostreams User's Guide. The Standard C++ Library documentation includes these companion volumes:

General User's Guide
Class Reference

In this document, you can view a

Comprehensive Table of Contents

showing all chapters, first- and second-level headings, or click on one of the chapter names below to go directly to that chapter. Each chapter begins with a chapter-level table of contents.

There is also a topic index.

Chapters

Part I: Introduction

Chapter 1: Overview

Part II: Locales

Chapter 2: Internationalization and Localization

Chapter 3: The C and C++ Locales

Chapter 4: Facets

Chapter 5: Building Your Own Facet Class

Part III: Iostreams

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Part I: Introduction

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 1: Overview

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Welcome

Thank you for choosing this Rogue Wave implementation of the Standard C++ Library. For a complete introduction to this product and the accompanying documentation, please see the *Standard C++ Library User's Guide*.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

About Locales and Iostreams

This manual deals with the Standard C++ Library treatment of locales and iostreams. We include these topics in a separate manual for the convenience of developers who would like to use them independently of the full library.

Both locales and iostreams existed prior to the recent standardization of the C++ language and library. Many developers are already comfortable with using iostreams, and have integrated them into their existing code. They have no reason to associate iostreams with the new standard, and in any case, this new implementation of iostreams is not so very different from the old.

Locales have been around a long time, too, but they change significantly in the Standard C++ Library. The old locale is part of the C library and consists of a group of C function calls, like `setlocale`. The new locale performs the same operations, but consists of a set of C++ classes and functions, so the interface is completely changed. Iostreams depend on locales, so they are discussed together in this manual.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Documentation Overview

You are reading the manual called *The Standard C++ Library Locales and Iostreams*. This manual is part of a complete set of documentation designed to help you effectively use this Rogue Wave implementation of the Standard C++ Library. The complete documentation for this product is summarized in Table 1:

Table 1 -- Documentation for Rogue Wave's Standard C++ Library

Documentation format	Description
The Standard C++ Library User's Guide	Covers all the essentials, including iterators, containers, algorithms, allocators, and error handling, and explanations of the tutorials. Online version ships with product; hard copy available for purchase.
The Standard C++ Library Locales and Iostreams	Covers the familiar iostreams and the newer implementation of locales; basics of internationalization. Online version ships with product; hard copy available for purchase.
The Standard C++ Library Class Reference	An alphabetical listing of all classes, algorithms, and function objects. Online version ships with product; hard copy available for purchase.
Readme files	Information on specific compilers and operating systems, how to use shared libraries and DLLs, solutions to common problems, and late-breaking product news.
Tutorials	Examples you can use to learn about the Standard C++ Library and to write your own applications.
man pages	For Unix platforms only.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Overview of This Manual

Organization

This manual is divided into three parts:

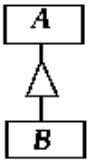
- **Part I** is an introduction to the manual. It explains the organization of the manual, and its role in the total documentation for the Rogue Wave implementation of the Standard C++ Library.
- **Part II** explains *locales*. After a general introduction to internationalization, it explains the C facilities for internationalizing software, and how the C locale differs from the C++ locale. It defines a C++ locale and a facet, and tells how locales are created, composed, used, and replaced. It includes a complex example of a user-defined facet, which demonstrates how facets can be built and used in conjunction with iostreams.
- **Part III** explains the C++ stream input and output facilities commonly called *iostreams*. Beginning chapters cover the iostreams facility, how it works, and how it should be used; also, the iostreams architecture, its components, and class hierarchy. Middle chapters cover the basic operation of iostreams, and both simple and advanced techniques for extending the iostreams framework. Final chapters describe the main differences between the Standard C++ Library iostreams, the traditional iostreams, and Rogue Wave's implementation of iostreams in the Standard C++ Library.

Please note that this manual is written as a part of the *Standard C++ Library User's Guide*, and so focuses on general principles and examples of using the product. More specific information on the classes and functions is contained in the *Class Reference*.

Conventions

This manual uses some distinctive terms and conventions. The Standard C++ Library consists of many class and function templates, so shorter forms for these templates are common in this manual, as in the *Standard C++ Library User's Guide*. For example, in the iostreams part of the documentation, `fstream` stands for `template <class charT, class traits> class basic_fstream`. A slightly more succinct notation for a class template is also frequently used: `basic_fstream <charT, traits>`.

Formal template declarations are used in the hierarchy diagrams, while shorter forms are used elsewhere. In the hierarchy diagrams, the notation:



indicates that class ***B*** inherits from class ***A***.

File stream stands for the abstract notion of the file stream class template; `badbit` stands for the state flag `ios_base::badbit`.

The term *algorithm* indicates functions in the generic algorithms portion of the Standard C++ Library, so as to avoid confusion with member functions, argument functions, and user-defined functions.

An empty pair of parentheses `()` follows function names and algorithm names, so as to avoid emphasizing their arguments. An underline character _ is used as a separator in both class names and function names.

Special fonts set off class names, code samples, and special meanings, as shown in Table 2:.

Table 2 -- Typographic conventions

©Copyright 1999 Rogue Wave Software, Inc.

Convention	Purpose	Example
`Courier`	Code, examples, function names, file names, directory names, operating system commands	`return result;`
italic	Emphasis. New terms. Titles.	operating system *family* special *ending* iterator *User's Guide*
bold	Emphasis. Commands from an interface. Rogue Wave product names.	do this **before** that the **OK** button a **Standard C++ Library** file
bold italic	Class names.	***priority_queue***

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Index

Table of Contents

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Top Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top

©Copyright 1998, Rogue Wave Software, Inc.

Use your browser's **Back** button to return to where you were.

Feedback on the Documentation

Write to us at onlinedocs@roguewave.com

Please use this only to:

1. Report errors in the documentation.
2. Make suggestions on how to improve the documentation.
3. Tell us how much you like the documentation.

Support Issues

The Rogue Wave support Web site at support.roguewave.com offers several ways to contact Rogue Wave technical support. The Web site includes an ever-expanding, searchable KnowledgeBase that can answer many support questions immediately.

You may also write to us at support@roguewave.com.

Telephone support is available at (303) 545 3205.

Top

©Copyright 1999 Rogue Wave Software, Inc.

Top

©Copyright 1998, Rogue Wave Software, Inc.
Send mail to report errors or comment on the documentation.
Banner | Buttons | Comments | User's Guide Features | Reference Guide Features | Want Books

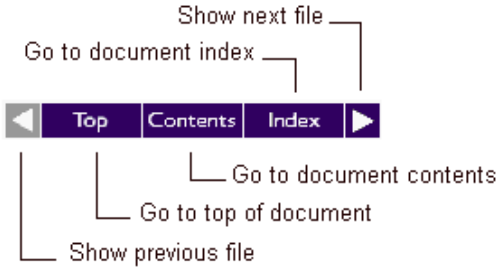
Tips on Using Rogue Wave Documentation

The banner

You can always click on the **Rogue Wave Online Documentation** banner to go to the master page for the Standard C++ Library User's Guide.

The buttons

The buttons do the following:



Use the **Show previous/next file** buttons to move sequentially through a document. A grayed-out button is inactive.

Contacting Rogue Wave

Don't like something? Or, more charitably, would you like make a suggestion or point out an error? Use the **Contact ...** link at the top of the first page and the bottom of all subsequent pages to view a contact page. This page provides a mailto link to the documentation team at Rogue Wave.

The contact page also provides ways to contact technical support. Please use these contact points and **not** the documentation mailto link for technical issues.

User's guide navigation features

The user's guides (also build guides, platform guides, and installation guides) have these navigation features:

- A chapter-name table of contents on the opening page, providing links to the manual's major topics.
- A section-name table of contents at the beginning of each chapter, providing links to the chapter's detailed topics.
- A comprehensive table of contents in case you want to see everything. The **Contents** button leads here.
- Usually, a topic index based on the index markup in the print document. The links are based on the name of the section to which the link leads to assist you in deciding which link to follow.

Reference guide navigation features

The reference guides have these navigation features:

- Links to the major sections on the opening page, with the final link being to the section of class descriptions.
- A hypertext list of class descriptions. The **Contents** button leads here.
- A section-name table of contents at the beginning of each class description, providing links to the categories of descriptions.
- A per-class method/data type hypertext index at the beginning of each class description.
- A comprehensive hypertext index of methods and data types. The links are based on the name of the class description to which the link leads, so for methods or data types that appear in multiple classes you can easily

©Copyright 1999 Rogue Wave Software, Inc.

select the link to class in which you are interested.

want books

If you prefer books, these may be purchased from Rogue Wave. Contact Rogue Wave by:

Telephone:(303) 473-9118 or (800) 487-3217

Email: sales@roguewave.com

WWW: http://www.roguewave.com

Top

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Part II: Locales

Chapters in This Part

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Chapter 2: Internationalization and Localization

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Defining the Terms

Computer users all over the world prefer to interact with their systems using their own local languages and cultural conventions. As a developer aiming for high international acceptance of your products, you need to provide users the flexibility for modifying input and output conventions to comply with local requirements, such as different currency and numeric representations. You must also provide the capability for translating interfaces and messages without necessitating many different language versions of your software.

Two processes that enhance software for worldwide use are *internationalization* and *localization*. *Internationalization* is the process of building into software the potential for worldwide use. It is the result of efforts by programmers and software designers during software development.

Internationalization requires that developers consciously design and implement software for adaptation to various languages and cultural conventions, and avoid hard-coding elements that can be localized, like screen positions and file names. For example, developers should never embed in their code any messages, prompts, or other kind of displayed text, but rather store the messages externally, so they can be translated without requiring the program to be recompiled. A developer of internationalized software should never assume specific conventions for formatting numeric or monetary values, or for displaying date and time.

Localization is the process of actually adapting internationalized software to the needs of users in a particular geographical or cultural area. It includes translation of messages by software translators. It requires the creation and availability of appropriate tables containing relevant local data for use in a given system. This typically is the function of system administrators and operating system vendors, who build facilities for these functions into the program execution environment. Users of internationalized software are involved in the process as well, when they select the local conventions they prefer from the set of conventions available in their environment.

The Standard C++ Library offers a number of classes that support internationalization of your programs. We describe them in detail in this chapter. Before we do, however, we would like to survey some of the cultural conventions that impact software internationalization, and are supported by the programming languages C and C++ and their respective standard libraries. Of course, there are many issues outside our list that need to be addressed, like orientation, sizing and positioning of screen displays, vertical writing and printing, selection of font tables, handling international keyboards, and so on. But let us begin here.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Localizing Cultural Conventions

The need for localizing software arises from differences in cultural conventions. These differences involve: language itself; representation of numbers and currency; display of time and date; and ordering and classification of characters and strings.

Language

Of course, *language* itself varies from country to country, and even within a country. Your program may require output messages in English, Deutsche, Français, Italiano, or any number of languages commonly used in the world today.

Languages may also differ in the *alphabet* they use. Examples of different languages with their respective alphabets are given below:

American English:	a-z, A-Z, and punctuation
German:	a-z, A-Z, punctuation, and äöü ÄöÜ ß
Greek:	α–ω, Α–Ω, and punctuation

Numbers

The representation of *numbers* depends on local customs, which vary from country to country. For example, consider the *radix character*, the symbol used to separate the integer portion of a number from the fractional portion. In American English, this character is a period; in much of Europe, it is a comma. Conversely, the thousands separator that separates numbers larger than three digits is a comma in American English, and a period in much of Europe.

The convention for grouping digits also varies. In American English, digits are grouped by threes, but there are many other possibilities. In the example below, the same number is written as it would be locally in three different countries:

1,000,000.55 US
1.000.000,55 Germany
10,00,000.55 Nepal

Currency

We are all aware that countries use different currencies. However, not everyone realizes the many different ways we can represent units of currency. For example, the symbol for a currency can vary. Here are two different ways of representing the same amount in US dollars:

$24.99	US
USD 24.99	International currency symbol for the US

The placement of the currency symbol varies for different currencies, too, appearing before, after, or even within the numeric value:

¥ 155	Japan
13,50 DM	Germany
£14 19s. 6d.	England before decimalization

The format of negative currency values differs:

öS 1,1	-öS 1,1	Austria
1,1 DM	-1,1 DM	Germany

©Copyright 1999 Rogue Wave Software, Inc.

SFr. 1.1 SFr.-1.1 Switzerland
HK$1.1 (HK$1.1) Hong Kong

Time and Date

Local conventions also determine how *time* and *date* are displayed. Some countries use a 24-hour clock; others use a 12-hour clock. Names and abbreviations for days of the week and months of the year vary by language.

Customs dictate the ordering of the year, month, and day, as well as the separating delimiters for their numeric representation. To designate years, some regions use seasonal, astronomical, or historical criteria, instead of the Western Gregorian calendar system. For example, the official Japanese calendar is based on the year of reign of the current Emperor.

The following example shows short and long representations of the same date in different countries:

10/29/96	Tuesday, October 29, 1996	US
1996. 10. 29.	1996. október 29.	Hungary
29/10/96	martedì 29 ottobre 1996	Italy
29/10/1996	Τριτη, 29 Οκτωβριου 1996	Greece
29.10.96	Dienstag, 29. Oktober 1996	Germany

The following example shows different representations of the same time:

4:55 pm US time
16:55 Uhr German time

And the following example shows different representations of the same time:

11:45:15 Digital representation, US
11:45:15 μμ Digital representation, Greece

Ordering

Languages may vary regarding *collating sequence*; that is, their rules for ordering or sorting characters or strings. The following example compares the same list of words ordered alphabetically by different collating sequences:

Sorted by ASCII rules:	***Sorted by German rules:***
Airplane	Airplane
Zebra	ähnlich
bird	bird
car	car
ähnlich	Zebra

The ASCII collation orders elements according to the numeric value of bytes, which does not meet the requirements of English language dictionary sorting. This is because lexicographical order sorts a after A and before B, whereas ASCII-based order sorts a after the entire set of uppercase letters.

The German alphabet sorts ä before b, whereas the ASCII order sorts an umlaut after all other letters.

In addition to specifying the ordering of individual characters, some languages specify that certain groups of characters should be clustered and treated as a single character. The following example shows the difference this can make in an ordering:

Sorted by ASCII rules:	***Sorted by Spanish rules:***
chaleco	cuna
cuna	chaleco
día	día
llava	loro

©Copyright 1999 Rogue Wave Software, Inc.

loro	llava
maíz	maíz

The word llava is sorted after loro and before maíz, because in Spanish ll is a digraph, i.e., it is treated as a single character that is sorted after l and before m. Similarly, the digraph ch in Spanish is treated as a single character to be sorted after c, but before d. Two characters that are paired and treated as a single character are referred to as a *two-to-one character code pair*.

In other cases, one character is treated as if it were actually two characters. The German single character ß, called the *sharp s*, is treated as ss. This treatment makes a difference in the ordering, as shown in the example below:

Sorted by ASCII rules:	***Sorted by German rules:***
Rosselenker	Rosselenker
Rostbratwurst	Roßhaar
Roßhaar	Rostbratwurst

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Character Encodings for Localizing Alphabets

We know that different languages can have different alphabets. The first step in localizing an alphabet is to find a way to represent, or *encode*, all its characters. In general, alphabets may have different *character encodings*.

The *7-bit ASCII codeset* is the traditional code on UNIX systems.

The *8-bit codesets* permit the processing of many Eastern and Western European, Middle Eastern, and Asian Languages. Some are strictly extensions of the 7-bit ASCII codeset; these include the 7-bit ASCII codes and additionally support 128-character codes beyond those of ASCII. Such extensions meet the needs of Western European users. To support languages that have completely different alphabets, such as Arabic and Greek, larger 8-bit codesets have been designed.

Multibyte character codes are required for alphabets of more than 256 characters, such as kanji, which consists of Japanese ideographs based on Chinese characters. Kanji has tens of thousands of characters, each of which is represented by two bytes. To ensure backward compatibility with ASCII, a multibyte codeset is a superset of the ASCII codeset and consists of a mixture of one- and two-byte characters.

For such languages, several encoding schemes have been defined. These encoding schemes provide a set of rules for parsing a byte stream into a group of coded characters.

Multibyte Encodings

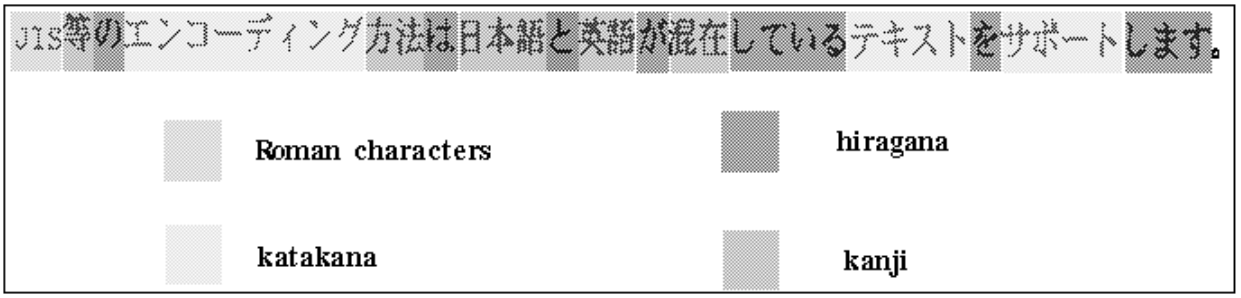
Handling multibyte character encodings is a challenging task. It involves parsing multibyte character sequences, and in many cases requires conversions between multibyte characters and wide characters.

Understanding multibyte encoding schemes is easier when explained by means of a typical example. One of the earliest and probably biggest markets for multibyte character support is in Japan. Therefore, the following examples are based on encoding schemes for Japanese text processing.

In Japan, a single text message can be composed of characters from four different writing systems. *Kanji* has tens of thousands of characters, which are represented by pictures. *Hiragana* and *katakana* are syllabaries, each containing about 80 sounds, which are also represented as ideographs. The *Roman* characters include some 95 letters, digits, and punctuation marks.

Figure 1 gives an example of an encoded Japanese sentence composed of these four writing systems:

Figure 1 -- A Japanese sentence mixing four writing systems



The sentence means: "Encoding methods such as JIS can support texts that mix Japanese and English."

A number of Japanese character sets are common:

JIS C 6226-1978 JIS X 0208-1983
JIS X 0208-1990 JIS X 0212-1990
JIS-ROMAN ASCII

©Copyright 1999 Rogue Wave Software, Inc.

There is no universally recognized multibyte encoding scheme for Japanese. Instead, we deal with the three common multibyte encoding schemes defined below:

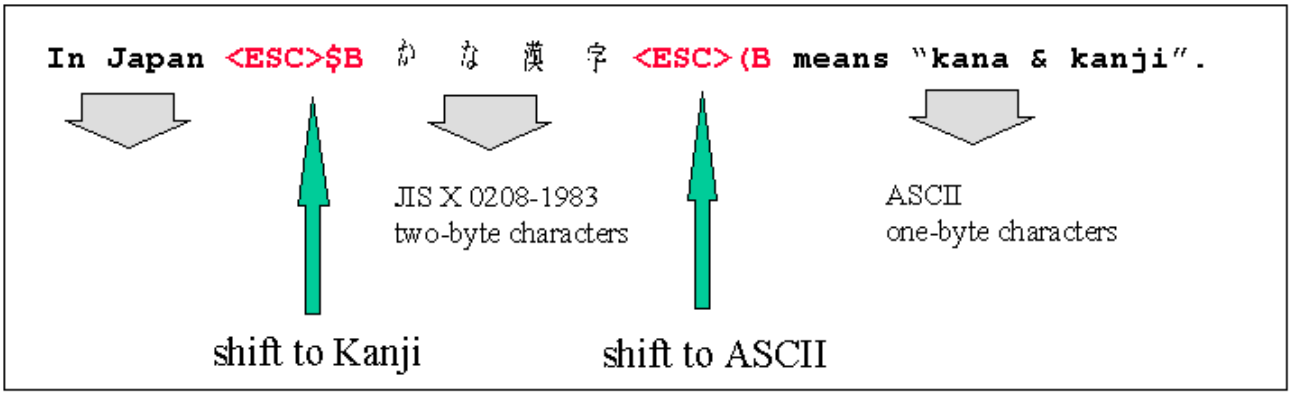
JIS (Japanese Industrial Standard)
Shift-JIS
EUC (Extended UNIX Code)

JIS Encoding

The *JIS*, or *Japanese Industrial Standard*, supports a number of standard Japanese character sets, some requiring one byte, others two. Escape sequences are required to shift between one- and two-byte modes.

Escape sequences, also referred to as *shift sequences*, are sequences of *control characters*. Control characters do not belong to any of the alphabets. They are artificial characters that do not have a visual representation. However, they are part of the encoding scheme, where they serve as separators between different character sets, and indicate a switch in the way a character sequence is interpreted. The use of the shift sequence is demonstrated in Figure 2.

Figure 2 -- An example of a Japanese text encoded in JIS



For encoding schemes containing shift sequences, like JIS, it is necessary to maintain a *shift state* while parsing a character sequence. In the example above, we are in some initial shift state at the start of the sequence. Here it is ASCII. Therefore, characters are assumed to be one-byte ASCII codes until the shift sequence <ESC>$B is seen. This switches us to two-byte mode, as defined by JIS X 0208-1983. The shift sequence <ESC>(B then switches us back to ASCII mode.

Encoding schemes that use shift state are not very efficient for internal storage or processing. Sometimes shift sequences require up to six bytes. Frequent switching between character sets in a file of strings could cause the number of bytes used in shift sequences to exceed the number of bytes used to represent the actual data!

Encodings containing shift sequences are used primarily as an external code, which allows information interchange between a program and the outside world.

Shift-JIS Encoding

Despite its name, Shift-JIS has nothing to do with shift sequences and states. In this encoding scheme, each byte is inspected to see if it is a one-byte character or the first byte of a two-byte character. This is determined by reserving a set of byte values for certain purposes. For example:

- Any byte having a value in the range 0x21-7E is assumed to be a one-byte ASCII/JIS Roman character.
- Any byte having a value in the range 0xA1-DF is assumed to be a one-byte half-width katakana character.
- Any byte having a value in the range 0x81-9F or 0xE0-EF is assumed to be the first byte of a two-byte character from the set JIS X 0208-1990. The second byte must have a value in the range 0x40-7E or 0x80-FC.

While this encoding is more compact than JIS, it cannot represent as many characters as JIS. In fact, Shift-JIS cannot represent any characters in the supplemental character set JIS X 0212-1990, which contains more than 6,000 characters.

EUC Encoding

©Copyright 1999 Rogue Wave Software, Inc.

Extended Unix Code (EUC) is not peculiar to Japanese encoding. It was developed as a method for handling multiple character sets, Japanese or otherwise, within a single text stream.

The EUC encoding is much more extensible than Shift-JIS since it allows for characters containing more than two bytes. The encoding scheme used for Japanese characters is as follows:

- Any byte having a value in the range 0x21-7E is assumed to be a one-byte ASCII/JIS Roman character.
- Any byte having a value in the range 0xA1-FE is assumed to be the first byte of a two-byte character from the set JIS X0208-1990. The second byte must also have a value in that range.
- Any byte having a value in the range 0x8E is assumed to be followed by a second byte with a value in the range 0xA1-DF, which represents a half-width katakana character.
- Any byte having the value 0x8F is assumed to be followed by two more bytes with values in the range 0xA1-FE, which together represent a character from the set JIS X0212-1990.

The last two cases involve a prefix byte with values 0x8E and 0x8F, respectively. These bytes are somewhat like shift sequences in that they introduce a change in subsequent byte interpretation. However, unlike the shift sequences in JIS which introduce a sequence, these prefix bytes must precede *every* multibyte character, not just the first in a sequence. For this reason, each multibyte character encoded in this manner stands alone and EUC is not considered to involve shift states.

Uses of the Three Multibyte Encodings

The three multibyte encodings just described are typically used in separate areas:

- JIS is the primary encoding method used for electronic transmission such as email because it uses only 7 bits of each byte. This is required because some network paths strip the eighth bit from characters. Escape sequences are used to switch between one- and two-byte modes, as well as between different character sets.
- Shift-JIS was invented by Microsoft and is used on MS-DOS-based machines. Each byte is inspected to see if it is a one-byte character or the first byte of a two-byte character. Shift-JIS does not support as many characters as JIS and EUC do.
- EUC encoding is implemented as the internal code for most UNIX-based platforms. It allows for characters containing more than two bytes, and is much more extensible that Shift-JIS. EUC is a general method for handling multiple character sets. It is not peculiar to Japanese encoding.

Wide Characters

Multibyte encoding provides an efficient way to move characters around outside programs, and between programs and the outside world. Once inside a program, however, it is easier and more efficient to deal with characters that have the same size and format. We call these *wide characters*.

Here is an example that illustrates how wide characters make text processing inside a program easier. Consider a filename string containing a directory path with adjacent names separated by a slash, like /CC/include/locale.h. To find the actual filename in a single-byte character string, we can start at the back of the string. When we find the first separator, we know where the filename starts. If the string contains multibyte characters, we scan from the front so we don't inspect bytes out of context. If the string contains wide characters, however, we can treat it like a single-byte character and scan from the back.

Conceptually, you can think of wide character sets as being extended ASCII or EBCDIC; each unique character is assigned a distinct value. Since they are used as the counterpart to a multibyte encoding, wide character sets must allow representation of all characters that can be represented in a multibyte encoding as wide characters. As multibyte encodings support thousands of characters, wide characters are usually larger that one byte-typically two or four bytes. All characters in a wide character set are of equal size. The size of a wide character is not universally fixed, although this depends on the particular wide character set.

There are many wide character standards, including those shown below:

ISO 10646.UCS-2	16-bit characters
ISO 10646.UCS-4	32-bit characters
Unicode	16-bit characters

©Copyright 1999 Rogue Wave Software, Inc.

The programming language C++ supports wide characters; their native type in C++ is called wchar_t. The syntax for wide character constants and wide character strings is similar to that for ordinary, tiny character constants and strings:

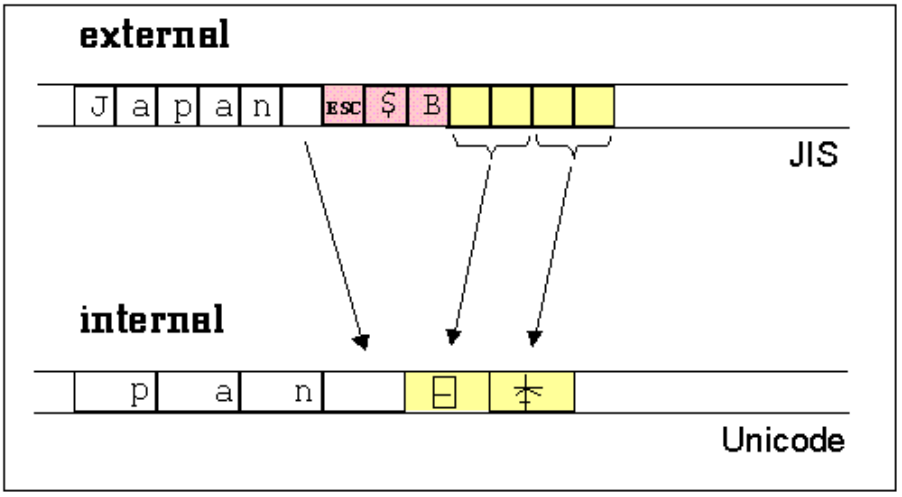
`L'a'` is a wide character constant, and

```
L"abc"6SAMP> is a wide character string.
```

Conversion between Multibyte and Wide Characters

Since wide characters are usually used for internal representation of characters in a program, and multibyte encodings are used for external representation, converting multibytes to wide characters is a common task during input/output operations. Input to and output from files is a typical example. The file usually contain multibyte characters. When you read such a file, you convert these multibyte characters into wide characters that you store in an internal wide character buffer for further processing. When you write to a multibyte file, you have to convert the wide characters held internally into multibytes for storage on a external file. Figure 3 demonstrates how this conversion during file input is done:

Figure 3 -- Conversion from a multibyte to a wide character encoding



The conversion from a multibyte sequence into a wide character sequence requires expansion of one-byte characters into two- or four-byte wide characters. Escape sequences are eliminated. Multibytes that consist of two or more bytes are translated into their wide character equivalents.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Summary

In this section, we discussed a variety of issues involved in developing software for worldwide use. For all of these areas in which cultural conventions differ from one region to another, the ***Standard C++ Library*** provides services that enable you to easily internationalize your C++ programs. These services include:

- Formatting and parsing of numbers, currency unit, dates, and time
- Handling different alphabets, their character classification, and collation sequences
- Converting codesets, including multibyte to wide character conversion
- Handling messages in different languages.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 3: The C and C++ Locales

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The C Locale

All the culture and language dependencies discussed in the previous section need to be represented in an operating system. This information is usually represented in a kind of language table, called a *locale*.

The X/Open consortium has standardized a variety of services for Native Language Support (NLS*)* in the programming language C. This standard is commonly known as XPG4. Internationalization services as well as localization support are included in X/Open's *Native Language Support*. The description below is based on this standard.

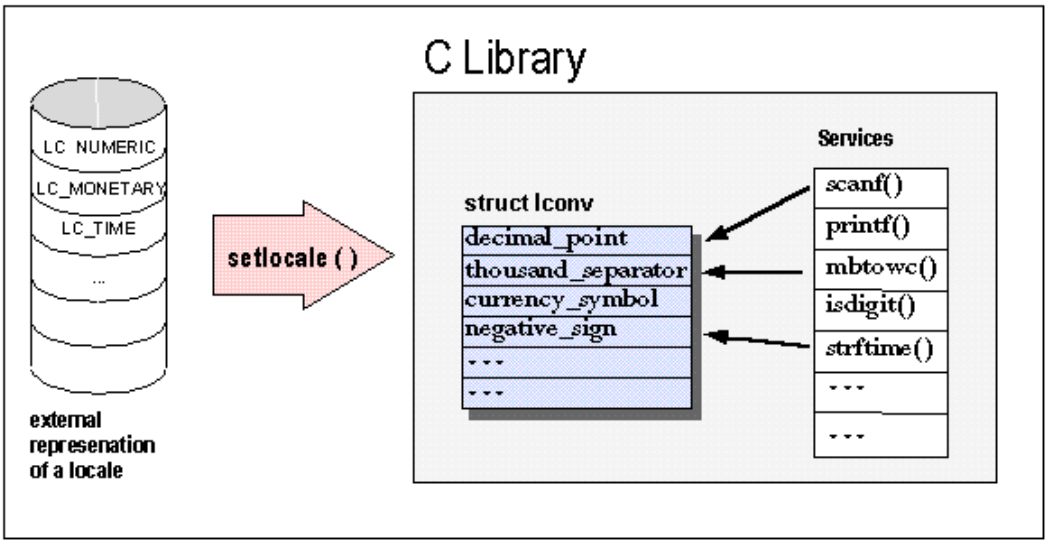
According to XPG4, the C locale is composed of several *categories*, which are given in Table 3.

Table 3 -- Categories of the C locale

Category	Content
`LC_NUMERIC`	Rules and symbols for numbers
`LC_TIME`	Values for date and time information
`LC_MONETARY`	Rules and symbols for monetary information
`LC_CTYPE`	Character classification and case conversion
`LC_COLLATE`	Collation sequence
`LC_MESSAGES`	Formats and values of messages

The external representation of a C locale is usually as a *file* in UNIX. Other operating systems may use other representations. The external representation is transformed into an internal memory representation by calling the function setlocale(), as shown in Figure 4:

Figure 4 -- Transformation of a C locale from external to internal representation



Inside a program, the C locale is represented by one or more global data structures. The C library provides a set of functions that use information from those global data structures to adapt their behavior to local conventions. Examples of these functions and the information they cover are listed in Table 4:

Table 4 -- C locale functions and the information they cover

C Locale Function	Information Covered
setlocale(), ...	Locale initialization and language information
isalpha(), isupper(), isdigit(), ...	Character classification

©Copyright 1999 Rogue Wave Software, Inc.

strftime(), ...	Date and time functions
strfmon()	Monetary functions
printf(), scanf(), ...	Number parsing and formatting
strcoll(), wcscoll(), ...	String collation
mblen(), mbtowc(), wctomb(), ...	Multibyte functions
catopen(), catgets(), catclose()	Message retrieval

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

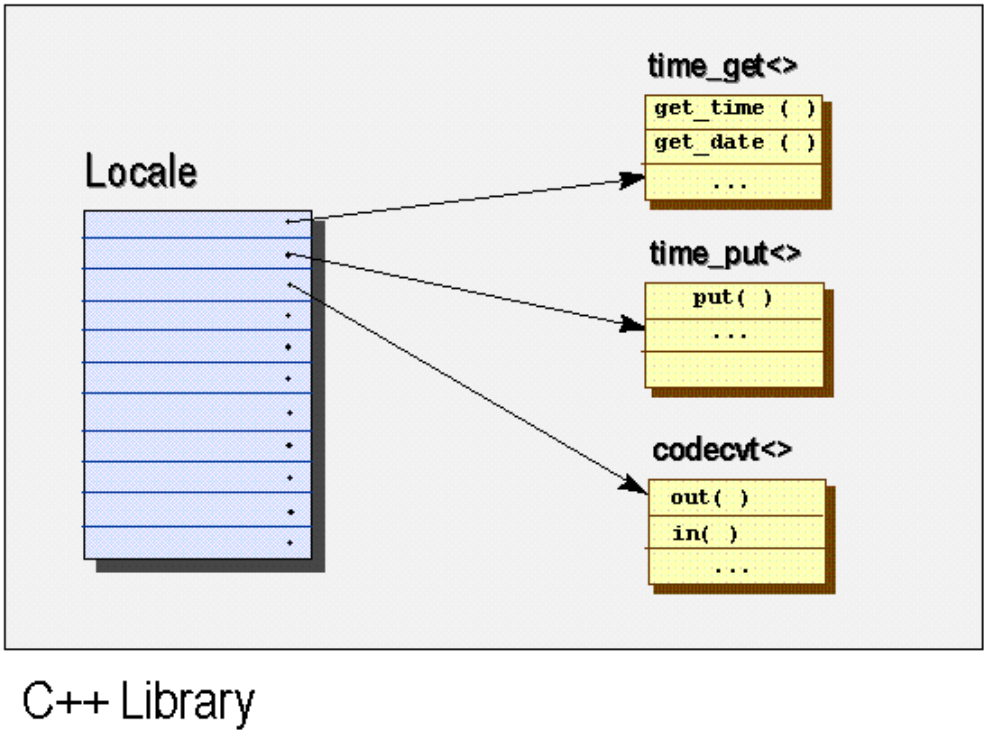
The C++ Locales

In C++, a locale is a class called locale provided by the Standard C++ Library. The C++ class locale differs from the C locale because it is more than a language table, or data representation of the various culture and language dependencies. It also includes the internationalization services, which in C are global functions.

In C++, internationalization semantics are broken out into separate classes called *facets*. Each facet handles a set of internationalization services; for example, the formatting of monetary values. Facets may also represent a set of culture and language dependencies, such as the rules and symbols for monetary information.

Each locale object maintains a set of facet objects. In fact, you can think of a C++ locale as a container of facets, as illustrated in Figure 5 below:

Figure 5 -- A C++ locale is a container of facets



Facets

Facet classes encapsulate data that represents a set of culture and language dependencies, and offer a set of related internationalization services. Facet classes are very flexible. They can contain just about any internationalization service you can invent. The ***Standard C++ Library*** offers a number of predefined standard facets, which provide services similar to those contained in the C library. However, you could bundle additional internationalization services into a new facet class, or purchase a facet library.

The Standard Facets

As listed in Table 1, the C locale is composed of six categories of locale-dependent information: LC_NUMERIC (rules and symbols for numbers), LC_TIME (values for date and time information), LC_MONETARY (rules and symbols for monetary information), LC_CTYPE (character classification and conversion), LC_COLLATE (collation sequence), and LC_MESSAGES (formats and values of messages).

Similarly, there are six groups of standard facet classes. A detailed description of these facets is contained in the *Class Reference*, but a brief overview is given below. Note that an abbreviation like num_get<charT,InputIterator> means that

©Copyright 1999 Rogue Wave Software, Inc.

num_get is a class template taking two template arguments, a character type, and an input iterator type. The groups of the standard facets are:

- **Numeric.** The facet classes num_get<charT,InputIterator> and num_put<charT,OutputIterator> handle numeric formatting and parsing. The facet classes provide get() and put() member functions for values of type long, double, etc.

  The facet class numpunct<charT> specifies numeric punctuation. It provides functions like decimal_point(), thousands_sep(), etc.

- **Time.** The facet classes time_get<charT,InputIterator> and time_put<charT,OutputIterator> handle date and time formatting and parsing. They provide functions like get_time(), get_date(), get_weekday(), etc.

- **Monetary.** The facet classes money_get<charT,InputIterator> and money_put<charT,OutputIterator> handle formatting and parsing of monetary values. They provide get() and put() member functions that parse or produce a sequence of digits, representing a count of the smallest unit of the currency. For example, the sequence $1,056.23 in a common US locale would yield 105623 units, or the character sequence "105623".

  The facet class moneypunct<charT,bool> handles monetary punctuation like the facet numpunct<charT> handles numeric punctuation. It comes with functions like curr_symbol(), etc.

- **Ctype.** The facet class ctype<charT> encapsulates the Standard C++ Library ctype features for character classification, like tolower(), toupper(), isspace(), isprint(), etc.

  The facet class codecvt<internT,externT,stateT> is used when converting from one encoding scheme to another, such as from the multibyte encoding JIS to the wide-character encoding Unicode. The main member functions are `in()` and `out()`. There is a template specialization <wchar_t,char,mbstate_t> for multibyte to wide character conversions.

- **Collate.** The facet class collate<charT> provides features for string collation, including a compare() function used for string comparison.

- **Messages.** The facet class messages<charT> implements the X/Open message retrieval. It provides facilities to access message catalogues via open() and close(catalog), and to retrieve messages via get(..., int msgid,...).

The names of the standard facets obey certain naming rules. The get facet classes, like num_get and time_get, handle parsing. The put facet classes handle formatting. The punct facet classes, like numpunct and moneypunct, represent rules and symbols.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Differences between the C Locale and the C++ Locales

As we have seen so far, the C locale and the C++ locale offer similar services. However, the semantics of the C++ locale are different from the semantics of the C locale:

- The *Standard C locale* is a global resource: there is only one locale for the entire application. This makes it hard to build an application that has to handle several locales at a time.
- The *Standard C++ locale* is a class. Numerous instances of class locale can be created at will, so you can have as many locale objects as you need.

To explore this difference in further detail, let us see how locales are typically used.

Common Uses of the C locale

The C locale is commonly used as a default locale, a native locale, or in multiple locale applications.

Default locale. As a developer, you may never require internationalization features, and thus you may never call `setlocale()`. If you can safely assume that users of your applications are accommodated by the classic US English ASCII behavior, you have no need for localization. Without even knowing it, you always use the default locale, which is the US English ASCII locale.

Native locale. If you do plan on localizing your program, the appropriate strategy may be to retrieve the native locale once at the beginning of your program, and never, ever change this setting again. This way your application adapts itself to one particular locale, and uses this throughout its entire runtime. Users of such applications can explicitly set their favorite locale before starting the application. On Unix systems, they do this by setting environment variables such as `LANG`; other operating systems may use other methods.

In your program, you can specify that you want to use the user's preferred native locale by calling `setlocale("")` at startup, passing an empty string as the locale name. The empty string tells `setlocale` to use the locale specified by the user in the environment.

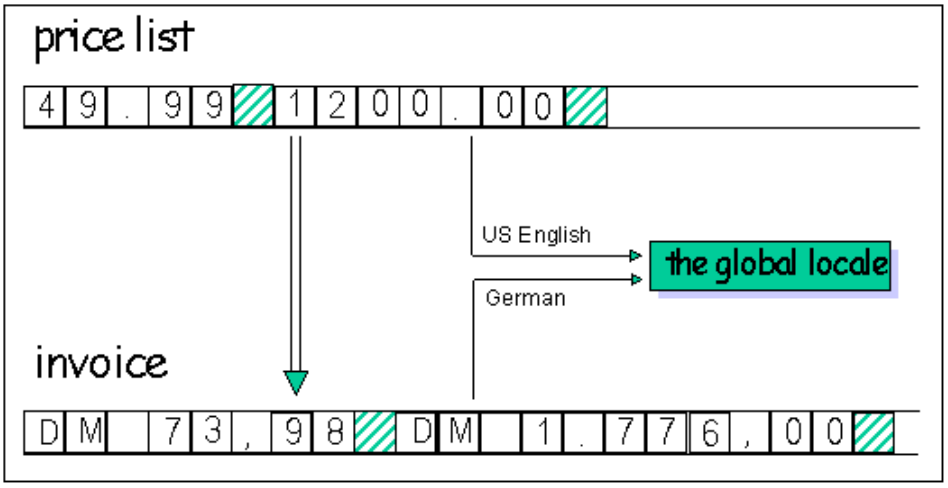
Multiple locales. It may well happen that you do have to work with multiple locales. For example, to implement an application for Switzerland, you might want to output messages in Italian, French, and German. As the C locale is a global data structure, you must switch locales several times.

Let's look at an example of an application that works with multiple locales. Imagine an application that prints invoices to be sent to customers all over the world. Of course, the invoices must be printed in the customer's native language, so the application must write output in multiple languages. Prices to be included in the invoice are taken from a single price list. If we assume that the application is used by a US company, the price list is in US English.

The application reads input (the product price list) in US English, and writes output (the invoice) in the customer's native language, say German. Since there is only one global locale in C that affects both input and output, the global locale must change between input and output operations. Before a price is read from the English price list, the locale must be switched from the German locale used for printing the invoice to a US English locale. Before inserting the price into the invoice, the global locale must be switched back to the German locale. To read the next input from the price list, the locale must be switched back to English, and so forth. Figure 6 summarizes this activity.

Figure 6 -- Multiple locales in C

©Copyright 1999 Rogue Wave Software, Inc.

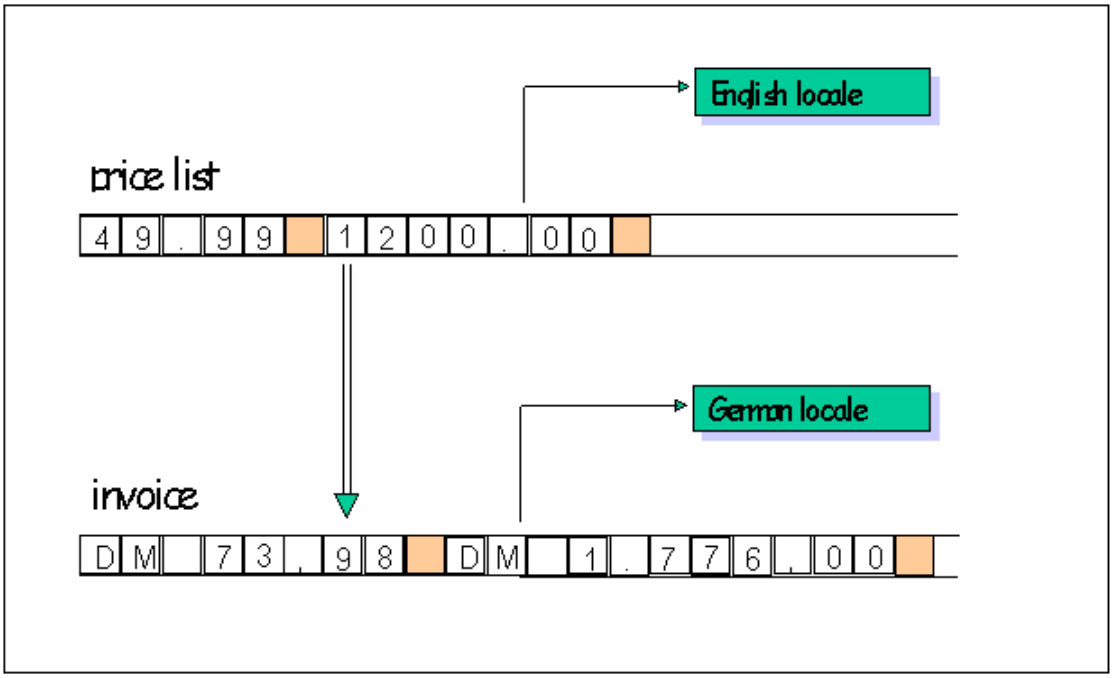


Here is the C code that corresponds to the previous example:

```
double price;
char buf[SZ];
while ( ... )  // processing the German invoice
{  setlocale(LC_ALL, "En_US");
   fscanf(priceFile,"%lf",&price);
   // convert $ to DM according to the current exchange rate
   setlocale(LC_ALL,"De_DE");
   strfmon(buf,SZ,"%n",price);
   fprintf(invoiceFile,"%s",buf);
}
```

Using C++ locale objects dramatically simplifies the task of communicating between multiple locales. The iostreams in the Standard C++ Library are internationalized so that streams can be imbued with separate locale objects. For example, the input stream can be imbued with an English locale object, and the output stream can be imbued with a German locale object. In this way, switching locales becomes unnecessary, as demonstrated in Figure 7:

Figure 7 -- Multiple locales in C++



Here is the C++ code corresponding to the previous example1:

```
priceFile.imbue(locale("En_US"));
invoiceFile.imbue(locale("De_DE"));
moneytype price;
while ( ... )  // processing the German invoice
{  priceFile >> price;
   // convert $ to DM according to the current exchange rate
```

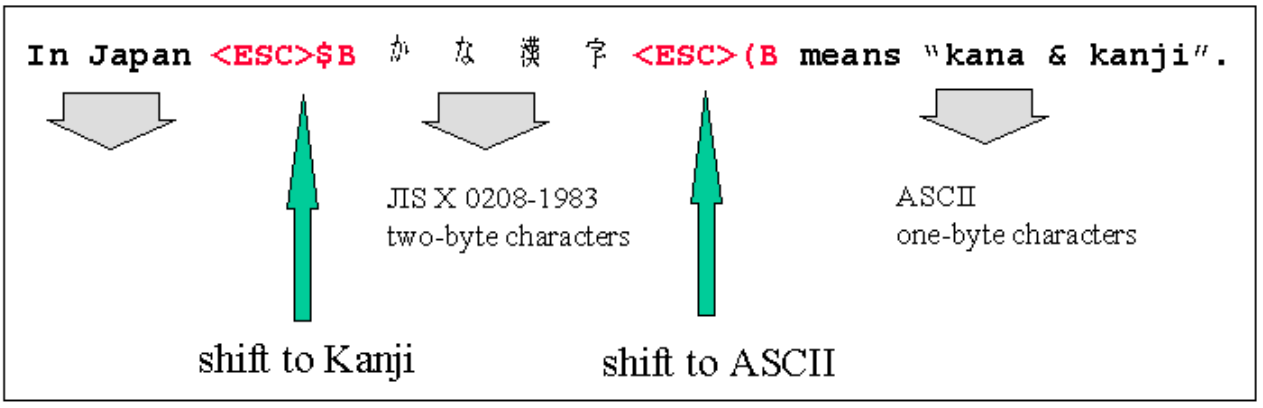
©Copyright 1999 Rogue Wave Software, Inc.

```
   invoiceFile << price;
}
```

Because the examples given above are brief, switching locales might look like a minor inconvenience. However, it is a major problem once code conversions are involved.

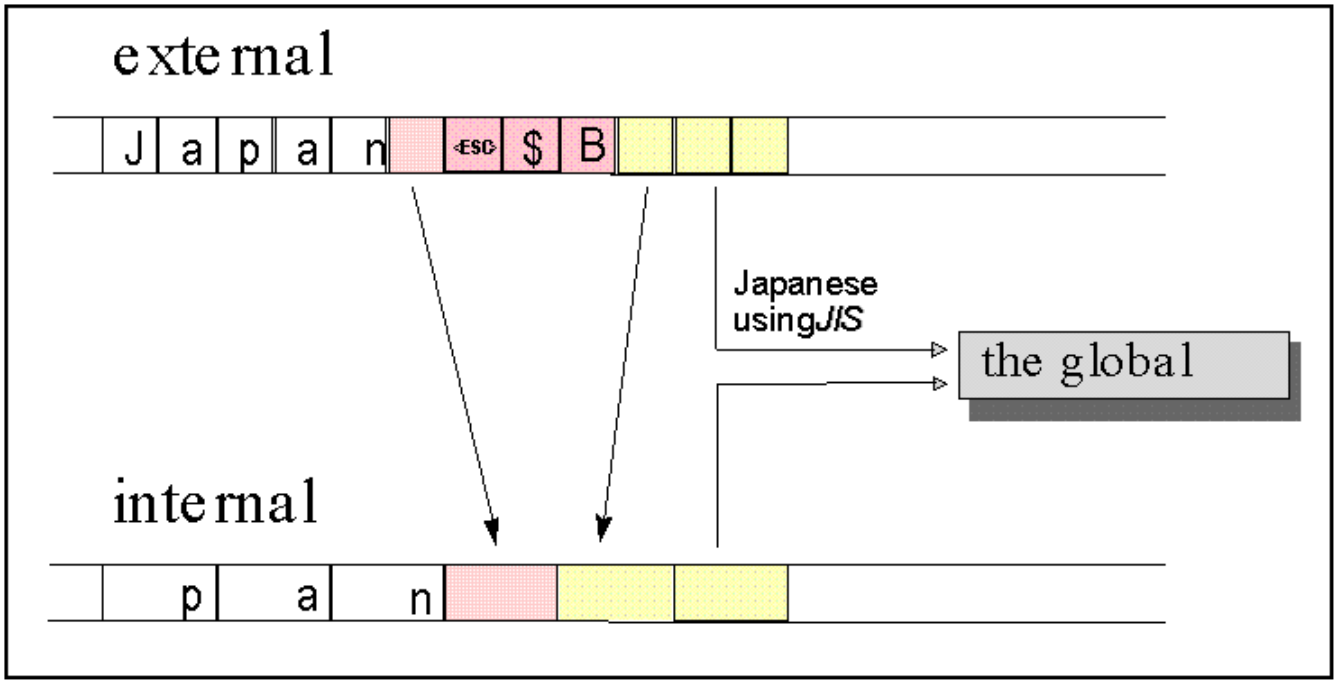
To underscore the point, let us revisit the JIS encoding scheme using the shift sequence described in Figure 2, which is repeated for convenience here as Figure 8. As you remember, you must maintain a *shift state* with these encodings while parsing a character sequence:

Figure 8 -- The Japanese text encoded in JIS from Figure 2



Suppose you are parsing input from a multibyte file which contains text that is encoded in JIS, as shown in Figure 9. While you parse this file, you have to keep track of the current shift state so you know how to interpret the characters you read, and how to transform them into the appropriate internal wide character representation.

Figure 9 -- Parsing input from a multibyte file using the global C locale



The global C locale can be switched during parsing; for example, from a locale object specifying the input to be in JIS encoding, to a locale object using EUC encoding instead. The current shift state becomes invalid each time the locale is switched, and you have to carefully maintain the shift state in an application that switches locales.

As long as the locale switches are intentional, this problem can presumably be solved. However, in multithreaded environments, the global C locale may impose a severe problem, as it can be switched inadvertently by another otherwise unrelated thread of execution. For this reason, internationalizing a C program for a multithreaded environment is difficult.

If you use C++ locales, on the other hand, the problem simply goes away. You can imbue each stream with a separate locale object, making inadvertent switches impossible. Let us now see how C++ locales are intended to be used.

©Copyright 1999 Rogue Wave Software, Inc.

Common Uses of C++ Locales

The C++ locale is commonly used as a default locale, with multiple locales, and as a global locale.

Classic locale. If you are not involved with internationalizing programs, you won't need C++ locales any more than you need C locales. If you can safely assume that users of your applications are accommodated by classic US English ASCII behavior, you do not require localization features. For you, the ***Standard C++ Library*** provides a predefined locale object, `locale::classic()`, that represents the US English ASCII locale.

Native locale. We use the term *native locale* to describe the locale that has been chosen as the preferred locale by the user or system administrator. On Unix systems, this is usually done by setting environment variables such as `LANG`. You can create a C++ locale object for the native locale by calling the constructor `locale("")`, that is, by requesting a named locale using an empty string as the name. The empty string tells the system to get the locale name from the environment, in the same way as the C library function `setlocale("")`.

Named locales. As implied above, a locale can have a name. The name of the classic locale is `"C"`. Unfortunately, the names of other locales are very much platform dependent. Consult your system documentation to determine what locales are installed and how they are named on your system. If you attempt to create a locale using a name that is not valid for your system, the constructor throws a `runtime_error` exception.

Multiple locales. Working with many different locales becomes easy when you use C++ locales. Switching locales, as you did in C, is no longer necessary in C++. You can imbue each stream with a different locale object. You can pass locale objects around and use them in multiple places.

Global locale. There is a global locale in C++, as there is in C. Initially, the global locale is the classic locale described above. You can change the global locale by calling locale::global().

You can create snapshots of the current global locale by calling the default constructor for a locale, locale::locale(). Snapshots are immutable locale objects and are not affected by any subsequent changes to the global locale.

Internationalized components like iostreams use the global locale as a default. If you do not explicitly imbue a stream with a particular locale, it is imbued by default with a snapshot of whatever locale was global at the time the stream was created.

Using the global C++ locale, you can work much as you did in C. You activate the native locale once at program start-in other words, you make it global-and use snapshots of it thereafter for all tasks that are locale-dependent. The following code demonstrates this procedure:

```
locale::global(locale(""));                          //1
...
string t = print_date(today, locale());              //2
...
locale::global(locale("Fr_CH"));                     //3
...
cout << something;                                   //4
```

//1 Make the native locale global.

//2 Use snapshots of the global locale whenever you need a locale object. Assume that print_date() is a function that formats dates. You would provide the function with a snapshot of the global locale in order to do the formatting.

//3 Switch the global locale; make a French locale global.

//4 Note that in this example, the standard stream `cout` is still imbued with the classic locale, because that was the global locale at program startup when `cout` was created. Changing the global locale does not change the locales of pre-existing streams. If you want to imbue the new global locale on `cout`, you should call `cout.imbue(locale())` after calling `locale::global()`.

The Relationship between the C Locale and the C++ Locale

The C locale and the C++ locales are mostly independent. However, if a C++ locale object has a name, making it global via locale::global()causes the C locale to change through a call to setlocale(). When this happens, locale-sensitive C functions called from within a C++ program use the changed C locale.

There is no way to affect the C++ locale from within a C program.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The Locale Object

A C++ locale object is a container of facet objects which encapsulate internationalization services, and represent culture and language dependencies. Here are some functions of class locale which allow you to create locales:

```
class locale {
    public:
// construct/copy/destroy:
      explicit locale(const char* std_name);                  //1
// global locale objects:
      static const locale& classic();                         //2
};
```

//1 You can create a locale object from a C locale's external representation. The constructor locale::locale(const char* std_name) takes the name of a C locale. This locale name is like the one you would use for a call to the C library function setlocale().

//2 You can also use a predefined locale object, locale::classic(), which represents the US English ASCII environment.

For a comprehensive description of the constructors described above, see the *Class Reference*.

It's important to understand that locales are immutable objects: once a locale object is created, it cannot be modified. This makes locales reliable and easy to use. As a programmer, you know that whenever you use pointers or references to elements held in a container, you have to worry about the validity of the pointers and references. If the container changes, pointers and references to its elements might not be valid any longer.

A locale object is a container, too. However, it is an immutable container; that is, it does not change. Therefore, you can take references to a locale's facet objects and pass the references around without worrying about their validity, as long as the locale object or any copy of it remains in existence. The locale object is never modified; no facets can be silently replaced.

At some time, you will most likely need locale objects other than the US classic locale or a snapshot of the global locale. Since locales are immutable objects, however, you cannot take one of these and replace its facet objects. You have to say at construction time how they shall be built.

Here are some constructors of class locale which allow you to build a locale object by composition; in other words, you construct it by copying an existing locale object, and replacing one or several facet objects.

```
class locale {
public:
  locale(const locale& other, const char* std_name, category);
  template <class Facet> locale(const locale& other, Facet* f);
  locale(const locale& other, const locale& one, category);
};
```

The following example shows how you can construct a locale object as a copy of the classic locale object, and take the numeric facet objects from a German locale object:

```
locale loc ( locale::classic(), locale("De_DE"), LC_NUMERIC );
```

For a comprehensive description of the constructors described above, see the *Class Reference*.

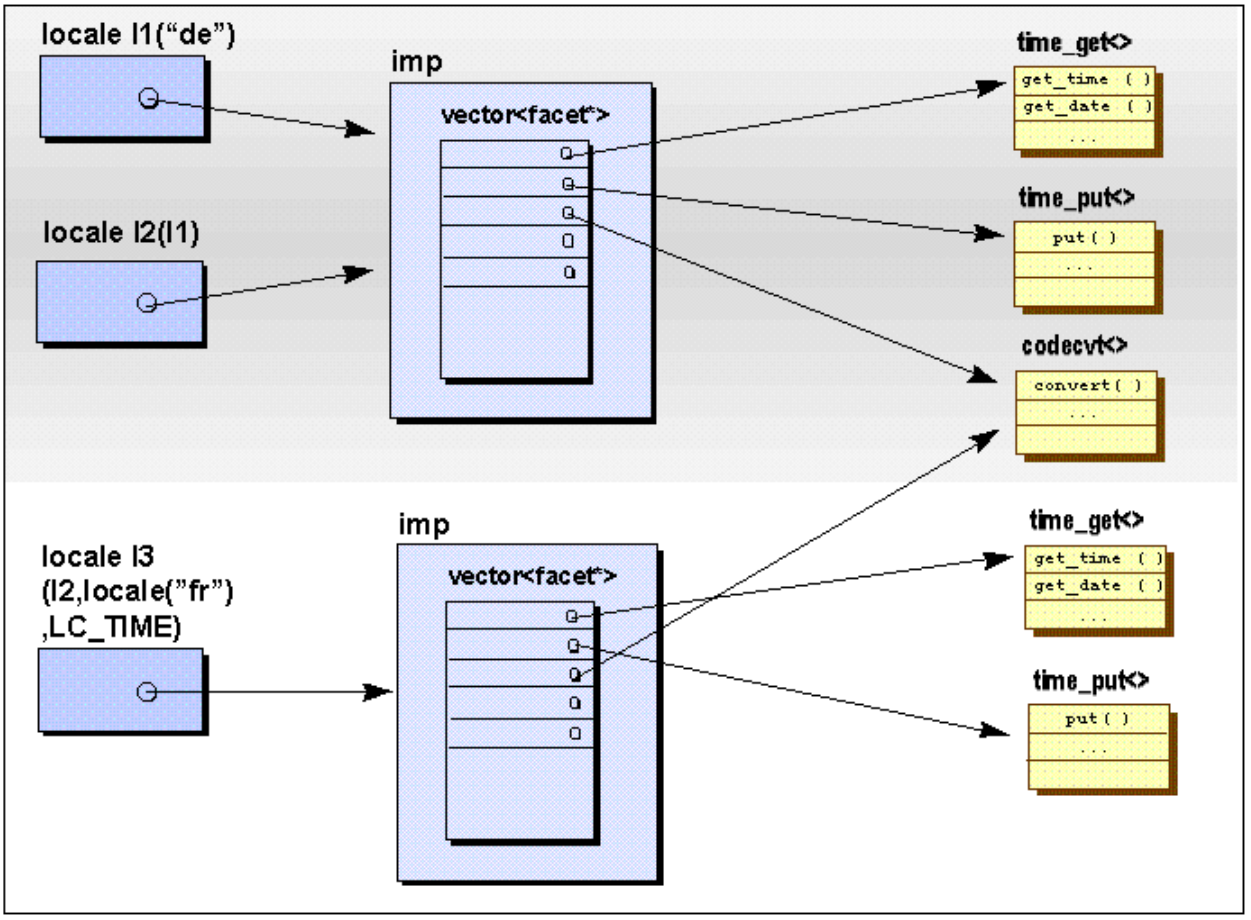
Copying a locale object is a cheap operation. You should have no hesitation about passing locale objects around by value. You may copy locale objects for composing new locale objects; you may pass copies of locale objects as arguments to functions, etc.

Locales are implemented using reference counting and the handle-body idiom: When a locale object is copied, only its handle is duplicated, a fast and inexpensive action. Similarly, constructing a locale object with the default constructor is cheap-this is equivalent to copying the global locale object. All other locale constructors that take a second locale as an argument are moderately more expensive, because they require cloning the body of the locale object. However, the facets are not all copied. The byname constructor is the most expensive, because it requires creating the locale from an external locale representation.

©Copyright 1999 Rogue Wave Software, Inc.

Figure 10 describes an overview of the locale architecture. It is a handle to a body that maintains a vector of pointers of facets. The facets are reference-counted, too.

Figure 10 -- The locale architecture



Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 4: Facets

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Understanding Facet Types

A *facet* is an object which can be stored in a locale and retrieved from the locale based on its type. Facets encapsulate data about cultural and language dependencies. They also contain services (functions) that use the encapsulated data to help you internationalize your programs.

The Standard C++ Library defines a number of facet classes, which we call *standard facets*. These facets were reviewed in the previous chapter; they are present in every locale. You can derive your own facet classes from the standard facets, to modify their behavior. You can also create entirely new facet classes, to handle aspects of internationalization that the standard facets don't cover. Both of these processes are described later in this chapter.

In talking about facet classes, we need to distinguish between *base facets* and *derived facets*. A *base facet* is a class with the following properties:

- It is derived from class `locale::facet`.
- It contains a static data member named `id`, of type `locale::id`.

These properties make it possible to store the facet in a locale, and to retrieve it from the locale based on its type. A locale contains at most one facet of each base facet type.

A *derived facet* is a class that is derived from a base facet, but does not contain its own static `locale::id` member. Instead, it inherits this member from the base facet it's derived from. Like any facet, a derived facet can be stored in a locale, but it occupies the same *slot* in the locale as its base facet, displacing any other facet that was in that slot. For example, the following skeletal code defines a mythical base facet and a facet derived from it, and shows some of the ways these mythical facets can be stored in and retrieved from locales:

```
class mythical: public locale::facet {                          //1
 public:
  static locale::id id;
  ... // etc
};

class mythical_byname: public mythical {                        //2
 public:
  mythical_byname (char *name,/*etc*/); // Constructor
  ... // etc
};

int main (void) {
  locale loc0(locale::classic());                               //3
  locale loc1(loc0,new mythical);                               //4
  locale loc2(loc1,new mythical_byname("he_DO"));               //5
  const mythical &m2=use_facet<mythical>(loc2);                 //6
  const mythical &m1=use_facet<mythical>(loc1);                 //7
  const mythical &m0=use_facet<mythical>(loc0);                 //8
}
```

//1 An example of a base facet class, derived from `locale::facet` and containing a static `locale::id` member name `id`.

//2 A derived facet class.

//3 A copy of the classic locale. Like all locales, it contains many facets. The facets are kept in *slots* indexed by the set of possible base facet types. Thus there is one slot for the base facet type `mythical`. In the classic locale, this `mythical` slot is empty.

//4 A copy of `loc0`, the classic locale, with its `mythical` slot now occupied by a newly-created `mythical` facet.

//5 A copy of `loc1`, but with the `mythical` slot now occupied by a newly-created `mythical_byname` facet.

//6 This returns a reference to the `mythical_byname` facet constructed in `//5`. However, note that you no longer know that it is a `mythical_byname` facet (short of using RTTI, which is cheating in this context). All you know is that it is a `mythical` facet, because it came from the `mythical` slot in the locale. (The `use_facet` function template is described in more detail later in this chapter.)

//7 This returns a reference to the mythical facet constructed in step `//4`.

©Copyright 1999 Rogue Wave Software, Inc.

//8 This causes a `runtime_error` exception to be thrown, because the `mythical` slot in `loc0` is empty.

The standard facets that come with the library are all defined as class templates. The first template parameter is always the character type the facet will operate on, which is usually `char` or `wchar_t`. Some of the facets have additional template parameters. As another example of the distinction between base facets and derived facets, here is a stripped-down version of the Standard facet template `numpunct`, which takes a single template parameter.

```
template <class charT>
class numpunct: public locale::facet {
 public:
  static locale::id id;
  ... // etc
};

template <class charT>
class numpunct_byname: public numpunct<charT> {
  ... // etc
};
```

Typically, these templates are instantiated on `char` and `wchar_t`, so the following are base facet types:

```
numpunct<char>
numpunct<wchar_t>
```

and the following are derived facet types:

```
numpunct_byname<char>
numpunct_byname<wchar_t>
```

If your application created its own character type `my_char_t`, then `numpunct<my_char_t>` would be a base facet type and `numpunct_byname<my_char_t>` would be a derived facet type.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Facet Lifetimes

In the `mythical` example in the previous section, we created two facet objects with operator `new` and incorporated them into locales, but we never deleted the facets with operator `delete`. This is poor coding for most objects, but for facet objects it is correct. Once a facet has been incorporated into a locale, the locale system takes over responsibility for deleting the facet when it is no longer needed, that is, when all locales that contain references to it have been destroyed.

This is a useful feature. For example, consider the following statement:

```
cout.imbue(locale(locale(),new numpunct_byname<char>("de_DE")));
     |     |      |        |
    (4)   (3)    (1)      (2)
```

What's happening here is fairly complex, so let's trace it out step by step:

1. We create a temporary locale object that is a copy of the current global locale.
2. We create a `numpunct` facet for a German locale.
3. We create another temporary locale which is a copy of the one created in `(1)`, but with its `numpunct` facet replaced by the facet created in `(2)`.
4. Using locale's copy constructor on most compilers, we pass this locale by value to `cout.imbue`, which saves a copy of it in the stream object.

At the end of this operation, all the temporary objects are quietly destroyed, but a reference to the new `numpunct` facet remains behind in the locale object inside `cout`. Other functions can retrieve this object using `cout.getloc()`, and make copies of it and its facets. All in all, it would be a burden on an application to have to keep track of all this, in order to determine when it is safe to delete the `numpunct_byname` object. We can be grateful that the locale system takes care of this chore.

It is possible, but not recommended, to override this default behavior. The constructors of all the standard facets, and the constructor of the base class `locale::facet`, all take an integer parameter named `refs`, which defaults to `0` when you don't specify it. This is the usual case. If you specify it as `1` or some other positive integer, the locale system does not delete the facet.

This is necessary for facet objects that can't have the `delete` operator applied to them, for instance, because they were created on the stack or as static objects. To discourage you from creating facets this way, all the standard facets have protected destructors. The following code causes a compilation error:

```
static numpunct<char> my_static_np(1);     // Error: no public
                                           // destructor

int main () {
  numpunct_byname<char> my_np("de_DE",1);  // Error: no public
                                           // destructor
  ...
}
```

You can circumvent this protection by deriving a class with a public destructor from the standard facet. This may seem reasonable if you are going to use it only as a stand-alone object, and do not plan to incorporate it into any locale. For example, the following code lets you retrieve some data from the standard `numpunct<char>` facet without the overhead of `new/delete`:

```
class my_numpunct: public numpunct<char> { };
int main () {
  my_numpunct np;
  cout << np.truename() << " or " << np.falsename() << '?' << endl;
  ...
}
```

©Copyright 1999 Rogue Wave Software, Inc.

However, the following code accomplishes the same thing even faster, because `use_facet` is very fast compared to the cost of calling most facets' constructors:

```
int main () {
  const numpunct<char> &np=
      use_facet<numpunct<char> >(locale::classic());
  cout << np.truename() << " or " << np.falsename() << '?' << endl;
  ...
}
```

In short, while it is possible to create facet objects on the stack, there is rarely a reason to do so. It is probably better to adapt a consistent policy of always creating facet objects with operator `new`. If you incorporate the facet into a locale, you do not need to delete it, and indeed, deleting it would be an error. If you never incorporate it into any locale, you should delete it when you are finished with it. Instead of creating and deleting a stand-alone facet in this case, however, it is usually possible (and faster) to accomplish the same thing by retrieving a reference to an existing facet from some locale.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Accessing a Locale's Facets

A locale object is like a container or a map, to be more precise, but it is indexed by type at compile time. The indexing operator, therefore, is not operator[], but rather the template operator <>. Access to the facet objects of a locale object is via two member function templates, use_facet and has_facet:

```
template <class Facet> const Facet&     use_facet(const locale&);
template <class Facet> bool             has_facet(const locale&);
```

The code below demonstrates how they are used. It is an example of the ctype facet's usage; all upper case letters of a string read from the standard input stream are converted to lower case letters and written to the standard output stream.

```
string in;
cin >> in;
if (has_facet< ctype<char> >(locale()))              //1
{  cout << use_facet< ctype<char> >(locale())        //2
               .tolower(in.begin(),in.end());        //3
}
```

//1 In the call to has_facet<...>(), the template argument chooses a base facet class. If no facet of this type is present in a locale object, has_facet returns false.

//2 The function template use_facet<...>() returns a reference to the locale's facet object of the specified base facet type. As locale objects are immutable, the reference stays valid throughout the lifetime of the locale object. If no facet of the specified type is present in the locale, `use_facet` throws a `runtime_error` exception.

//3 The facet object's member function tolower() is called. It has the functionality of the C function tolower(); it converts all upper case letters in the string into lower case letters.

In this example, the call to `has_facet` is actually unnecessary because `ctype<char>` is a standard facet. Every locale always contains a complete complement of the standard facets, so `has_facet<ctype<char> >` always returns `TRUE`. However, a call to has_facet() can be useful when you are dealing with nonstandard facets, such as the `mythical` facet described earlier in this chapter, which may not be present in every locale.

When you are coding `use_facet` and `has_facet` calls, it is important that the facet type you specify as the template parameter is a base facet type, and not a derived facet type. The following is an error:

```
locale loc;
const numpunct_byname<char> &np =             // Error, use_facet
      use_facet<numpunct_byname<char> >(loc); // instantiated on
                                              // a non-base facet
                                              // type
```

Depending on the facet type, and on your compiler, this will probably cause a compile-time error. If it does not, it may result in unpredictable runtime behavior. The `use_facet` call returns the facet that occupies the slot for the type `numpunct_byname<char>`. But in fact, as described earlier, this is the same as the slot for the base facet type, `numpunct<char>`. So the above code may cause `np` to be initialized to a reference to an object that is not, in fact, of type `numpunct_byname<char>`.

To avoid errors like this, make sure that you only instantiate `use_facet` and `has_face` on base facet types, that is, on facet types that contain a static `locale::id` member.

```
locale loc;
const numpunct<char> &np =                    // Correct
      use_facet<numpunct<char> >(loc);
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Using a Stream's Facet

Here is a more advanced example that uses a `time_put` facet to print a date. Let us assume we have a date and want to print it this way:

```
struct tm aDate;                                          //1

memset(aDate,0,sizeof aDate);                             //2
aDate.tm_year = 1989;
aDate.tm_mon = 9;
aDate.tm_mday = 1;

cout.imbue(locale::locale("De_CH"));                      //3
cout << aDate;                                            //4
```

//1 A date object is created. It is of type tm, which is the time structure defined in the standard C library.

//2 The date object is initialized with a particular date, September 1, 1989.

//3 Let's assume our program is supposed to run in a German-speaking canton of Switzerland. Hence, we imbue the standard output stream with a German-Swiss locale.

//4 The date is printed in German to the standard output stream.

The output is: 1. September 1989

As there is no operator<<() defined in the Standard C++ Library for the time structure tm from the C library, we must provide this inserter ourselves. The following code suggests a way this can be done. If you are not familiar with iostreams, please refer to the iostreams part of this *User's Guide*.

To keep it simple, the handling of exceptions thrown during the formatting is omitted.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT,traits>& os, const tm& date)   //1
{
 locale loc = os.getloc();                                    //2
 typedef ostreambuf_iterator<charT,traits> outIter_t;         //3
 const time_put<charT,outIter_t>& fac                         //4
     = use_facet < time_put<charT, bufIter_t > > (loc);       //5
 fac.put(os,os,os.fill(),&date,'x');                          //6
 return os;
}
```

//1 This is a typical signature of a stream inserter; it takes a reference to an output stream and a constant reference to the object to be printed, and returns a reference to the same stream.

//2 The stream's locale object is obtained via the stream's member function getloc(). This is the locale object where we expect to find a time-formatting facet object.

We define a type for an output iterator to a stream buffer.

//3 Time-formatting facet objects write the formatted output via an iterator into an output container (see the sections on containers and iterators in the *Standard C++ Library User's Guide*). In principle, this can be an arbitrary container that has an output iterator, such as a string or a C++ array.

Here we want the time-formatting facet object to bypass the stream's formatting layer and write directly to the output stream's underlying stream buffer. Therefore, the output container shall be a stream buffer.

//4 We define a variable that will hold a reference to the locale object's time_put facet object. The time formatting facet class time_put has two template parameters:

- The *first* template parameter is the character type used for output. Here we provide the stream's character type as the template argument.

©Copyright 1999 Rogue Wave Software, Inc.

- The *second* template parameter is the output iterator type. Here we provide the stream buffer iterator type outIter_t that we had defined as before.

//5 Here we get the time-formatting facet object from the stream's locale via use_facet().

The facet object's formatting service put() is called. Let us see what arguments it takes. Here is the function's interface:

```
iter_type put (iter_type (a)
               ,ios_base& (b)
               ,char_type (c)
               ,const tm* (d)
               ,char)     (e)
```

The types iter_type and char_type stand for the types that were provided as template arguments when the facet class was instantiated. In this case, they are ostreambuf_iterator<charT,traits> and charT, where charT and traits are the respective streams template arguments.

Here is the actual call:

```
nextpos = fac.put(os,os,os.fill(),&date,'x');
```

Now let's see what the arguments mean:

//6 a. The first parameter is supposed to be an output iterator. We provide an iterator to the stream's underlying stream buffer. The reference os to the output stream is converted to an output iterator, because output stream buffer iterators have a constructor taking an output stream, that is, basic_ostream<charT,traits>&.

b. The second parameter is of type ios_base&, which is one of the stream base classes. The class ios_base contains data for format control (see the section on iostreams for details). The facet object uses this formatting information. We provide the output stream's ios_base part here, using the automatic cast from a reference to an output stream, to a reference to its base class.

c. The third parameter is the fill character. It is used when the output has to be adjusted and blank characters have to be filled in. We provide the stream's fill character, which one can get by calling the stream's fill() function.

d. The fourth parameter is a pointer to a time structure tm from the C library.

e. The fifth parameter is a format character as in the C function strftime(); the x stands for the locale's appropriate date representation.

f. The value returned is an output iterator that points to the position immediately after the last inserted character.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Modifying a Standard Facet's Behavior

At times you may need to replace a facet object in a locale by another kind of facet object. In the following example, let us derive from one of the standard facet classes, numpunct, and create a locale object in which the standard numpunct facet object is replaced by an instance of our new, derived facet class.

Here is the problem we want to solve. When you print boolean values, you can choose between the numeric representation of the values "true" and "false", or their alphanumeric representation.

```
int main(int argc, char** argv)
{
   bool any_arguments = (argc > 1);                          //1
   cout.setf(ios_base::boolalpha);                           //2
   cout << any_arguments << '\n';                            //3
   // ...
}
```

//1 A variable of type bool is defined. Its initial value is the boolean value of the logical expression (argc > 1), so the variable any_arguments contains the information, whether the program was called with or without arguments.

//2 The format flag ios_base:: boolalpha is set in the predefined output stream cout. The effect is that the string representation of boolean values is printed, instead of their numerical representation 0 or 1, which is the default representation.

//3 Here either the string "true" or the string "false" are printed.

Of course, the string representation depends on the language. Hence, the alphanumeric representation of boolean values is provided by a locale. It is the numpunct facet of a locale that describes the cultural conventions for numerical formatting. It contains services that return the string representations of the boolean values true and false.

This is the interface of facet numpunct:

```
template <class charT>
    class numpunct : public locale::facet {
    public:
      typedef charT               char_type;
      typedef basic_string<charT> string_type;
      explicit numpunct(size_t refs = 0);
      string_type  decimal_point()   const;
      string_type  thousands_sep()   const;
      vector<char> grouping()        const;
      string_type  truename()        const;
      string_type  falsename()       const;
      static locale::id id;
};
```

Now let us replace this facet. To make it more exciting, let's use not only a different language, but also different words for true and false, such as Yes! and No!. For just using another language, we would not need a new facet; we would simply use the right native locale, and it would contain the right facet.

```
template <class charT>                                       //1
class change_bool_names
    : public numpunct_byname<charT>                          //2
{
 public:
  typedef basic_string<charT> string_type;
  explicit change_bool_names (const char* name,              //3
      const charT* t, const charT* f, size_t refs=0)
      : numpunct_byname<charT> (name,refs),
        true_string(t), false_string(f) { }
 protected:
  string_type do_truename () const { return true_string; }   //4
  string_type do_falsename () const { return false_string; }
 private:
  string_type true_string, false_string;
};
```

©Copyright 1999 Rogue Wave Software, Inc.

//1 The new facet is a class template that takes the character type as a template parameter.
The new facet is derived from the numpunct_byname<charT> facet.

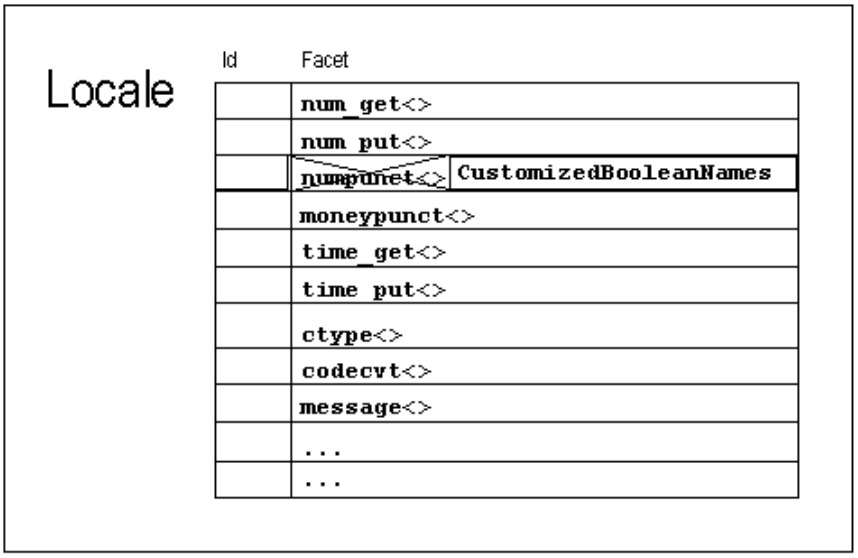
//2 The byname facets read the respective locale information from the external representation of a C locale. The name provided to construct a byname facet is the name of a locale, as you would use it in a call to setlocale().

//3 A constructor is provided that takes a locale name and the new values we want to display for the alpha versions of `true` and `false`. The fourth parameter, `refs`, controls the facet's lifetime, as described in an earlier section.

//4 The virtual member functions do_truename() and do_falsename() are reimplemented. They are called by the public member functions truename() and falsename(). See the *Class Reference* for further details.

Now let's create a German locale in which the `numpunct<char>` facet has been replaced by an object of our new derived facet type, as shown in Figure 11:

Figure 11 -- Replacing the numpunct<char> facet object



The code looks like this:

```
void main(int argc, char** argv)
{
  locale loc(locale("de_DE"),                                    //1
       new change_bool_names<char>("de_DE","Ja.","Nein."));      //2
  cout.imbue(loc);                                               //3
  cout << "Argumente vorhanden? "  //Any arguments?
       << boolalpha << (argc > 1) << endl;                       //4
}
```

//1 A locale object is constructed with an instance of the new facet class. The locale object has all facet objects from a German locale object, except that the new facet object change_bool_names substitutes for the numpunct facet object.

//2 The new facet object takes all information from a German numpunct facet object, and replaces the default native names for true and false with the provided strings "Ja."("Yes.") and "Nein."("No.").

//3 The standard output stream cout is imbued with the newly created locale.

//4 The expression (argc > 1) yields a boolean value, which indicates whether the program was called with arguments. This boolean value's alphanumeric representation is printed to the standard output stream. The output might be:

```
Argument vorhanden? Ja.
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Creating a New Base Facet Class

At times you may need to add a facet object to a locale without displacing any of the existing facets. To do this, you must define a new base facet class.

Here is an example of a new facet class like that. It is a facet that provides a service to check whether a character is a German umlaut, that is, one of the special characters äöüÄÖÜ.

```
class Umlaut : public locale::facet {                          //1
  public:
    static locale::id id;                                      //2
    bool is umlaut(char c) { return do_isumlaut(c);}           //3
    Umlaut(size_t refs=0): locale::facet(refs) {}              //4
  protected:
    virtual bool do_isumlaut(char);                            //5
};
```

//1 All base facet classes must be derived from class locale::facet.

//2 In addition, all base facet classes must contain a static member named `id`, of type `locale::id`. The locale system uses this object internally to identify the slot in locale objects where facets of this type are stored.

(Derived facet classes do not contain their own `id` members. Instead, they inherit the member from a base facet class, and therefore are stored in the same slot as the base class.)

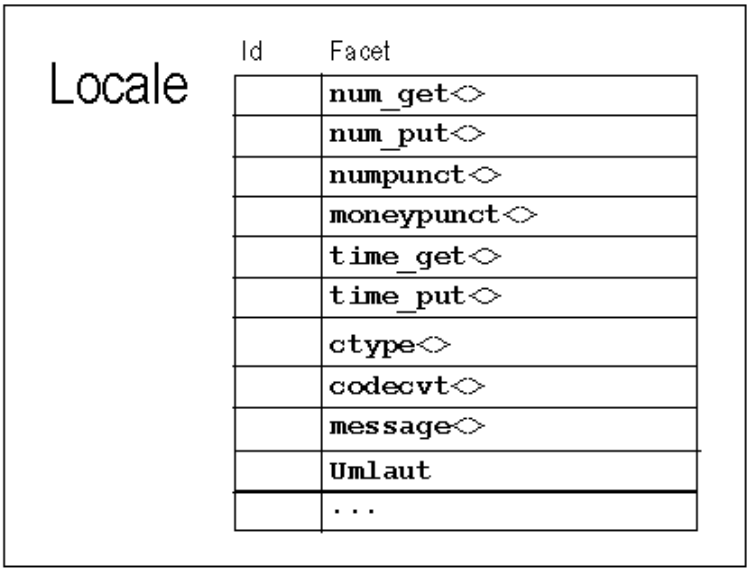
//3 A member function is_umlaut() is declared that returns the result of calling the protected virtual function do_umlaut.

//4 The constructor takes a `refs` parameter which is passed to the base class `locale::facet` for lifetime control, as described earlier.

//5 The actual functionality of determining whether a character is an umlaut is implemented in a protected virtual member function. In general, all localization services in a facet should be implemented in virtual functions this way, so that derived facets can override them when necessary.

Now let's create a locale with a facet of the new type, as shown in Figure 12:

Figure 12 -- Adding a new facet to a locale



The code for this procedure is given below:

```
locale loc(locale(""),  // native locale
           new Umlaut); // the new facet                        //1
char c,d;
while (cin >> c){
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 d = use_facet<ctype<char> >(loc).tolower(c);                //2
 if (has_facet<Umlaut>(loc))                                 //3
 { if (use_facet<Umlaut>(loc).is umlaut(d))                  //4
       cout << c << "belongs to the German alphabet!" << '\n';
 }
}
```

//1 A locale object is constructed with an instance of the new facet class. The locale object has all facet objects from the native locale object, plus an instance of the new facet class Umlaut.

//2 Let's assume our new umlaut facet class is somewhat limited; it can handle only lower case characters. Thus we have to convert each character to a lower case character before we hand it over to the umlaut facet object. This is done by using a ctype facet object's service function tolower().

//3 Before we use the umlaut facet object, we check whether such an object is present in the locale. In a toy example like this it is obvious, but in a real application it is advisable to check for the existence of nonstandard facet objects before trying to use them.

//4 The umlaut facet object is used, and its member function is umlaut() is called. Note that the syntax for using this newly contrived facet object is exactly like the syntax for using the standard ctype facet.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 5: Building Your Own Facet Class

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

An Example of Formatting Phone Numbers

The previous chapters explained how you can use locales and the standard facet classes, and how you can build new facet classes. This chapter introduces you to the technique of building your own facet class and using it in conjunction with the input/output streams of the Standard C++ Library, the iostreams. This material is rather advanced, and requires some knowledge of standard iostreams.

In the following sections, we will work through a complete example on formatting telephone numbers. Formatting telephone numbers involves local conventions that vary from culture to culture. For example, the same US phone number can have all of the formats listed below:

754-3010	Local
(541) 754-3010	Domestic
+1-541-754-3010	International
1-541-754-3010	Dialed in the US
001-541-754-3010	Dialed from Germany
191 541 754 3010	Dialed from France

Now consider a German phone number. Although a German phone number consists of an area code and an extension like a US number, the format is different. Here is the same German phone number in a variety of formats:

636-48018	Local
(089) / 636-48018	Domestic
+49-89-636-48018	International
19-49-89-636-48018	Dialed from France

Note the difference in formatting domestic numbers. In the US, the convention is `1 (area code) extension`, while in Germany it is `(0 area code)/extension`.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

A Phone Number Class

An application that has to handle phone numbers will probably have a class that represents a phone number. We will also want to read and write telephone numbers via iostreams, and therefore define suitable extractor and inserter functions. For the sake of simplicity, we will focus on the inserter function in our example.

To begin, here is the complete class declaration for the telephone number class `phoneNo`:

```
class phoneNo
{
public:
 typedef basic_ostream<char> outStream_t;
 typedef string string_t;

 phoneNo(const string_t& cc,const string_t& ac,const string_t& ex)
    : countryCode(cc), areaCode(ac), extension(ex) {}

private:
   string_t countryCode;                                    //"de"
   string_t areaCode;                                       //"89"
   string_t extension;                                   //"636-48018"

friend phoneNo::outStream_t& operator<<
   (phoneNo::outStream_t&, const phoneNo&);
};
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

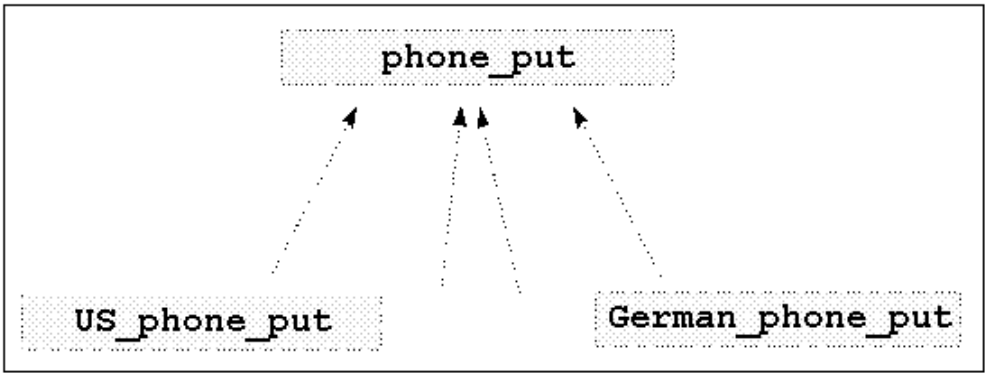
©Copyright 1999 Rogue Wave Software, Inc.

A Phone Number Formatting Facet Class

Now that we have locales and facets in C++, we can encapsulate the locale-dependent parsing and formatting of telephone numbers into a new facet class. Let's focus on formatting in this example. We call the new facet class phone_put, analogous to time_put, money_put, etc.

The phone_put facet class serves solely as a base class for facet classes that actually implement the locale-dependent formatting. The relationship of class phone_put to the other facet classes is illustrated in Figure 13:

Figure 13 -- The relationship of the phone_put facet to the implementing facets



Here is a first tentative declaration of the new facet class phone_put:

```
class phone_put: public locale::facet                        //1
{
public:
   static locale::id id;                                     //2
   phone_put(size_t refs = 0) : locale::facet(refs) { }      //3

   string_t put(const string_t& ext
                ,const string_t& area
                ,const string_t& cnt) const;                 //4
};
```

//1 Derive from the base class locale::facet, so that a locale object is able to maintain instances of our new phone facet class.

//2 New base facet classes need to define a static data member id of type locale::id.

//3 Define a constructor that takes the reference count that is handed over to the base class.

//4 Define a function put() that does the actual formatting.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

An Inserter for Phone Numbers

Now let's take a look at the implementation of the inserter for our phone number class:

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
 locale loc = os.getloc();                                    //1
 const phone_put& ppFacet = use_facet<phone_put> (loc);       //2
 os << ppFacet.put(pn.extension, pn.areaCode, pn.countryCode);//3
 return (os);
}
```

//1 The inserter function uses the output stream's locale object (obtained via getloc()),

//2 uses the locale's phone number facet object,

//3 and calls the facet object's formatting service put().

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The Phone Number Facet Class Revisited

Let us now try to implement the phone number facet class. What does this facet need to know?

- A facet needs to know its own locality, because a phone number is formatted differently for domestic and international use; for example, a German number looks like (089) / 636-48018 when used in Germany, but it looks like +1-49-89-636-48018 when used internationally.
- A facet needs information about the prefix for dialing international numbers; for example, 011 for dialing foreign numbers from the US, or 00 from Germany, or 19 from France.
- A facet needs access to a table of all country codes, so that one can enter a mnemonic for the country instead of looking up the respective country code. For example, I would like to say: "This is a phone number somewhere in Japan" without having to know what the country code for Japan is.

Adding Data Members

The following class declaration for the telephone number formatting facet class is enhanced with data members for the facet object's own locality, and its prefix for international calls (see //2 and //3 in the code below). Adding a table of country codes is omitted for the time being.

```
class phone_put: public locale::facet {
public:
  typedef string string_t;
  static locale::id id;
  phone_put(size_t refs = 0) : locale::facet(refs)
                             , myCountryCode_("")
                             , intlPrefix_("")      {  }
 string_t put(const string_t& ext,
              const string_t& area,
              const string_t& cnt) const;
protected:
  phone_put( const string_t& myC                              //1
           , const string_t& intlP
           , size_t refs = 0)
           : locale::facet(refs)
           , myCountryCode_(myC)
           , intlPrefix_(intlP)   {  }
  const string_t myCountryCode_;                              //2
  const string_t intlPrefix_;                                 //3
};
```

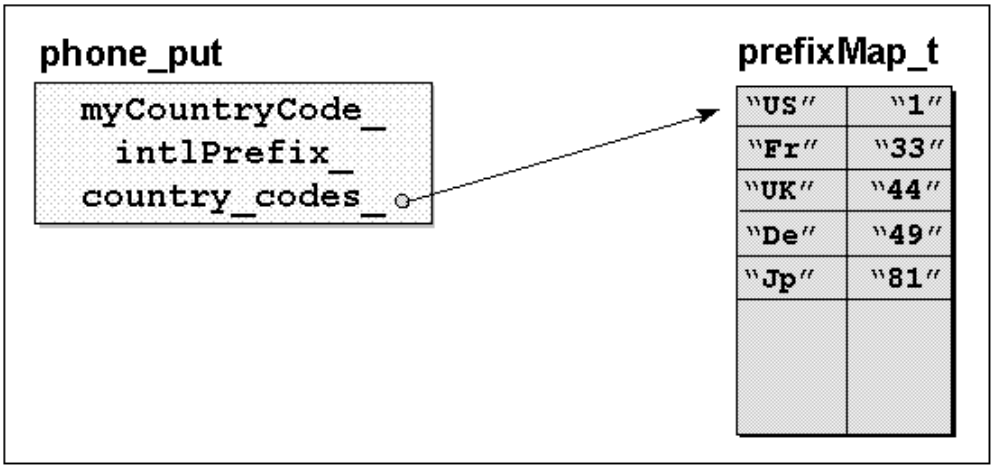
Note how this class serves as a base class for the facet classes that really implement a locale-dependent phone number formatting. Hence, the public constructor does not need to be extended, and a protected constructor is added instead (see //1 above).

Adding Country Codes

Let us now deal with the problem of adding the international country codes that were omitted from the previous class declaration. These country codes can be held as a map of strings that associates the country code with a mnemonic for the country's name, as shown in Table 14:

Figure 14 -- Map associating country codes with mnemonics for countries' names

©Copyright 1999 Rogue Wave Software, Inc.



In the following code, we add the table of country codes:

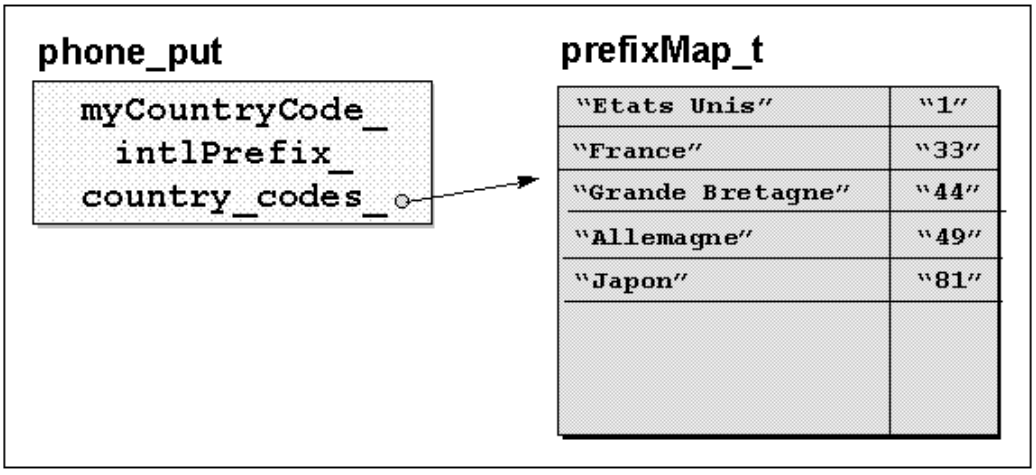
```
class phone_put: public locale::facet
{
 public:
   class prefixMap_t : public map<string,string>             //1
   {
    public:
       prefixMap_t() { insert(tab_t(string("US"),string("1")));
       insert(tab_t(string("De"),string("49")));
       // ...
   }
};
   static const prefixMap_t* std_codes()                     //2
          { return &stdCodes_; }
 protected:
   static const prefixMap_t stdCodes_;                       //3
};
```

Since the table of country codes is a constant table that is valid for all telephone number facet objects, it is added as a static data member stdCodes_ (see `//3`). The initialization of this data member is encapsulated in a class, prefixMap_t (see `//1`). For convenience, a function std_codes() is added to give access to the table (see //2).

Despite its appealing simplicity, however, having just one static country code table might prove too inflexible. Consider that mnemonics might vary from one locale to another due to different languages. Maybe mnemonics are not called for, and you really need more extended names associated with the actual country code.

In order to provide more flexibility, we can build in the ability to work with an arbitrary table. A pointer to the respective country code table can be provided when a facet object is constructed. The static table, shown in Figure 15 below, serves as a default:

Figure 15 -- Map associating country codes with country names



Since we hold the table as a pointer, we need to pay attention to memory management for the table pointed to. We use a flag for determining whether the provided table needs to be deleted when the facet is destroyed. The following code demonstrates use of the table and its associated flag:

©Copyright 1999 Rogue Wave Software, Inc.

```
class phone_put: public locale::facet {
public:
  typedef string string_t;
  class prefixMap_t;
  static locale::id id;

  phone_put( const prefixMap_t* tab=0                          //1
           , bool del = false
           , size_t refs = 0)
           : locale::facet(refs)
           , countryCodes_(tab), delete_it_(del)
           , myCountryCode_(""), intlPrefix_("")
  { if (tab)   {  countryCodes_ = tab;
                  delete_it_ = del;   }
    else       {  countryCodes_ = &stdCodes_;                  //2
                  delete_it_ = false; }
  }
  string_t put(const string_t& ext,
               const string_t& area,
               const string_t& cnt) const;

  const prefixMap_t* country_codes() const                     //3
  { return countryCodes_; }

  static const prefixMap_t* std_codes()    { return &stdCodes_; }
protected:
  phone_put(const string_t& myC, const string_t& intlP
          , const prefixMap_t* tab=0, bool del = false
          , size_t refs = 0)
          : locale::facet(refs)
          , countryCodes_(tab), delete_it_(del)
          , myCountryCode_(myC), intlPrefix_(intlP)
  { ... }
  virtual ~phone_put()
  { if(delete_it_)
      countryCodes_->prefixMap_t::~prefixMap_t();              //4
  }

  const prefixMap_t* countryCodes_;                            //5
  bool delete_it_;
  static const prefixMap_t stdCodes_;
  const string_t myCountryCode_;
  const string_t intlPrefix_;
};
```

//1 The constructor is enhanced to take a pointer to the country code table, together with the flag for memory management of the provided table.

//2 If no table is provided, the static table is installed as a default.

//3 For convenience, a function that returns a pointer to the current table is added.

//4 The table is deleted if the memory management flags says so.

//5 Protected data members are added to hold the pointer to the current country code table, as well as the associated memory management flag.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

An Example of a Derived Facet Class

As mentioned previously, the phone number facet class is intended to serve as a base class. Let's now present an example of a derived facet class, the US phone number formatting facet. It works by default with the static country code table and "US" as its own locality. It also knows the prefix for dialing foreign numbers from the US. Here is the class declaration for the facet:

```
class US_phone_put : public phone_put {
public:
   US_phone_put( const prefixMap_t* tab=0
               , const string_t& myCod = "US"
               , bool del = false
               , size_t refs = 0)
               : phone_put(myCod,"011",tab,del,refs)
   { }
};
```

Other concrete facet classes are built similarly.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Using Phone Number Facets

Now that we have laid the groundwork, we are almost ready to format phone numbers. Here is an example of how instances of the new facet class can be used:

```
ostream ofstr("/tmp/out");
ostr.imbue(locale(locale::classic(),new US_phone_put));       //1
ostr << phoneNo("Fr","1","60 17 07 16") << endl;
ostr << phoneNo("US","541","711-PARK") << endl;

ostr.imbue(locale(locale("Fr")                                //2
          ,new Fr_phone_put (&myTab,"France")));
ostr << phoneNo("Allemagne","89","636-40938") << endl;        //3
```

//1 Imbue an output stream with a locale object that has a phone number facet object. In the example above, it is the US English ASCII locale with a US phone number facet, and

//2 A French locale using a French phone number facet with a particular country code table.

//3 Output phone numbers using the inserter function. The output is:

1-33-1-60170716
(541) 711-PARK
19 49 89 636 40938

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Formatting Phone Numbers

Even now, however, the implementation of our facet class is incomplete. We still need to mention how the actual formatting of a phone number is implemented. In the example below, it is done by calling two virtual functions, put_country_code() and put_domestic_area_code():

```
class phone_put: public locale::facet {
public:
  // ...
  string put(const string& ext,
             const string& area,
             const string& cnt) const;
protected:
  // ...
  virtual string_t put_country_code
          (const string_t& country) const = 0;
  virtual string_t put_domestic_area_code
          (const string_t& area) const = 0;
};
```

Note that the functions put_country_code() and put_domestic_area_code() are purely virtual in the base class, and thus must be provided by the derived facet classes. For the sake of brevity, we spare you here the details of the functions of the derived classes. For more information, please consult the directory of sample code delivered on disk with this product.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Improving the Inserter Function

Let's turn here to improving our inserter function. Consider that the country code table might be huge, and access to a country code might turn out to be a time-consuming operation. We can optimize the inserter function's performance by caching the country code table, so that we can access it directly and thus reduce performance overhead.

Primitive Caching

The code below does some primitive caching. It takes the phone facet object from the stream's locale object and copies the country code table into a static variable.

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
   locale loc = os.getloc();
   const phone_put& ppFacet = use_facet<phone_put> (loc);

   // primitive caching
   static prefixMap_t codes = *(ppFacet.country_codes());

   // some sophisticated output using the cached codes
   ...
   return (os);
}
```

Now consider that the locale object imbued on a stream might change, but the cached static country code table does not. The cache is filled once, and all changes to the stream's locale object have no effect on this inserter function's cache. That's probably not what we want. What we do need is some kind of notification each time a new locale object is imbued, so that we can update the cache.

Registration of a Callback Function

In the following example, notification is provided by a callback function. The iostreams allow registration of callback functions. Class ios_base declares:

```
enum event { erase_event, imbue_event, copyfmt_event };       //1
typedef void (*event_callback) (event, ios_base&, int index);
void register_callback (event_callback fn, int index);        //2
```

Registered callback functions are called for three events:

- Destruction of a stream

//1

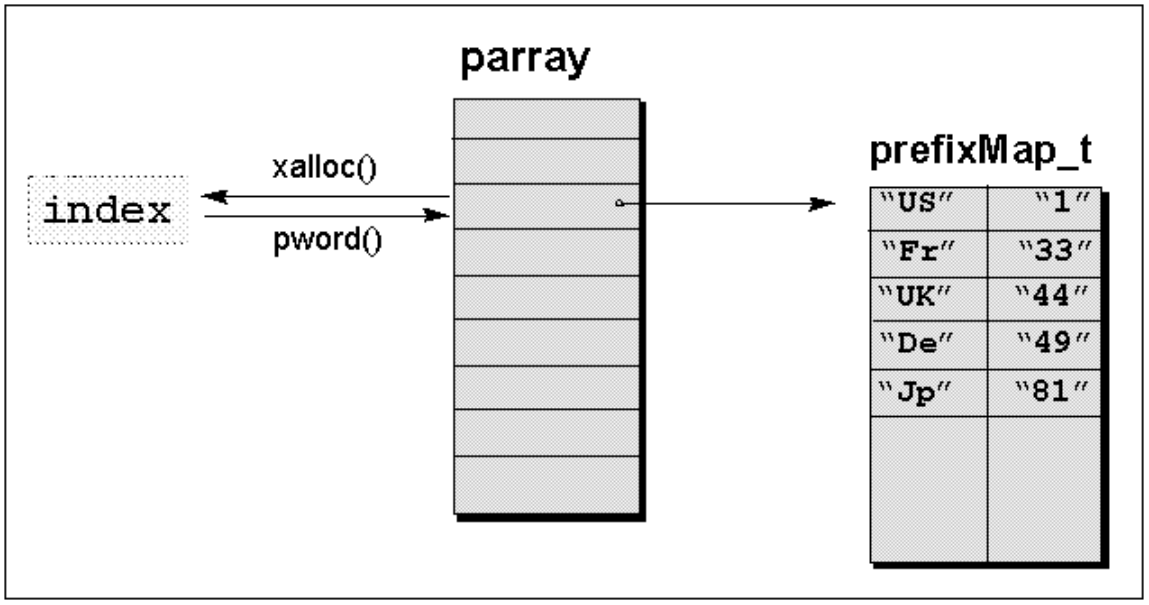
- Imbuing a new locale
- Copying the stream state.

//2 The register_callback() function registers a callback function and an index to the stream's parray. During calls to imbue(), copyfmt(), or ~ios_base(), the function fn is called with argument index. Functions registered are called when an event occurs, in opposite order of registration.

The parray is a static array in base class ios_base. One can obtain an index to this array via xalloc(), and access the array via pword(index) or iword(index), as shown in Figure 16:

Figure 16 -- The static array parray

©Copyright 1999 Rogue Wave Software, Inc.



In order to install a callback function that updates our cache, we implement a class that retrieves an index to parray and creates the cache, then registers the callback function in its constructor. The procedure is shown in the following code:

```
class registerCallback_t {
public:
registerCallback_t(ostream& os
                  ,ios_base::event_callback fct
                  ,prefixMap_t* codes)
  {
   int index = os.xalloc();                                  //1
   os.pword(index) = codes;                                  //2
   os.register_callback(fct,index);                          //3
  }
};
```

//1 An index to the array is obtained via xalloc().

//2 The pointer to the code table is stored in the array via pword().

//3 The callback function and the index are registered.

The actual callback function will later have access to the cache via the index to parray. At this point, we still need a callback function that updates the cache each time the stream's locale is replaced. Such a callback function could look like this:

```
void cacheCountryCodes(ios_base::event event
                      ,ios_base& str,int cache)
{  if (event == ios_base::imbue_event)                       //1
   {
      locale loc = str.getloc();
      const phone_put<char>& ppFacet =
                  use_facet<phone_put<char> > (loc);         //2

      *((phone_put::prefixMap_t*) str.pword(cache)) =
                  *(ppFacet.country_codes());                //3
   }
}
```

//1 It checks whether the event was a change of the imbued locale,

//2 retrieves the phone number facet from the stream's locale, and

//3 stores the country code table in the cache. The cache is accessible via the stream's parray.

Improving the Inserter

We now have everything we need to improve our inserter. It registers a callback function that updates the cache whenever necessary. Registration is done only once, by declaring a static variable of class registerCallback_t.

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
 static phone_put::prefixMap_t codes =
```

©Copyright 1999 Rogue Wave Software, Inc.

```
    *(use_facet<phone_put>(os.getloc()).country_codes());     //1

 static registerCallback_t cache(os,cacheCountryCodes,&codes);//2

  // some sophisticated output using the cached codes
  ...
}
```

//1 The current country code table is cached.

//2 The callback function cacheCountryCodes is registered.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Part III: Iostreams

Chapters in This Part

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 6: The Architecture of Iostreams

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

What Are the Standard Iostreams?

The Standard C++ Library includes classes for data stream input/output. Before the current ANSI/ISO standard, most C++ compilers were delivered with a class library commonly known as the i*ostreams* library. In this manual, we refer to this library as the *traditional iostreams*, in contrast to the *standard iostreams* that are now part of the ANSI/ISO Standard C++ Library. The standard iostreams are to some extent compatible with the traditional iostreams, in that the overall architecture and the most commonly used interfaces are retained. Chapter 24 describes the incompatibilities in greater detail.

We can compare the standard iostreams not only with the traditional C++ iostreams library, but also with the I/O support in the Standard C Library. Many former C programmers still prefer the input/output functions offered by the C library, often referred to as *C stdio*. Their familiarity with the C library is justification enough for using the C stdio instead of C++ iostreams, but there are other reasons as well. For example, calls to the C functions `printf()` and `scanf()` are admittedly more concise with C stdio. However, C stdio has drawbacks, too, such as type insecurity and inability to extend consistently for user-defined classes. We'll discuss these in more detail in the following sections.

Type Safety

Let us compare a call to stdio functions with the use of standard iostreams. The stdio call reads as follows:

```
int i = 25;
char name[50] = "Janakiraman";
fprintf(stdout, "%d %s", i, name);
```

It correctly prints: `25 Janakiraman`.

But what if we inadvertently switch the arguments to `fprintf`? The error is detected no sooner than run time. Anything can happen, from peculiar output to a system crash. This is not the case with the standard iostreams:

```
cout << i << ' ' << name << '\n';
```

Since there are overloaded versions of the shift operator `operator<<()`, the right operator is always called. The function `cout << i` calls `operator<<(int)`, and `cout << name` calls `operator<<(const char*)`. Hence, the standard iostreams are typesafe.

Extensibility to New Types

Another advantage of the standard iostreams is that user-defined types can be made to fit in seamlessly. Consider a type `Pair` that we want to print:

```
struct Pair { int x; string y; }
```

All we need to do is overload `operator<<()` for this new type `Pair`, and we can output pairs this way:

```
Pair p(5, "May");
cout << p;
```

The corresponding `operator<<()` can be implemented as:

```
basic_ostream<char>&
operator<<(basic_ostream<char>& o, const Pair& p)
{ return o << p.x << ' ' << p.y; }
```

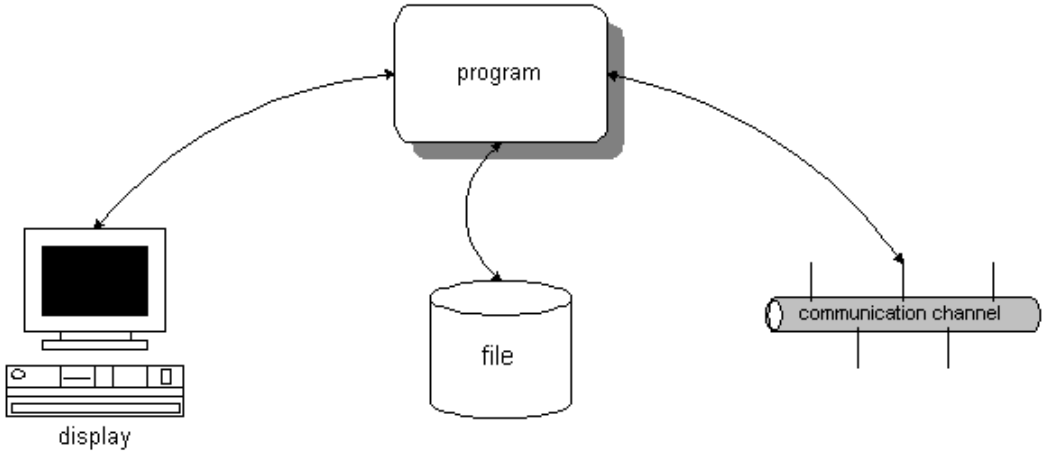

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

How Do the Standard Iostreams Work?

The main purpose of the standard iostreams is to serve as a tool for input and output of data. Generally, input and output are the transfer of data between a program and any kind of external device, as illustrated in Figure 17:

Figure 17 -- Data transfer supported by iostreams



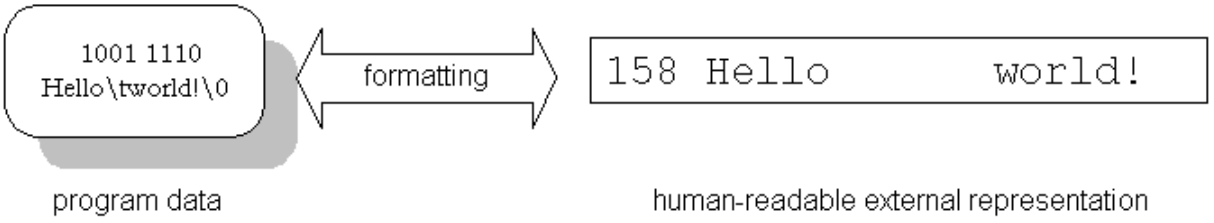
IOStreams supports data transfer between a program and external devices.

The internal representation of such data is meant to be convenient for data processing in a program. On the other hand, the external representation can vary quite a bit: it might be a display in human-readable form, or a portable data exchange format. The intent of a representation, such as conserving space for storage, can also influence the representation.

T*ext I/O* involves the external representation of a sequence of characters; every other case involves *binary I/O*. Traditionally, iostreams are used for text processing. Such text processing through iostreams involves two processes: *formatting* and *code conversion*.

Formatting is the transformation from a byte sequence representing internal data into a human-readable character sequence; for example, from a floating point number, or an integer value held in a variable, into a sequence of digits. Figure 18 illustrates the formatting process:

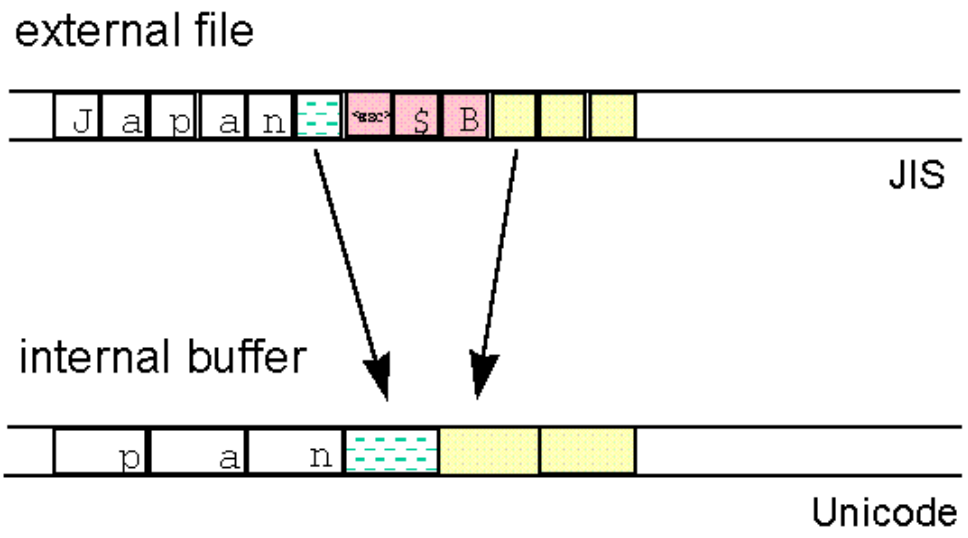
Figure 18 -- Formatting program data



Code conversion is the process of translating one character representation into another; for example, from wide characters held internally to a sequence of multibyte characters for external use. Wide characters are all the same size, and thus are convenient for internal data processing. Multibyte characters have different sizes and are stored more compactly. They are typically used for data transfer, or for storage on external devices such as files. Figure 19 illustrates the conversion process:

Figure 19 -- Code conversion between multibyte and wide characters

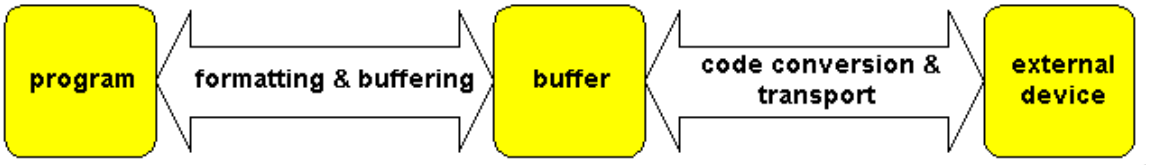
©Copyright 1999 Rogue Wave Software, Inc.



The Iostream Layers

The iostreams facility has two layers: one that handles formatting, and another that handles code conversion and transport of characters to and from the external device. The layers communicate through a buffer, as illustrated in Figure 20:

Figure 20 -- The iostreams layers



In the next sections we'll look at the function of each layer in more detail.

The Formatting Layer

In the formatting layer, the transformation between a program's internal data representation and a readable representation as a character sequence takes place. This formatting and parsing may involve, among other things:

- Precision and notation of floating point numbers
- Hexadecimal, octal, or decimal representation of integers
- Skipping of white space in the input
- Field width for output
- Adapting of number formatting to local conventions

The Transport Layer

The transport layer is responsible for producing and consuming characters. It encapsulates knowledge about the properties of a specific external device. Among other things, this involves:

- Block-wise output to files through system calls
- Code conversion to multibyte encodings

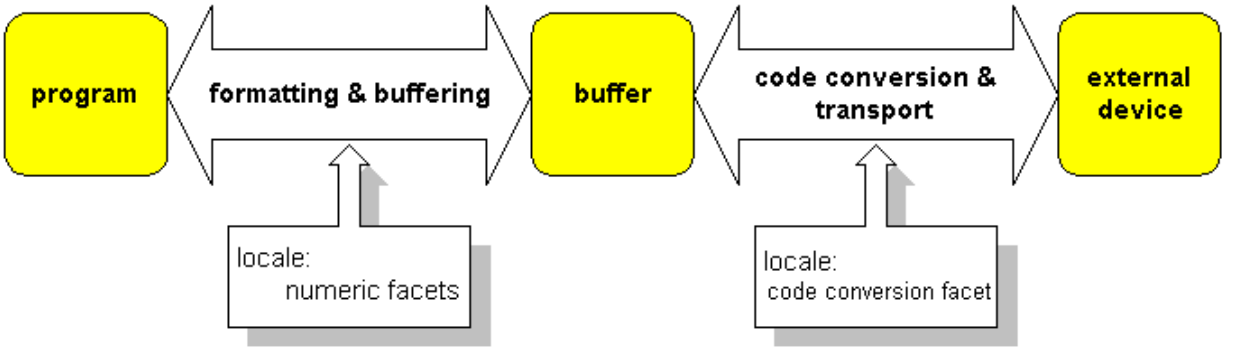
  To reduce the number of accesses to the external device, a buffer is used. For output, the formatting layer sends sequences of characters to the transport layer, which stores them in a *stream buffer*. The actual transport to the external device happens only when the buffer is full or explicitly flushed. For input, the transport layer reads from the external device and fills the buffer. The formatting layer receives characters from the buffer. When the buffer is empty, the transport layer is responsible for refilling it.

©Copyright 1999 Rogue Wave Software, Inc.

Locales

Both the formatting and the transport layers use the stream's locale. The formatting layer delegates the handling of numeric entities to the locale's numeric facets. The transport layer uses the locale's code conversion facet for character-wise transformation between the buffer content and characters transported to and from the external device. Figure 21 shows how locales are used with iostreams:

Figure 21 -- Use of locales in iostreams



File and In-Memory I/O

Iostreams support two kinds of I/O: file I/O and in-memory I/O.

File I/O involves the transfer of data to and from an external device. The device need not necessarily be a file in the usual sense of the word. It could just as well be a communication channel, or another construct that conforms to the file abstraction.

In contrast, *in-memory I/O* involves no external device. Thus code conversion and transport are not necessary; only formatting is performed. The result of such formatting is maintained in memory, and can be retrieved in the form of a character string.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

How Do the Standard Iostreams Help Solve Problems?

There are many situations in which iostreams are useful:

- **File I/O.** Iostreams can still be used for input and output to files, although file I/O has lost some of it former importance. In the past, alpha-numeric user interfaces were often built using file input/output to the standard input and output channels. Today almost all applications have graphical user interfaces.

  Nevertheless, iostreams are still useful for input and output to files other than the standard input and output channels, and to all other kinds of external media that fit into the file abstraction. For example, in Rogue Wave's **Tools.h++ Professional**, the `net` class library for network communications programming uses iostreams for input and output to communication streams, like sockets and pipes.

- **In-Memory I/O.** Iostreams can perform in-memory formatting and parsing. Even with a graphical user interface, you must format the text you want to display. The standard iostreams offer internationalized in-memory I/O, which is a great help for text processing tasks like formatting. The formatting of numeric values, for example, depends on cultural conventions. The formatting layer uses a locale's numeric facets to adapt its formatting and parsing to cultural conventions.

- **Internationalized Text Processing.** This function is actively supported by iostreams. Iostreams use locales. As locales are extensible, any kind of facet can be carried by a locale, and thus used by a stream. By default, iostreams use only the numeric and the code conversion facets of a locale. However, date, time, and monetary facets are available in the Standard C++ Library. Other cultural dependencies can be encapsulated in unique facets and made accessible to a stream. You can easily internationalize your use of iostreams to meet your needs.

- **Binary I/O.** The traditional iostreams suffer from a number of limitations. The biggest is the lack of conversion abilities: if you insert a `double` into a stream, for example, you do not know what format will be used to represent this `double` on the external device. There is no portable way to insert it as binary.

  Standard iostreams are by far more flexible. The code conversion performed on transfer of internal data to external devices can be customized: the transport layer delegates the task of converting to a code conversion facet. To provide a stream with a suitable code conversion facet for binary output, you can insert a `double` into a file stream in a portable binary data exchange format. No such code conversion facets are provided by the Standard C++ Library, however, and implementing such a facet is not trivial. As an alternative, you might consider implementing an entire stream buffer layer that can handle binary I/O.

- **Extending Iostreams.** In a way, you can think of iostreams as a framework that can be extended and customized. You can add input and output operators for user-defined types, or create your own formatting elements, the manipulators. You can specialize entire streams, usually in conjunction with specialized stream buffers. You can provide different locales to represent different cultural conventions, or to contain special purpose facets. You can instantiate iostreams classes for new character types, other than `char` or `wchar_t`.

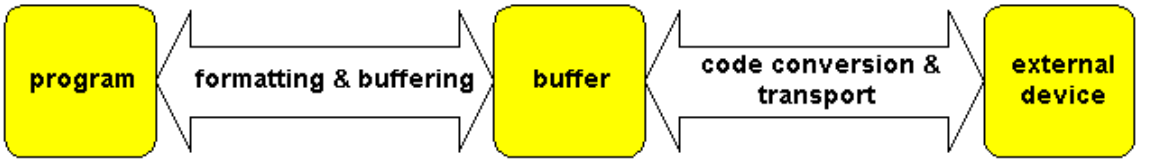
OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The Internal Structure of the Iostreams Layers

As explained earlier, the standard iostreams have two layers, one for formatting, and another for code conversion and transport of characters to and from the external device. For convenience, let's repeat here as Figure 22 the same illustration of the iostreams layers given as Figure 20:

Figure 22 -- The iostreams layers



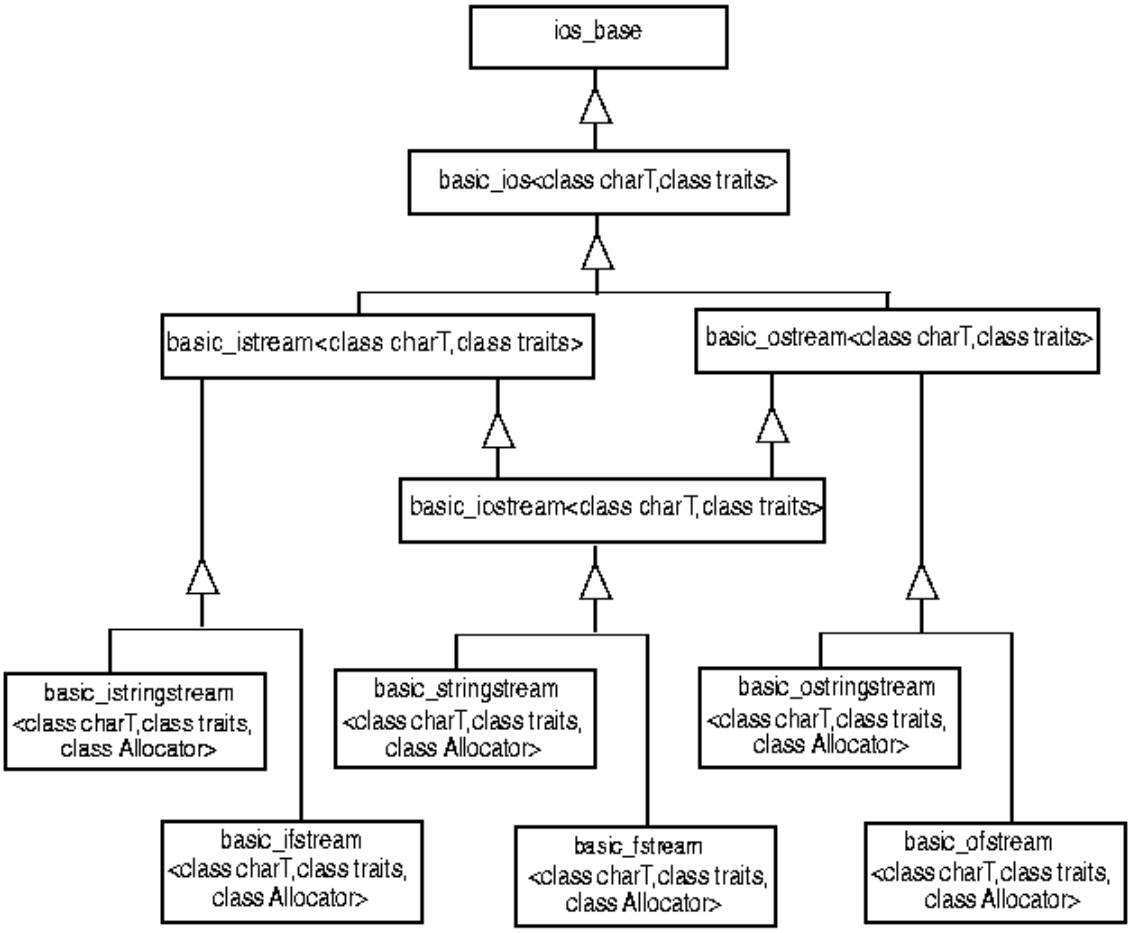
The next sections gives a more detailed description of the iostreams software architecture, including the classes and their inheritance relationship and respective responsibilities. If you would rather start using iostreams directly, go on to Chapter 7.

The Internal Structure of the Formatting Layer

Classes that belong to the formatting layer are often referred to as the stream classes. Figure 23 illustrates the class hierarchy of all the stream classes:

Figure 23 -- Internal class hierarchy of the formatting layer

©Copyright 1999 Rogue Wave Software, Inc.



You may notice that the classes `strstream`, `istrstream`, and `ostrstream` are not included in this diagram, even though we describe them in the *Class Reference*. Sometimes called *deprecated features* in the standard, these classes are provided solely for the sake of compatibility with the traditional iostreams, and will not be supported in future versions of the standard iostreams.

In the next sections, we discuss in more detail the components and characteristics of the components that are included in the class hierarchy given in Figure 23.

Iostreams Base Class ios_base

This class is the base class of all stream classes. Independent of character type, it encapsulates information that is needed by all streams. This information includes:

- Control information for parsing and formatting
- Additional information for the user's special needs, that is, a way to extend iostreams
- The locale imbued on the stream

Additionally, ***ios_base*** defines several types that are used by all stream classes, such as format flags, status bits, open mode, exception class, and so on.

The Iostreams Character Type-Dependent Base Class

Here is the virtual base class for the stream classes:

```
basic_ios<class charT, class traits=char_traits<charT> >
```

The class holds a pointer to the stream buffer, and to state information that reflects the integrity of the stream buffer. Note that ***basic_ios<>*** is a class template taking two parameters, the type of character handled by the stream, and the *character traits*.

©Copyright 1999 Rogue Wave Software, Inc.

The type of character can be type `char` for single-byte characters, or type `wchar_t` for wide characters, or any other user-defined character type. There are instantiations for `char` and `wchar_t` provided by the Standard C++ Library.

For convenience, there are typedefs for these instantiations:

`typedef basic_ios<char>` **ios** and `typedef basic_ios<wchar_t>` **wios**

Note that **ios** is not a class anymore, as it was in the traditional iostreams. If you have existing programs that use the old iostreams, they may no longer be compatible with the standard iostreams. (See Chapter 24.)

Character Traits

Character traits describe the properties of a character type. Many things change with the character type, such as:

- **The end-of-file value**. For type `char`, the end-of file value is represented by an integral constant called `EOF`. For type `wchar_t`, there is a constant defined that is called `WEOF`. For an arbitrary user-defined character type, the associated character traits define what the end-of-file value for this particular character type is.
- **The type of the `EOF` value**. This needs to be a type that can hold the `EOF` value. For example, for single-byte characters, this type is `int`, different from the actual character type `char`.
- **The equality of two characters**. For an exotic user-defined character type, the equality of two characters might mean something different from just bit-wise equality. Here you can define it.

A complete list of character traits is given in the *Class Reference* entry for `char_traits`.

There are specializations defined for type `char` and `wchar_t`. In general, this class template is not meant to be instantiated for a character type. You should always define class template specializations.

Fortunately, the Standard C++ Library is designed to make the most common cases the easiest. The traits template parameter has a sensible default value, so usually you need not bother with character traits at all.

The Input and Output Streams

The three stream classes for input and output are:

```
basic_istream <class charT, class traits=char_traits<charT> >
basic_ostream <class charT, class traits=char_traits<charT> >
basic_iostream<class charT, class traits=char_traits<charT> >
```

Class `istream` handles input, class `ostream` is for output. Class `iostream` deals with input *and* output; such a stream is called a *bidirectional* stream.

The three stream classes define functions for parsing and formatting, which are overloaded versions of `operator>>()` for input, called *extractors*, and overloaded versions of `operator<<()` for output, called *inserters*.

Additionally, there are member functions for unformatted input and output, like `get()`, `put()`, etc.

The File Streams

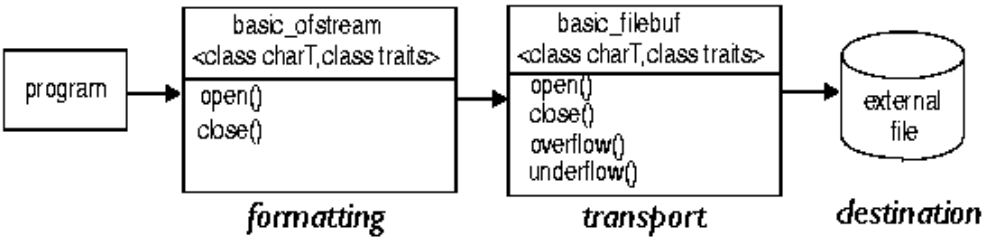
The file stream classes support input and output to and from files. They are:

```
basic_ifstream<class charT, class traits=char_traits<charT> >
basic_ofstream<class charT, class traits=char_traits<charT> >
basic_fstream<class charT, class traits=char_traits<charT> >
```

There are functions for opening and closing files, similar to the C functions `fopen()` and `fclose()`. Internally they use a special kind of stream buffer, called a *file buffer,* to control the transport of characters to/from the associated file. The function of the file streams is illustrated in Figure 24:

Figure 24 -- File I/O

©Copyright 1999 Rogue Wave Software, Inc.



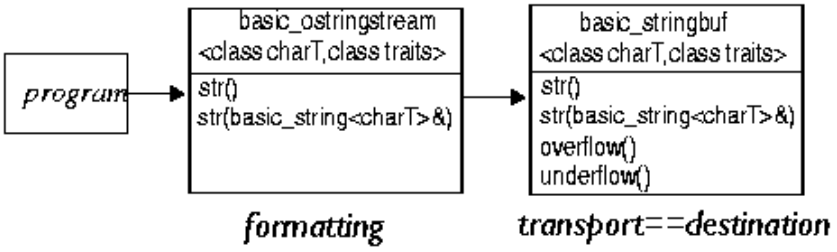
The String Streams

The string stream classes support in-memory I/O; that is, reading and writing to a string held in memory. They are:

```
basic_istringstream<class charT, class traits=char_traits<charT> >
basic_ostringstream<class charT, class traits=char_traits<charT> >
basic_stringstream<class charT, class traits=char_traits<charT> >
```

There are functions for getting and setting the string to be used as a buffer. Internally a specialized stream buffer is used. In this particular case, the buffer and the external device are the same. Figure 25 illustrates how the string stream classes work:

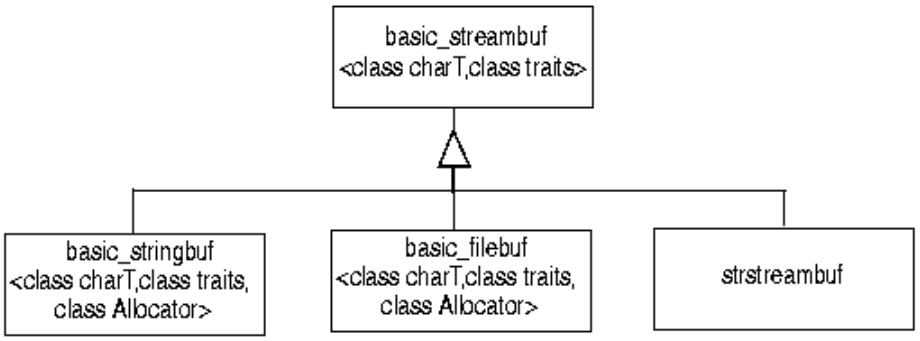
Figure 25 -- In-memory I/O



The Transport Layer's Internal Structure

Classes of the transport layer of the standard iostreams are often referred to as the stream buffer classes. Figure 26 gives the class hierarchy of all stream buffer classes.

Figure 26 -- Hierarchy of the transport layer



The stream buffer classes are responsible for transfer of characters from and to external devices.

The Stream Buffer

This class represents an abstract stream buffer:

©Copyright 1999 Rogue Wave Software, Inc.

```
basic_streambuf<class charT, class traits=char_traits<charT> >
```

It does not have any knowledge about the external device. Instead, it defines two virtual functions, `overflow()` and `underflow()`, to perform the actual transport. These two functions have knowledge of the peculiarities of the external device they are connected to. They must be overwritten by all concrete stream buffer classes, like file and string buffers.

The stream buffer class maintains two character sequences: the *get area*, which represents the input sequence read from an external device, and the *put area*, which is the output sequence to be written to the device. There are functions for providing the next character from the buffer, such as `sgetc()`, etc. They are typically called by the formatting layer in order to receive characters for parsing. Accordingly, there are also functions for placing the next character into the buffer, such as `sputc()`, etc.

A stream buffer also carries a locale object.

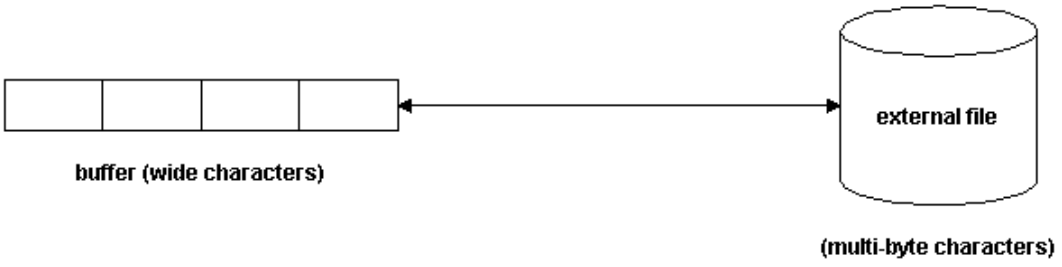
The File Buffer

The file buffer classes associate the input and output sequences with a file. A file buffer takes the form:

```
basic_filebuf<class charT, class traits=char_traits<charT> >
```

The file buffer has functions like `open()` and `close()`. The file buffer class inherits a locale object from its stream buffer base class. It uses the locale's code conversion facet for transforming the external character encoding to the encoding used internally. Figure 27 shows how the file buffer works:

Figure 27 -- Character code conversion performed by the file buffer



The String Stream Buffer

These classes implement the in-memory I/O:

```
basic_stringbuf<class charT, class traits=char_traits<charT> >
```

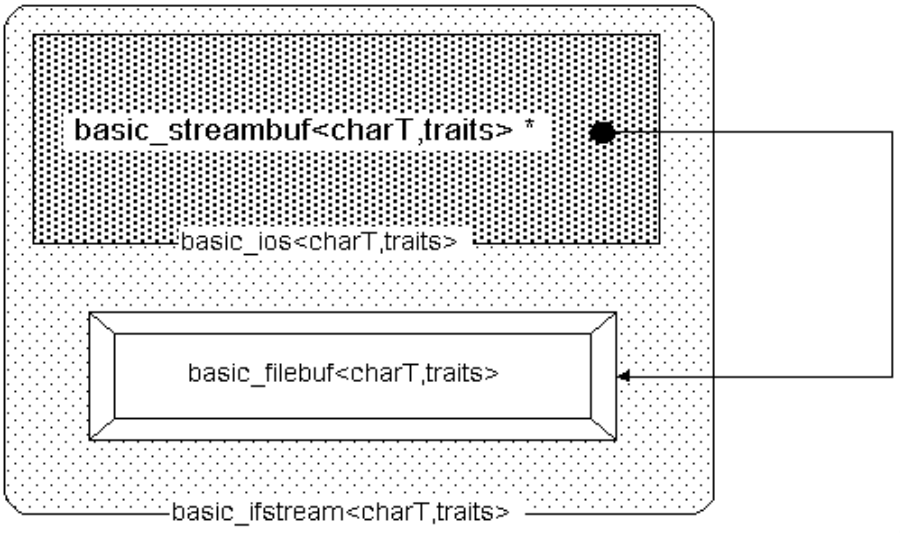
With string buffers, the internal buffer and the external device are one and the same. The internal buffer is dynamic, in that it is extended if necessary to hold all the characters written to it. You can obtain copies of the internally held buffer, and you can provide a string to be copied into the internal buffer.

Collaboration of Streams and Stream Buffers

The base class ***basic_ios<>*** holds a pointer to a stream buffer. The derived stream classes, like file and string streams, contain a file or string buffer object. The stream buffer pointer of the base class refers to this embedded object. This architecture is illustrated in Figure 28:

Figure 28 -- How an input file stream uses a file buffer

©Copyright 1999 Rogue Wave Software, Inc.



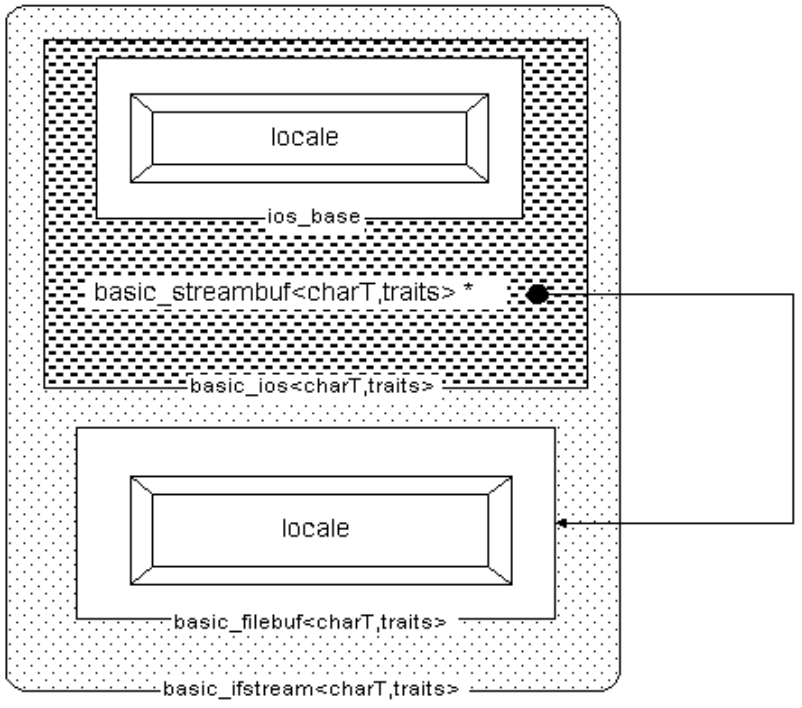
Stream buffers can be used independently of streams, as for unformatted I/O, for example. However, streams always need a stream buffer.

Collaboration of Locales and Iostreams

The base class ***ios_base*** contains a locale object. The formatting and parsing functions defined by the derived stream classes use the *numeric facets* of that locale.

The class ***basic_ios<charT>*** holds a pointer to the stream buffer. This stream buffer has a locale object, too, usually a copy of the same locale object used by the functions of the stream classes. The stream buffer's input and output functions use the *code conversion facet* of the attached locale. Figure 29 illustrates the architecture:

Figure 29 -- How an input file stream uses locales



Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 7: Formatted Input and Output

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

The Predefined Streams

There are eight predefined standard streams that are automatically created and initialized at program start. These standard streams are associated with the C standard files `stdin`, `stdout`, and `stderr`, as shown in Table 5:

Table 5 -- Predefined standard streams with their associated C standard files

Narrow character stream	Wide character stream	Associated C standard files
`cin`	`wcin`	`stdin`
`cout`	`wcout`	`stdout`
`cerr`	`wcerr`	`stderr`
`clog`	`wclog`	`stderr`

Like the C standard files, these streams are all associated by default with the terminal.

The difference between `clog` and `cerr` is that `clog` is fully buffered, whereas output to `cerr` is written to the external device after each formatting. With a fully buffered stream, output to the actual external device is written only when the buffer is full. Thus `clog` is more efficient for redirecting output to a file, while `cerr` is mainly useful for terminal I/O. Writing to the external device after every formatting, to the terminal in the case of `cerr`, serves the purpose of synchronizing output to and input from the terminal.

The standard streams are initialized in such a way that they can be used in constructors and destructors of static objects. Also, the predefined streams are synchronized with their associated C standard files. See Section 14.6 for details.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Input and Output Operators

Now let's try to do some simple input and output to the predefined streams. The iostreams facility defines shift operators for formatted stream input and output. The output operator is the shift operator `operator<<()`, also called the *inserter:*

```
cout << "result: " << x << '\n';
```

Input is done through another shift operator `operator>>()`, often referred to as the *extractor*:

```
cin >> x >> y;
```

Both operators are overloaded for all built-in types in C++, as well as for some of the types defined in the Standard C++ Library; for example, there are inserters and extractors for `bool`, `char`, `int`, `long`, `float`, `double`, `string`, etc. When you insert or extract a value to or from a stream, the C++ function overload resolution chooses the correct extractor operator, based on the value's type. This is what makes C++ iostreams type-safe and better than C stdio (see Chapter 6).

It is possible to print several units in one expression. For example:

```
cout << "result: " << x;
```

is equivalent to:

```
(cout.operator<<("result: ")).operator<<(x);
```

This is possible because each shift operator returns a reference to the respective stream. Almost all shift operators for built-in types are member functions of their respective stream class.1 They are defined according to the following patterns:

```
template<class charT, class traits>
basic_istream<charT, traits>&
basic_istream<charT, traits>::operator>>(type& x)
{
  // read x
  return *this;
}
```

and:

```
template<class charT, class traits>
basic_ostream<charT, traits>&
basic_ostream<charT, traits>::operator<<(type x)
{
  // write x
  return *this;
}
```

Simple input and output of units as shown above is useful, yet not sufficient in many cases. For example, you may want to vary the way output is formatted, or input is parsed. Iostreams allow you to control the formatting features of its input and output operators in many ways. With the standard iostreams, you can specify:

- The width of an output field and the adjustment of the output within this field
- The precision and format of floating point numbers, and whether or not the decimal point should always be included
- Whether you want to skip white spaces when reading from an input stream
- Whether integral values are displayed in decimal, octal or hexadecimal format

and many other formatting options.

There are two mechanisms that have an impact on formatting:

©Copyright 1999 Rogue Wave Software, Inc.

- Formatting control through a stream's format state
- Localization through a stream's locale

The stream's format state is the main means of format control, as we demonstrate in the next section.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Format Control Using the Stream's Format State

Format Parameters

Associated with each stream are a number of *format state variables* that control the details of formatting and parsing. Format state variables are classes inherited from a stream's base class, either ***ios_base*** or ***basic_ios<charT,traits>***. There are two kinds of format parameters: those that can have an arbitrary value, and those that can have only a few different values.

Parameters That Can Have an Arbitrary Value

The value is stored as a private data member in one of the base classes, and set and retrieved through public member functions inherited from that base class. There are three such parameters, described in Table 6:

Table 6 -- Format parameters with arbitrary values

Access function	Defined in base class	Effect	Default
`width()`	***ios_base***	Minimal field width	0
`precision()`	***ios_base***	Precision of floating point values	6
`fill()`	***basic_ios <charT,traits>***	Fill character for padding	The space character

Parameters That Can Have Only a Few Different Values

Typically, these would have just two or three different values. These parameters are represented by one or more bits in a data member of type `fmtflags` in class ***ios_base***. These are usually called *format flags*. You can set format flags using the `setf()` function in class ***ios_base***, clear them using `unsetf()`, and retrieve them through the `flags()` function.

Some format flags are grouped because they are mutually exclusive; for example, output within an output field can be adjusted to the left or to the right, or to an internally specified adjustment. One and only one of the corresponding three format flags, `left`, `right`, or `internal`, can be set.2 If you want to set one of these bits to `1`, you need to set the other two to `0`. To make this easier, there are *bit groups* whose main function is to reset all bits in one group. The bit group for adjustment is `adjustfield`, defined as `left | right | internal`.

Table 7 gives an overview of all format flags and their effects on input and output operators. (For details on how the format flags affect input and output operations, see the *Class Reference* entry for ***ios_base***.) The first column below, *format flag*, lists the flag names; for example, `showpos` stands for `ios_base::showpos`. The *group* column lists the name of the group for flags that are mutually exclusive. The third column gives a brief description of the effect of setting the flag. The *stdio* column refers to format characters used by the C functions `scanf()` or `printf()` that have the same or similar effect. The last column, *default*, lists the setting that is used if you do not explicitly set the flag.

Table 7 -- Flags and their effects on operators

Format flag	Group	Effect	stdio	Default
		Adds fill characters to certain generated output for adjustment:		`left` See footnote [1]
`left`	`adjustfield`	left	`-`	
`right`		right	`0`	
`internal`		Adds fill characters at designated internal point	none	
	`basefield`	Converts integer input or generates integer output in:	`%i`	`dec`
`dec`		decimal base	`%d,%u`	

©Copyright 1999 Rogue Wave Software, Inc.

`oct`		octal base	`%o`	
`hex`		hexadecimal base	`%x`	
		Generates floating point output:	`%g,%G`	`fixed`
`fixed`	`floatfield`	in fixed-point notation	`%f`	
`scientific`		in scientific notation	`%e,%E`	
`boolalpha`		Inserts and extracts `bool` values in alphabetic format		`0`
`showpos`		Generates a `+` sign in non-negative generated numeric output	`+`	`0`
`showpoint`		Always generates a decimal-point in generated floating-point output	`.n` See footnote [2]	`0`
`showbase`		Generates a prefix indicating the numeric base of a generated integer output	`#`	`0`
`skipws`		Skips leading white space before certain input operations	none	`1`
`unitbuf`		Flushes output after each formatting operation	none	`0`
`uppercase`		Replaces certain lowercase letters with their uppercase equivalents in generated output	`%X` `%E` `%G`	`0`

1 Initially, none of the bits is set. This is more or less equivalent to left.

2 Where n indicates the number of decimals

The effect of setting a format parameter is usually permanent; that is, the parameter setting is in effect until the setting is explicitly changed. The only exception to this rule is the field width. The width is automatically reset to its default value, `0`, after each input or output operation that uses the field width. Here is an example:

```
int i; char* s[11];
cin >> setw(10) >> i >> s;                              //1
cout << setw(10) <<i << s;                              //2
```

//1 Extracting an integer is independent of the specified field width. The extractor for integers always reads as many digits as belong to the integer. As extraction of integers does not use the field width setting, the field width of `10` is still in effect when a character sequence is subsequently extracted. Only 10 characters are extracted in this case. After the extraction, the field width is reset to `0`.

//2 The inserter for integers uses the specified field width and fills the field with padding characters if necessary. After the insertion, it resets the field width to `0`. Hence, the subsequent insertion of the string does not fill the field with padding characters for a string with less than 10 characters.

NOTE: With the exception of the field width, all format parameter settings are permanent. The field width parameter is reset after each use.

The following code sample shows how you can control formatting by using some of the parameters:

```
#include <iostream>
using namespace ::std;
// ...
ios_base::fmtflags original_flags = cout.flags();           //1
cout<< 812<<'|';
cout.setf(ios_base::left,ios_base::adjustfield);            //2
cout.width(10);                                             //3
cout<< 813 << 815 << '\n';
cout.unsetf(ios_base::adjustfield);                         //4
cout.precision(2);
cout.setf(ios_base::uppercase|ios_base::scientific);        //5
cout << 831.0 << ` ` << 8e2;
cout.flags(original_flags);                                 //6
```

//1 Store the current format flag setting, in order to restore it later on.

//2 Change the adjustment from the default setting `right` to `left`.

//3 Set the field width from its default `0` to `10`. A field width of `0` means that no padding characters are inserted, and this is the default behavior of all insertions.

//4 Clear the adjustment flags.

//5 Change the precision for floating-point values from its default `6` to `2`, and set yet another couple of format flags that affect floating-point values.

//6 Restore the original flags.

©Copyright 1999 Rogue Wave Software, Inc.

The output is:

```
812|813       815
8.31E+02 8.00E+02
```

Manipulators

Format control requires calling a stream's member functions. Each such call interrupts the respective shift expression. But what if you need to change formats within a shift expression? This is possible in iostreams. Instead of writing:

```
cout<< 812 << '|';
cout.setf(ios_base::left,ios_base::adjustfield);
cout.width(10);
cout<< 813 << 815 << '\n';
```

you can write:

```
cout<< 812 << '|' << left << setw(10) << 813 << 815 << endl;
```

In this example, objects like `left`, `setw`, and `endl` are called *manipulators*. A manipulator is an object of a certain type; let's call the type `manip` for the time being. There are overloaded versions of:

```
basic_istream <charT,traits>:: operator>>()
```

and:

```
basic_ostream <charT,traits>:: operator<<()
```

for type `manip`. Hence a manipulator can be extracted from or inserted into a stream together with other objects that have the shift operators defined. (Section 7.3.2 explains in greater detail how manipulators work and how you can implement your own manipulators.)

The effect of a manipulator need not be an actual input to or output from the stream. Most manipulators set just one of the above described format flags, or do some other kind of stream manipulation. For example, an expression like:

```
cout << left;
```

is equivalent to:

```
cout.setf (ios_base::left, ios_base::adjustfield);.
```

Nothing is inserted into the stream. The only effect is that the format flag for adjusting the output to the left is set.

On the other hand, the manipulator `endl` inserts the newline character to the stream, and flushes to the underlying stream buffer. The expression:

```
cout << endl;
```

is equivalent to:

```
cout << '\n'; cout.flush();
```

Some manipulators take arguments, like `setw(int)`. The `setw` manipulator sets the field width. The expression:

```
cout << setw(10);
```

is equivalent to:

```
cout.width(10);
```

In general, you can think of a manipulator as an object you can insert into or extract from a stream, in order to manipulate that stream. Some manipulators can be applied only to output streams, others only to input streams. Most manipulators change format bits only in one of the stream base classes, ***ios_base*** or ***basic_ios<charT,traits>***. These can be applied to input and output streams.

Table 8 gives an overview of all manipulators defined by iostreams. The first column, **Manipulator**, lists its name. All manipulators are classes defined in the namespace `::std`. The second column, **Use**, indicates whether the manipulator is intended to be used with istreams (`i`), ostreams (`o`), or both (`io`). The third column, **Effect**, summarizes the effect of the manipulator. The last column, **Equivalent**, lists the corresponding call to the stream's member function.

©Copyright 1999 Rogue Wave Software, Inc.

Note that the second column indicates only the *intended* use of a manipulator. In many cases, it is possible to apply an output manipulator to an input stream, and vice versa. Generally, this kind of non-intended manipulation is harmless in that it has no effect. For instance, if you apply the output manipulator `showpoint` to an input stream, the manipulation is simply ignored. However, if you use an output manipulator on a bidirectional stream during input, the manipulation does not affect current input operations, but subsequent output operations.

Table 8 -- Manipulators[1]

Manipulator	Use	Effect	Equivalent
`boolalpha`	`io`	Puts `bool` values in alphabetic format	`io.setf(ios_base::boolalpha)`
`dec`	`io`	Converts integers to/from decimal notation	`io.setf(ios_base::dec, ios_base::basefield)`
`endl`	`o`	Inserts newline and flushes buffer	`o.put(o.widen('\n')); o.flush()`
`ends`	`o`	Inserts end of string character	`o.put(o.widen('\0'))`
`fixed`	`o`	Puts floating point values in fixed-point notation	`o.setf(ios_base::fixed, ios_base::floatfield)`
`flush`	`o`	Flushes stream buffer	`o.flush()`
`hex`	`io`	Converts integers to/from hexadecimal notation	`io.setf(ios_base::hex, ios_base::basefield)`
`internal`	`o`	Adds fill characters at a designated internal point	`o.setf(ios_base::internal, ios_base::adjustfield)`
`left`	`o`	Adds fill characters for adjustment to the left	`o.setf(ios_base::left, ios_base::adjustfield)`
`noboolalpha`	`io`	Resets the above	`io.unsetf(ios_base::boolalpha)`
`noshowbase`	`o`	Resets the above	`o.unsetf (ios_base::showbase)`
`noshowpoint`	`o`	Resets the above	`o.unsetf (ios_base::showpoint)`
`noshowpos`	`o`	Resets the above	`o.unsetf (ios_base::showpos)`
`noskipws`	`i`	Resets the above	`i.unsetf(ios_base::skipws)`
`nounitbuf`	`o`	Resets the above	`o.unsetf(ios_base::unitbuf)`
`nouppercase`		Resets the above	`o.unsetf (ios_base::uppercase)`
`oct`	`io`	Converts to/from octal notation	`io.setf(ios_base::oct, ios_base::basefield)`
`resetiosflags (ios_base::fmtflags mask)`	`io`	Clears ***ios*** flags	`io.setf((ios_base::fmtflags) 0, mask)`
`right`	`o`	Adds fill characters for adjustment to the right	`o.setf(ios_base::right, ios_base::adjustfield)`
`scientific`		Puts floating point values in scientific notation	`o.setf(ios_base::scientific, ios_base::floatfield)`
`setbase (int base)`	`io`	Sets base for integer notation (base = 8, 10, 16)	`io.setf (base == 8?ios_base::oct: base == 10 ? ios_base::dec : base == 16 ? ios_base::hex : ios_base::fmtflags(0), ios_base::basefield)`
`setfill(charT c)`	`io`	Sets fill character for padding	`io.fill(c)`
`setiosflags (ios_base::fmtflags mask)`	`io`	Sets ***ios*** flags	`io.setf(mask)`
`setprecision (int n)`	`io`	Sets precision of floating point values	`io.precision(n)`
`setw(int n)`	`io`	Sets minimal field width	`io.width(n)`
`showbase`	`o`	Generates a prefix indicating the numeric base of an integer	`o.setf(ios_base::showbase)`
`showpoint`	`o`	Always generates a decimal-point for floating-point values	`o.setf(ios_base::showpoint)`
`showpos`	`o`	Generates a + sign for non-negative numeric values	`o.setf(ios_base::showpos)`

©Copyright 1999 Rogue Wave Software, Inc.

`skipws`	`i`	Skips leading white space	`i.setf(ios_base::skipws)`
`unitbuf`	`o`	Flushes output after each formatting operation	`o.setf(ios_base::unitbuf)`
`uppercase`	`o`	Replaces certain lowercase letters with their uppercase equivalents	`o.setf(ios_base::uppercase)`
`ws`	`i`	Skips white spaces	

The Rogue Wave implementation of the Standard C++ Library also specifies the non-standard manipulators:

1 `__lock` locks the stream for multithread use;
`__unlock` unlocks the stream for multithread use.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Localization Using the Stream's Locale

Associated with each stream is a locale that impacts the parsing and formatting of numeric values. This is how localization of software takes place. As discussed in Section 2.2.2, the representation of numbers often depends on cultural conventions. In particular, the *decimal point* need not be a period, as in the following example:

```
cout.imbue(locale("De_DE"));
cout << 1000000.50 << endl;
```

The output is:

```
1000000,50
```

Other cultural conventions, like the grouping of digits, are irrelevant. There is no formatting of numeric values that involves grouping.3

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Formatted Input

In principle, input and output operators behave symmetrically. There is only one important difference: for output you control the precise format of the inserted character sequence, while for input the format of an extracted character sequence is never exactly described. This is for practical reasons. You may want to extract the next floating point value from a stream, for example, without necessarily knowing its exact format. You want it whether it is signed or not, or in exponential notation with a small or capital E for the exponent, etc. Hence, extractors in general accept an item in any format permitted for its type.

Formatted input is handled as follows:

1. Extractors automatically ignore all white space characters (blanks, tabulators, newlines 4) that precede the item to be extracted.

2. When the first relevant character is found, they extract characters from the input stream until they find a separator; that is, a character that does not belong to the item. White space characters in particular are separators.

3. The separator remains in the input stream and becomes the first character extracted in a subsequent extraction.

Several format parameters, which control insertion, are irrelevant for extraction. The format parameter fill character, `fill()`, and the adjustment flags, `left`, `right`, and `internal`, have no effect on extraction. The field width is relevant only for extraction of strings, and ignored otherwise.

Skipping Characters

You can use the manipulator `noskipws` to `switch off the automatic skipping of white space characters. For example, extracting white space characters may be necessary if you expect the input has a certain format, and you need to check for violations of the format requirements. This procedure is shown in the following code:`

```
cin >> noskipws;
char c;
do
{ float fl;
  c = ' '; cin >> fl >> c;  // extract number and separator
if (c == ',' || c == '\n')  // next char is ',' or newline ?
    process(fl);            // yes: use the number
}
while (c == ',');
if (c != '\n') error();     // no: error!
```

If you must skip a sequence of characters other than white spaces, you can use the `istream`'s member function `ignore()`. The call:

```
basic_ifstream<myChar,myTraits> InputStream("file-name");
InputStream.ignore(numeric_limits<streamsize>::max()
                   ,myChar('\n'));
```

or, for ordinary tiny characters of type `char`:

```
ifstream InputStream("file-name");
InputStream.ignore(INT_MAX,'\n');
```

ignores all characters until the end of the line. This example uses a file stream that is not predefined. File streams are described in Section 6.4.1.5.

Input of Strings

When you extract strings or character arrays from an input stream, characters are read until:

- A white space character is found

©Copyright 1999 Rogue Wave Software, Inc.

- The end of the input is reached
- A certain number of characters is extracted, if `width() != 0;` . In the case of a string, this number is the field width `width()`. In the case of a character array, this number is `width()-1`.

Note that the field width is reset to `0` after the extraction of a string.

There are subtle differences between extracting a character sequence into a character array and extracting it into a string object. For example:

```
char buf[SZ];
cin >> buf;
```

is different from

```
string s;
cin >> s;
```

Extraction into a string is safe, because strings automatically extend their capacity as necessary. You can extract as many characters as you want since the string always adjusts its size accordingly. Character arrays, on the other hand, have fixed size and cannot dynamically extend their capacity. If you extract more characters than the character array can take, the extractor writes beyond the end of the array. To prevent this, you must set the field width as follows each time you extract characters into a character array:

```
char buf[SZ];
cin >> width(SZ) >> buf;
```

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 8: Error State of Streams

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

About Flags

It probably comes as no surprise that streams have an error state. When an error occurs, flags are set in the state according to the general category of the error. Flags and their error categories are summarized in Table 9:

Table 9 -- Flags and corresponding error categories

iostate flag	Error category
`ios_base::goodbit`	Everything's fine
`ios_base::eofbit`	An input operation reached the end of an input sequence
`ios_base::failbit`	An input operation failed to read the expected character, or an output operation failed to generate the desired characters
`ios_base::badbit`	Indicates the loss of integrity of the underlying input or output sequence

Note that the flag `ios_base::goodbit` is not really a flag; its value, `0`, indicates the absence of any error flag. It means the stream is OK. By convention, all input and output operations have no effect once the stream state is different than 0.

There are several situations when both `eofbit` and `failbit` are set; however, the two have different meanings and do not always occur in conjunction. The flag `ios_base::eofbit` is set when there is an attempt to read past the end of an input sequence. This occurs in the following two typical examples:

1. Assume the extraction happens character-wise. Once the last character is read, the stream is still in good state; `eofbit` is not yet set. Any subsequent extraction, however, is considered an attempt to read past the end of the input sequence. Thus, `eofbit` is set.

2. If you do not read character-wise, but extract an integer or a string, for example, you always read past the end of the input sequence. This is because the input operators read characters until they find a separator, or hit the end of the input sequence. Consequently, if the input contains the sequence `... 912749<eof>` and an integer is extracted, `eofbit` is set.

The flag `ios_base::failbit` is set as the result of a read or write operation that fails. For example, if you try to extract an integer from an input sequence containing only white spaces, the extraction of an integer fails, and the `failbit` is set. Let's see whether `failbit` would be set in the previous examples:

1. After reading the last available character, the extraction not only reads past the end of the input sequence; it also fails to extract the requested character. Hence, `failbit` is set in addition to `eofbit`.

2. Here it is different. Although the end of the input sequence is reached by extracting the integer, the input operation does not fail and the desired integer is indeed read. Hence, in this situation only the `eofbit` is set.

In addition to these input and output operations, there are other situations that can trigger failure. For example, file streams set `failbit` if the associated file cannot be opened (see Section 9.2.2).

The flag `ios_base::badbit` indicates problems with the underlying stream buffer. These problems could be:

- ***Memory shortage***. There is no memory available to create the buffer, or the buffer has size `0` for other reasons[5], or the stream cannot allocate memory for its own internal data, as with `iword` and `pword`.

- ***The underlying stream buffer throws an exception***. The stream buffer might lose its integrity, as in memory shortage, or code conversion failure, or an unrecoverable read error from the external device. The stream buffer can indicate this loss of integrity by throwing an exception, which is caught by the stream and results in setting the `badbit` in the stream's state.

Generally, you should keep in mind that `badbit` indicates an error situation that is likely to be unrecoverable, whereas `failbit` indicates a situation that might allow you to retry the failed operation. The flag `eofbit` simply indicates the end of the input sequence.

©Copyright 1999 Rogue Wave Software, Inc.

What can you do to check for such errors? You have two possibilities for detecting stream errors:

- You can declare that you want to have an exception raised once an error occurs in any input or output operation.
- You can actively check the stream state after each input or output operation.

We explore these possibilities in the next two sections.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Checking the Stream State

Let's first examine how you can check for errors using the stream state. A stream has several member functions for this purpose, which are summarized with their effects in Table 10:

Table 10 -- Stream member functions for error checking

ios_base member function	Effect
`bool good()`	`TRUE` if no error flag is set
`bool eof()`	`TRUE` if `eofbit` is set
`bool fail()`	`TRUE` if `failbit` or `badbit` is set
`bool bad()`	`TRUE` if `badbit` is set
`bool operator!()`	As `fail()`
`operator void*()`	Null pointer if `fail()` and non-null value otherwise
`iostate rdstate()`	Value of stream state

It is a good idea to check the stream state in some central place, for example:

```
if (!cout) error();
```

The state of `cout` is examined with `operator!()`, which returns `TRUE` if the stream state indicates that an error occurred.

An ostream can also appear in a boolean position to be tested as follows:

```
if (cout << x) // okay!
```

The magic here is the `operator void*()` that returns a nonzero value when the stream state is nonzero.

Finally, the explicit member functions can also be used:

```
if (cout << x, cout.good()) // okay!;
```

Note that there is a difference between `good()` and `operator!()`. The function `good()` takes all flags into account; `operator!()` and `fail()` ignore `eofbit`.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Catching Exceptions

By default a stream does not throw any exceptions.[6] You must explicitly activate an exception because a stream contains *an exception mask*. Each flag in this mask corresponds to one of the error flags. For example, once the `badbit` flag is set in the exception mask, an exception is thrown each time the `badbit` flag gets set in the stream state. The following code demonstrates how to activate an exception on an input stream `InStr`:

```
try {
   InStr.exceptions(ios_base::badbit | ios_base::failbit);    //1
   in >> x;
   // do lots of other stream i/o
}
catch(ios_base::failure& exc)                                 //2
{   cerr << exc.what() << endl;
    throw;
}
```

//1 In calling the `exceptions()` function, you indicate what flags in the stream's state shall cause an exception to be thrown.[7]

//2 Objects thrown by the stream's operations are of types derived from `ios_base::failure`. Hence this catch clause catches all stream exceptions in principle. We qualify this generalization because a stream might fail to catch certain exceptions like `bad_alloc`, for example, so that exceptions other than `ios_base::failure` might be raised. That's how exception handling in C++ works: you never know what exceptions will be raised.

Generally, it is a good idea to activate the `badbit` exception and suppress the `eofbit` and `failbit` exceptions, because the latter do not represent exceptional states. A `badbit` situation, however, is likely to be a serious error condition similar to the memory shortage indicated by a `bad_alloc` exception. Unless you want to suppress exceptions thrown by iostreams altogether, we would recommend that you switch on the `badbit` exception and turn off `eofbit` and `failbit`.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 9: File Input and Output

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

About File Streams

File streams allow input and output to files. Unlike the C stdio functions for file I/O, however, file streams follow Stroustrup's idiom: "Resource acquisition is initialization."[8] In other words, file streams provide an advantage in that you can open a file on construction of a stream, and the file is closed automatically on destruction of the stream. Consider the following code:

```
void use_file(const char* fileName)
{
   FILE* f = fopen("fileName", "w");
   // use file
   fclose(f);
}
```

If an exception is thrown while the file is in use here, the file is never closed. With a file stream, however, the file is closed whenever the file stream goes out of scope, as in the following example:

```
void use_file(const char* fileName)
{
   ofstream f("fileName");
   // use file
}
```

Here the file is closed even if an exception occurs during use of the open file.

There are three class templates that implement file streams: ***basic_ifstream <charT,traits>***, ***basic_ofstream <charT,traits>***, and ***basic_fstream <charT,traits>***. These templates are derived from the stream base class ***basic_ios <charT, traits>***. Therefore, they inherit all the functions for formatted input and output described in Chapter 7, as well as the stream state. They also have functions for opening and closing files, and a constructor that allows opening a file and connecting it to the stream. For convenience, there are the regular typedefs `ifstream`, `ofstream`, and `fstream`, with `wifstream`, `wofstream`, and `wfstream` for the respective tiny and wide character file streams.

The buffering is done through a specialized stream buffer class, ***basic_filebuf <charT,traits>***.

The Difference between Predefined Streams and File Streams

The main differences between a predefined standard stream and a file stream are:

- A file stream needs to be connected to a file before it can be used. The predefined streams can be used right away, even in static constructors that are executed before the `main()` function is called.
- You can reposition a file stream to arbitrary file positions. This usually does not make any sense with the predefined streams, as they are connected to the terminal by default.

Code Conversion in Wide Character Streams

In a large character set environment, a file is assumed to contain multibyte characters. To provide the contents of a such a file as a wide character sequence for internal processing, `wifstream` and `wofstream` perform corresponding conversions. The actual conversion is delegated to the file buffer, which relays the task to the imbued locale's code conversion facet.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Working with File Streams

Creating and Opening File Stream Objects

There are two ways to create a file stream9: you can create an empty file stream, open a file, and connect it to the stream later on; or you can open the file and connect it to a stream at construction time. These two procedures are demonstrated in the two following examples, respectively:

```
ifstream file;                                          //1
...;
file.open(argv[1]);                                     //2
if (!file) // error: unable to open file for input
```

or:

```
ifstream source("src.cpp");                             //3
if (!source) // error: unable to open src.cpp for input
```

//1 A file stream is created that is not connected to any file. Any operation on the file stream fails.

//2 Here a file is opened and connected to the file stream. If the file cannot be opened, `ios_base::failbit` is set; otherwise, the file stream is now ready for use.

//3 Here the file is both opened and connected to the stream.

Checking a File Stream's Status

Generally you can check whether the attempt to open a file was successful by examining the stream state afterwards; `failbit` is set in case of failure.

There is also a function called `is_open()` that indicates whether a file stream is connected to an open file. This function does *not* mean that a previous call to `open()` was successful. To understand the subtle difference, consider the case of a file stream that is already connected to a file. Any subsequent call to `open()` fails, but `is_open()` still returns `TRUE`, as shown in the following code:

```
void main(int argc, char* argv[])
{
  if (argc > 2)
  {  ofstream fil;                                      //1
     fil.open(argv[1]);
     // ...
     fil.open(argv[2]);                                 //2
     if (fil.fail())                                    //3
     {  // open failed  }
     if (fil.is_open())                                 //4
     {  // connected to an open file  }
  }
}
```

//1 Open a file and connect the file stream to it.

//2 Any subsequent open on this stream fails.

//3 Hence the `failbit` is set.

//4 However, `is_open()` still returns `TRUE`, because the file stream still is connected to an open file.

Closing a File Stream

In the example above, it would be advisable to close the file stream before you try to connect it to another file. This is done implicitly by the file streams destructor in the following code:

```
void main(int argc, char* argv[])
{
  if (argc > 2)
```

©Copyright 1999 Rogue Wave Software, Inc.

```
  {  ofstream fil;
     fil.open(argv[1]);
     // ...
  }                                                    //1
  {  ofstream fil;
     fil.open(argv[2]);
    // ...
  }
}
```

//1 Here the file stream `fil` goes out of scope and the file it is connected to is closed automatically.

You can explicitly close the connected file. The file stream is then empty, until it is reconnected to another file:

```
ifstream f;                                            //1
for (int i=1; i<argc; ++i)
{
    f.open(argv[i]);                                   //2
    if (f)                                             //3
    {
        process(f);                                    //4
        f.close();                                     //5
    }
    else
        cerr << "file " << argv[i] << " cannot be opened.\n";
}
```

//1 An empty file stream is created.

//2 A file is opened and connected to the file stream.

//3 Here we check whether the file was successfully opened. If the file could not be opened, the `failbit` would be set.

//4 Now the file stream is usable, and the file's content can be read and processed.

//5 Close the file again. The file stream is empty again.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

The Open Mode

There may be times when you want to modify the way a file is opened or used in a program. For example, in some cases it is desirable that writes append to the end of a file rather than overwriting the existing values. The file stream constructor takes a second argument, the *open mode*, that allows such variations to be specified. Here is an example:

```
fstream Str("inout.txt",
            ios_base::in|ios_base::out|ios_base::app);
```

The Open Mode Flags

The open mode argument is of type `ios_base::openmode`, which is a bitmask type like the format flags and the stream state. Table 11 defines the following bits:

Table 11 -- Flag names and effects

Flag Names	Effects
`ios_base::in`	Open file for reading
`ios_base::out`	Open file for writing
`ios_base::ate`	Start position is at file end
`ios_base::app`	Append file; that is, always write to the end of the file
`ios_base::trunc`	Truncate file; that is, delete file content
`ios_base::binary`	Binary mode

The in and out Open Modes

Input (and output) file streams always have the `in` (or `out`) open mode flag set implicitly. An output file stream, for instance, knows that it is in output mode and you need not set the output mode explicitly. Instead of writing:

```
ofstream Str("out.txt",ios_base::out|ios_base::app);
```

you can simply say:

```
ofstream Str("out.txt",ios_base::app);
```

Bidirectional file streams, on the other hand, do not have the flag set implicitly. This is because a bidirectional stream does not have to be in both input and output mode in all cases. You might want to open a bidirectional stream for reading only or writing only. Bidirectional file streams therefore have no implicit input or output mode. You must always set a bidirectional file stream's open mode explicitly.

The Open modes ate, app, and trunc

Each file maintains a *file position* that indicates the position in the file where the next byte will be read or written. When a file is opened, the initial file position is usually at the beginning of the file. The open modes `ate` (meaning *at end*) and `app` (meaning *append*) change this default to the end of the file.

There is a subtle difference between `ate` and `app` mode. If the file is opened in append mode, all output to the file is done at the current end of the file, regardless of intervening repositioning. Even if you modify the file position to a position before the file's end, you cannot write there. With `at-end` mode, you can navigate to a position before the end of file and write to it.

If you open an already existing file for writing, you usually want to overwrite the content of the output file. The open mode `trunc` (meaning *truncate*) has the effect of discarding the file content, in which case the initial file length is set to `0`. Therefore, if you want to replace a file's content rather than extend the file, you must open the file in `out|trunc`.[10] Note that the file position is at the beginning of the file in this case, which is exactly what you expect for overwriting an output file.

©Copyright 1999 Rogue Wave Software, Inc.

If you want to extend an output file, you open it with the logical `or` of `at end` and `append` mode. In this case, the file content is retained because the `trunc` flag is not set, and the initial file position is at the file's end. However, you may additionally set the `trunc` flag; the file content is discarded and the output is done at the end of an empty file.

Input mode only works for files that already exist. Otherwise, the stream construction fails, as indicated by `failbit` set in the stream state. Files that are opened for writing are created if they do not yet exist. The constructor only fails if the file cannot be created.

The binary Open Mode

The `binary` open mode is discussed in Section 9.4.

Combining Open Modes

The effect of combining these open modes is similar to the mode argument of the C library function `fopen(name,mode)`. Table 12 gives an overview of all permitted combinations of open modes for text files and their counterparts in C stdio. Combinations of modes that are not listed in the table (such as both `trunc` and `app`) are invalid, and the attempted `open()` operation fails.

Table 12 -- Open modes and their C stdio counterparts

Open Mode	C stdio Equivalent	Effect
`in`	`"r"`	Open text file for reading only
`out|trunc` `out`	`"w"`	Truncate to `0` length, if existent, or create text file for writing only
`out|app`	`"a"`	Append; open or create text file only for writing at end of file
`in|out`	`"r+"`	Open text file for update (reading and writing)
`in|out|trunc`	`"w+"`	Truncate to `0` length, if existent, or create text file for update
`in|out|app`	`"a+"`	Append; open or create text file for update, writing at end of file

Default Open Modes

The open mode parameter in constructors and `open()` functions of file stream classes have a default value. The default open modes are listed in Table 13. Note that abbreviations are used; for example, `ifstream` stands for `basic_ifstream <charT,traits>`.

Table 13 -- Default open modes

File Stream	Default Open Mode
`ifstream`	`in`
`ofstream`	`out`
`fstream`	`in|out`

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Binary and Text Mode

The representation of text files varies among operating systems. For example, the end of a line in a UNIX environment is represented by the *linefeed* character `'\n'`. On PC-based systems, the end of the line consists of two characters, *carriage return* `'\r'` and *linefeed* `'\n'`. The end of the file differs as well on these two operating systems. Peculiarities on other operating systems are also conceivable.

To make programs more portable among operating systems, an automatic conversion can be done on input and output. The carriage return or linefeed sequence, for example, can be converted to a single `'\n'` character on input; the `'\n'` can be expanded to `"\r\n"` on output. This conversion mode is called *text mode*, as opposed to *binary mode*. In binary mode, no such conversions are performed.

The mode flag `ios_base::binary` has the effect of opening a file in binary mode. This has the effect described above; in other words, all automatic conversions, such as converting `"\r\n"` to `'\n'`, are suppressed. 11

If you must process a binary file, you should always set the `binary` mode flag, because most likely you do not want any kind of implicit, system-specific conversion performed.

The effect of the binary open mode is frequently misunderstood. It does *not* put the inserters and extractors into a binary mode, and hence suppress the formatting they usually perform. Binary input and output is done solely by `basic_istream <charT> ::read()` and `basic_ostream <charT> ::write()`.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

File Positioning

Iostream support for file positioning operations depends on the character encoding of a particular stream. For fixed-width encodings, such as ASCII or UNICODE, the file stream classes allow a full set of positioning operations comparable to those offered by `C' stdio. The options for variable-width and state-dependent encodings, such as JIS, are much more limited. For these more complex encodings, the only allowed positioning operations are 'seek to beginning', 'seek to end', or 'seek to a previously known location'. The last option requires that you arrive at and then store a particular position before a seek can be performed. Attempting to seek to an arbitrary offset on a stream with variable-width or state-dependent character encodings has no effect on the file position. Here's an example of the correct way to seek to a position in any file stream, regardless of encoding:

```
int main(int argc, char argv[])
{
  ofstream fs("foo.out");
  fs << "Anyone";                        //1
  ofstream::pos_type p = fs.tellp();     //2
  fs << " remember J.P. Patches?";       //3
  fs.seekp(p);                           //4

  return 0;
}
```

//1 Here we output some characters in order to move the file position to some arbitrary location that we'll later seek back to.

//2 Now we use the `tellp()` function to save the current file position.

//3 Output some more text to move the file position along.

//4 Finally, seek back to our previously saved position.

This is the only possible method for seeking to an *arbitrary position*--that is, a position other than beginning or end of file--in a stream with variable or state-dependent character encoding. Of course, the method also works with fixed-width encodings.

The example above shows one use of two of the output file positioning functions, `tellp` and a version of `seekp`. An ***ofstream*** also has another version of the `seekp` function that allows a seek to an arbitrary offset in much the same way that the `C' stdio `fseek` function works. This function can be used to seek to the beginning or end of any ***ofstream***, or anywhere else in an ***ofstream*** with a fixed-width character encoding. For instance:

```
ofstream fs("foo.out");
fs << "Anyone remember J.P. Patches?";
fs.seekp(-2,ios_base::cur);             //1
fs.seekp(0,ios_base::beg);              //2
```

//1 Seek back two characters. Position at the `s` in `Patches`.

//2 Seek to beginning of the file.

The first parameter of this function is an `ofstream::off_type`, and the second is one of three constants indicating starting position for the seek. These three values correspond to the three possible seek types available with the `C' stdio function `fseek`. They are defined in the base class `ios_base`. The table below summarizes the three different kinds of seeks possible with this version of `seekp`:

Table 14 -- Possible seeks for seekp

Type of seek	Argument to seekp	`C' stdio equivalent
seek from beginning of file	`ios_base::beg`	`SEEK_SET`
seek from end of file	`ios_base::end`	`SEEK_END`
seek from current position	`ios_base::cur`	`SEEK_CUR`

©Copyright 1999 Rogue Wave Software, Inc.

As in the example, passing `0` as the offset with `ios_base::beg` as the seek type seeks to the beginning of the file. Likewise, using `0` with `ios_base::end` seeks to the end of the file. Since the function returns the current position after the seek operation, passing `0` along with `ios_base::cur` gets you the current file position without moving it. This is equivalent to calling the `tellp()` function.

The ***ifstream*** class provides the same set of functions but with slightly different names: `tellg()` instead of `tellp()`, and `seekg(...)` instead of `seekp(...)`. The reason for this specialized naming scheme can be seen in the ***fstream*** class, which provides both sets of functions so that the input and output streams can be manipulated separately.

How Positioning Works with the Iostream Architecture

If you look at the ***iostream*** class definitions, you notice that the seek functions are not defined in these classes. Instead, they are obtained from a base class: ***basic_ostream*** for `tellp` and `seekp`, and ***basic_istream*** for `tellg` and `seekg`. These functions then call virtual functions in the stream buffer, where seeking is actually implemented. Seek functions for ***ofstream***, ***ifstream***, and ***fstream*** actually call ***seekoff*** and ***seekpos*** in ***filebuf***. The code looks like this:

```
basic_ofstream
basic_ostream::seekp(pos)

    ->basic_streambuf::pubseekpos(pos)

        ->(virtual)   basic_filebuf::seekpos(pos)
```

Calling virtual functions in the stream buffer maintains the fundamental separation of buffer manipulation and I/O from string formatting. While it's not necessary to know this in order to use file seek operations, it is important if you ever need to subclass a stream buffer.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 10: Input and Output In Memory

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

About String Streams

The iostreams facility supports not only input and output to external devices like files. It also allows in-memory parsing and formatting. Source and sink of the characters read or written becomes a string held somewhere in memory. You use in-memory I/O if the information to be read is already available in the form of a string, or if the formatted result is processed as a string. For example, to interpret the contents of the string `argv[1]` as an integer value, the code might look like this:

```
int i;
if (istringstream(argv[1]) >> i)                              //1
   // use the value of i
```

//1 The parameter of the input string stream constructor is a string; here a character array, namely `argv[1]`, is provided as an argument and is implicitly converted to a string. From this newly constructed input string stream, which contains `argv[1]`, an integer value is extracted.

The inverse operation, taking a value and converting it to characters that are stored in a string, might look like this:

```
struct date {
  int day,month,year;
} today = {8,4,1996};
ostringstream ostr;                                            //1
ostr << today.month << '-' << today.day <<'-' << today.year;  //2
if (ostr)
   display(ostr.str());                                        //3
```

//1 An output string stream is allocated.

//2 Values are inserted into the output string stream.

//3 The result of the formatting can be retrieved in the form of a string, which is returned by `str()`.

As with file streams, there are three class templates that implement string streams: ***basic_istringstream <charT,traits,Allocator>***, ***basic_ostringstream <charT,traits,Allocator>***, and ***basic_stringstream <charT,traits,Allocator>***. These are derived from the stream base classes, ***basic_istream <charT, traits>***, ***basic_ostream <charT, traits>***, and ***basic_iostream <charT, traits>***. Therefore they inherit all the functions for formatted input and output described in Chapter 7, as well as the stream state. They also have functions for setting and retrieving the string that serves as source or sink, and constructors that allow you to set the string before construction time. For convenience, there are the regular typedefs `istringstream`, `ostringstream`, and `stringstream`, with `wistringstream`, `wostringstream`, and `wstringstream` for the respective tiny and wide character string streams.

The buffering is done through a specialized stream buffer class, ***basic_stringbuf <charT,traits,Allocator>***.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The Internal Buffer

String streams can take a string, provided either as an argument to the constructor, or set later through the `str(const basic_string <charT, traits, Allocator>&)` function. This string is copied into an internal buffer, and serves as source or sink of characters to subsequent insertions or extractions. Each time the string is retrieved through the `str()` function, a copy of the internal buffer is created and returned.

Output string streams are *dynamic*.12 The internal buffer is allocated once an output string stream is constructed. The buffer is automatically extended during insertion each time the internal buffer is full.

Input string streams are always *static*. You can extract as many items as are available in the string you provided the string stream.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

The Open Modes

The only open modes that have an effect on string streams are `in`, `out`, `ate`, and `app`. They have more or less the same meaning that they have with file streams (see Section 9.3). The `binary` open mode is irrelevant, because there is no conversion to and from the dependent file format of the operating system. The `trunc` open mode is simply ignored.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 11: Input and Output of User Types

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

A Note on User-Defined Types

One of the major advantages of the iostreams facility is extensibility. Just as you have inserters and extractors for almost all types defined by the C++ language and library, you can implement extractors and inserters for all your own user-defined types. To avoid surprises, the input and output of user-defined types should follow the same conventions used for insertion and extraction of built-in types. In this chapter, we give guidelines for building a typical extractor and inserter for a user-defined type.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

An Example with a User-Defined Type

Let us work through a complete example with the following date class as the user-defined type:

```
class date {
public:
   date(int d, int m, int y);
   date(const tm& t);
   date();
   // more constructors and useful member functions
private:
   tm tm_date;
};
```

This class has private data members of type `tm`, which is the time structure defined in the C library (in header file `<ctime>`).

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

A Simple Extractor and Inserter for the Example

Following the read and write conventions of iostreams, we would insert and extract the date object like this:

```
date birthday(2,6,1952);
cout << birthday << '\n';
```

or

```
date aDate;

cout << '\n' << "Please, enter a date (day month year)" << '\n';
cin  >> aDate;
cout << aDate << '\n';
```

For the next step, we would implement shift operators as inserters and extractors for `date` objects. Here is an extractor for class `date`:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&                              //1
operator>> (basic_istream<charT,Traits>& is,               //2
            date& dat)                                     //3
{
 is >> dat.tm_date.tm_mday;                                //4
 is >> dat.tm_date.tm_mon;
 is >> dat.tm_date.tm_year;
 return is;                                                //5
}
```

//1 The returned value for extractors (and inserters) is a reference to the stream, so that several extractions can be done in one expression.

//2 The first parameter usually is the stream from which the data shall be extracted.

//3 The second parameter is a reference, or alternatively a pointer, to an object of the user-defined type.

//4 In order to allow access to private data of the date class, the extractor must be declared as a friend function in class `date`.

//5 The return value is the stream from which the data was extracted.

As the extractor accesses private data members of class date, it must be declared as a friend function to class date. The same holds for the inserter. Here's a more complete version of class `date`:

```
class date {
public:
   date(int d, int m, int y);
   date(tm t);
   date();
   // more constructors and useful member functions

private:
   tm tm_date;

template<class charT, Traits>
friend basic_istream<charT, Traits> &operator >>
  (basic_istream<charT, Traits >& is, date& dat);

template<class charT, Traits>
friend basic_ostream<charT, Traits> &operator <<
  (basic_ostream<charT, Traits >& os, const date& dat);
};
```

The inserter can be built analogously, as shown below. The only difference is that you would hand over a *constant* reference to a date object, because the inserter is not supposed to modify the object it prints.

```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
```

©Copyright 1999 Rogue Wave Software, Inc.

```
operator << (basic_ostream<charT, Traits >& os, const date& dat)
{
 os << dat.tm_date.tm_mon << '-';
 os << dat.tm_date.tm_mday << '-';
 os << dat.tm_date.tm_year ;
 return os;
}
```

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Improved Extractors and Inserters

The format of dates depends on local and cultural conventions. Naturally, we want our extractor and inserter to parse and format the date according to such conventions. To add this functionality to them, we use the time facet contained in the respective stream's locale as follows:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&
operator >> (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;

 use_facet<time_get<charT,Traits> >(is.getloc())              //1
   .get_date(is, istreambuf_iterator<charT,Traits>()          //2
             ,is, err, &dat.tm_date);                         //3

 return is;
}
```

//1 Use the `time_get` facet of the input stream's locale to handle parsing of dates according to cultural conventions defined by the locale. The locale in question is obtained through the stream's `getloc()` function. Its `time_get` facet is accessed through a call to the global `use_facet<..>()` function. The type argument to the `use_facet` function template is the facet type. (See the chapter on internationalization for more details on locales and facets.).

The facet's member function `get_date()` is called. It takes a number of arguments, including:

//2 *A range of input iterators.* For the sake of performance and efficiency, facets directly operate on a stream's buffer. They access the stream buffer through stream buffer iterators. (See the section on stream buffer iterators in the *Standard C++ Library User's Guide*.) Following the philosophy of iterators in the Standard C++ Library, we must provide a range of iterators. The range extends from the iterator pointing to the first character to be accessed, to the character past the last character to be accessed (the *past-the-end-position*).

The beginning of the input sequence is provided as a reference to the stream. The `istreambuf_iterator` class has a constructor taking a reference to an input stream. Therefore, the reference to the stream is automatically converted into an `istreambuf_iterator` that points to the current position in the stream. As end of the input sequence, an end-of-stream iterator is provided. It is created by the default constructor of class `istreambuf_iterator`. With these two stream buffer iterators, the input is parsed from the current position in the input stream until a date or an invalid character is found, or the end of the input stream is reached.

The other parameters are:

Formatting flags. A reference to the `ios_base` part of the stream is provided here, so that the facet can use the stream's formatting information through the stream's members `flags()`, `precision()`, and `width()`.

//3 *An iostream state*. It is used for reporting errors while parsing the date.

A pointer to a time object. It must be a pointer to an object of type `tm`, which is the time structure defined by the C library. Our date class maintains such a time structure, so we hand over a pointer to the respective data member `tm_date`.

The inserter is built analogously:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const date& dat)
{
   use_facet
   <time_put<charT,ostreambuf_iterator<charT,Traits> > >      //1
   (os.getloc())
   .put(os,os,os.fill(),&dat.tm_date,'x');                    //2
   return os;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

//1 Here we use the `time_put` facet of the stream's locale to handle formatting of dates. The facet's `put()` function takes the following arguments:

An output iterator. We use the automatic conversion from a reference to an output stream to an `ostreambuf_iterator`. This way the output is inserted into the output stream, starting at the current write position.

The formatting flags. Again we provide a reference to the `ios_base` part of the stream to be used by the facet for retrieving the stream's formatting information.

//2 *The fill character*. We would use the stream's fill character here. Naturally, we could use any other fill character; however, the stream's settings are normally preferred.

A pointer to a time structure. This structure is filled with the result of the parsing.

A format specifier. This can be a character, like `'x'` in our example here, or alternatively, a character sequence containing format specifiers, each consisting of a `%` followed by a character. An example of such a format specifier string is `"%A, %B %d, %Y"`. It has the same effect as the format specifiers for the `strftime()` function in the C library; it produces a date like: `Tuesday, June 11, 1996`. We don't use a format specifier string here, but simply the character `'x'`, which specifies that the locale's appropriate date representation shall be used.

Note how these versions of the inserter and extractor differ from previous simple versions: we no longer rely on existing inserters and extractors for built-in types, as we did when we used `operator<<(int)` to insert the `date` object's data members individually. Instead, we use a low-level service like the `time` facet's `get_date()` service. The consequence is that we give away all the functionality that high-level services like the inserters and extractors already provide, such as format control, error handling, etc.

The same happens if you decide to access the stream's buffer directly, perhaps for optimizing your program's runtime efficiency. The stream buffer's services, too, are low-level services that leave to you the tasks of format control, error handling, etc.

In the following sections, we explain how you can improve and complete your inserter or extractor if it directly uses low-level components like locales or stream buffers.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

More Improved Extractors and Inserters

Insertion and extraction still do not fit seamlessly into the iostream framework. The inserters and extractors for built-in types can be controlled through formatting flags that our operators thus far ignore. Our operators don't observe a field width while inserting, or skip whitespaces while extracting, and so on.

They don't care about error indication either. So what if the extracted date is February 31? So what if the insertion fails because the underlying buffer can't access the external device for some obscure reason? So what if a facet throws an exception? We should certainly set some state bits in the respective stream's state and throw or rethrow exceptions, if the exception mask says so.

However, the more general question here is: What are inserters and extractors supposed to do? Some recommendations follow.

Regarding* format flags*, inserters and extractors should:

- Create a sentry object right at the beginning of every inserter and extractor. In its constructor and destructor, the sentry performs certain standard tasks, like skipping white characters, flushing tied streams, etc. See the *Class Reference* for a detailed explanation.
- Reset the width after each usage.

Regarding* state bits*, inserters and extractors should:

- Set `badbit` for all problems with the stream buffer.
- Set `failbit` if the formatting or parsing itself fails.
- Set `eofbit` when the end of the input sequence is reached.

Regarding the* exception mask*, inserters and extractors should:

- Use the `setstate()` function for setting the stream's error state. It automatically throws the `ios_base::failure` exception according to the exceptions switch in the stream's exception mask.
- Catch exceptions thrown during the parsing or formatting, set `failbit` or `badbit`, and rethrow the *original* exception.

Regarding* locales*, inserters and extractors should:

- Use the stream's locale, not the stream buffer's locale. The stream buffer's locale is supposed to be used solely for code conversion.

Regarding the* stream buffer*:

- If you use a sentry object in your extractor or inserter, you should not call any functions from the formatting layer. This would cause a deadlock in a multithreading situation, since the sentry object locks the stream through the stream's *mutex* (= mutual exclusive lock). A nested call to one of the stream's member functions would again create a sentry object, which would wait for the same mutually exclusive lock and, voilà, you have deadlock. Use the stream buffer's functions instead. They do not use the stream's mutex, and are more efficient anyway.

NOTE: Do not call the stream's input or output functions after creating a sentry object in your inserter or extractor. Use the stream buffer's functions instead.

Applying the Recommendations to the Example

Let us now go back and apply the recommendations to the extractor and inserter for class `date` in the example we are constructing. Here is an improved version of the extractor:

©Copyright 1999 Rogue Wave Software, Inc.

```
template<class charT, class Traits>
basic_istream<charT, Traits>& operator >>
  (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;                                    //1

 try {                                                         //2

  typename basic_istream<charT, Traits>::sentry ipfx(is);      //3

  if(ipfx)                                                     //4
  {
   use_facet<time_get<charT,Traits> >(is.getloc())
        .get_date(is, istreambuf_iterator<charT,Traits>()
                  ,is, err, &dat.tm_date);                     //5
   if (!dat) err |= ios_base::failbit;                         //6
  }
 } // try
 catch(...)                                                    //7
 {
    bool flag = FALSE;
    try { is.setstate(ios_base::failbit); }                    //8
    catch( ios_base::failure ) { flag= TRUE; }                 //9
    if ( flag ) throw;                                        //10
 }

 if ( err ) is.setstate(err);                                  /11

 return is;
}
```

//1 The variable `err` keeps track of errors as they occur. In this example, it is handed over to the `time_get` facet, which sets the respective state bits.

//2 All operations inside an extractor or inserter should be inside a try-block, so that the respective error states could be set correctly before the exception is actually thrown.

//3 Here we define the sentry object that does all the preliminary work, like skipping leading white spaces.

//4 We check whether the preliminaries were done successfully. Class `sentry` has a conversion to `bool` that allows this kind of check.

//5 This is the call to the time parsing facet of the stream's locale, as in the primitive version of the extractor.

//6 Let's assume our date class allows us to check whether the date is semantically valid; for example, it would detect wrong dates like February 30. Extracting an invalid date should be treated as a failure, so we set the `failbit`.

//7 Note that in this case it is not advisable to set the `failbit` through the stream's `setstate()` function, because `setstate()` also raises exceptions if they are switched on in the stream's exception mask. We don't want to throw an exception at this point, so we add the `failbit` to the state variable `err`.

//8 Here we catch all exceptions that might have been thrown so far. The intent is to set the stream's error state before the exception terminates the extractor, and to rethrow the original exception.

//9 Now we eventually set the stream's error state through its `setstate()` function. This call might throw an `ios_base::failure` exception according to the stream's exception mask.

//10 We catch this exception because we want the original exception thrown rather than the `ios_base::failure` in all cases.

//11 We rethrow the original exception. If there was no exception raised so far, we set the stream's error state through its `setstate()` function.

The inserter is implemented using the same pattern:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const date& dat)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
   char patt[3] = "%x";
   charT fmt[3];
   use_facet<ctype<charT> >(os.getloc())
      .widen(patt,patt+2,fmt);                                 //1
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   if (
   use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >
     (os.getloc())
     .put(os,os,os.fill(),&dat.tm_date,fmt,(fmt+2))          //2
     .failed()                                               //3
   )
      err = ios_base::badbit;                                //4
   os.width(0);                                              //5
  }
 } //try
 catch(...)
 {
    bool flag = FALSE;
    try {
           os.setstate(ios_base::failbit);
         }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }

 if ( err ) os.setstate(err);

 return os;
}
```

The inserter and the extractor have only a few minor differences:

//1 We prefer to use the other `put()` function of the locale's `time_put` facet. It is more flexible and allows us to specify a sequence of format specifiers instead of just one. We declare a character array that contains the sequence of format specifiers and *widen* it to wide characters, if necessary.

//2 Here we provide the format specifiers to the `time_put` facet's `put()` function.

//3 The `put()` function returns an iterator pointing immediately after the last character produced. We check the success of the previous output by calling the iterators `failed()` function.

//4 If the output failed then the stream is presumably broken, and we set `badbit`.

//5 Here we reset the field width, because the facet's `put()` function uses the stream's format settings and adjusts the output according to the respective field width. The rule is that the field width shall be reset after each usage.

An Afterthought

Why is it seemingly so complicated to implement an inserter or extractor? Why doesn't the first simple approach suffice?

First, it is not really as complicated as it seems if you stick to the patterns: we give these patterns in the next section. Second, the simple extractors and inserters in our first approach do suffice in many cases, when the user-defined type consists mostly of data members of built-in types, and runtime efficiency is not a great concern.

However, whenever you care about the runtime efficiency of your input and output operations, it is advisable to access the stream buffer directly. In such cases, you use fast low-level services and hence need not add format control, error handling, and so on, because low-level services do not handle this for you. In our example, we aimed at optimal performance; the extractor and inserter for locale-dependent parsing and formatting of dates are very efficient because the facets directly access the stream buffer. In all these cases, you should follow the patterns we are about to give.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Patterns for Extractors and Inserters of User-Defined Types

Here is the pattern for an extractor:

```
template<class charT, class Traits>
basic_istream<charT, Traits>& operator >>
  (basic_istream<charT, Traits >& is, UserDefinedType& x)
{
 ios_base::iostate err = 0;

 try {

  typename basic_istream<charT, Traits>::sentry ipfx(is);

  if(ipfx)
  {
   // Do whatever has to be done!
   // Typically you access the stream's locale or buffer.
   // Don't call stream member functions here in MT environments!

   // Add state bits to the err variable if necessary, for example,
   // if (...) err |= ios_base::failbit;
  }
 } // try
 catch(...)                                                  //1
 {
    bool flag = FALSE;
    try { is.setstate(ios_base::failbit); }                  //2
    catch( ios_base::failure ) { flag= TRUE; }               //3
    if ( flag ) throw;                                       //4
 }

 if ( err ) is.setstate(err);                                //5

 return is;
}
```

Similarly, the pattern for the inserter looks like this:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const UserDefinedType& x)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
   // Do whatever has to be done!
   // Typically you access the stream's locale or buffer.
   // Don't call stream member functions here in MT environments!

   // Add state bits to the err variable if necessary, for example,
   // if (...) err |= ios_base::failbit;

   // Reset the field width after usage, that is,
   // os.width(0);
  }
 } //try
 catch(...)
 {
    bool flag = FALSE;
    try { os.setstate(ios_base::failbit); }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 if ( err ) os.setstate(err);

 return os;
}
```

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 12: Manipulators

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

A Recap of Manipulators

We showed examples of manipulators in Section 7.3. There we learned that:

- Manipulators are objects that can be inserted into or extracted from a stream.
- Such insertions and extractions have specific desirable side effects.

As a recap, here is a typical example of two manipulators:

```
cout << setw(10) << 10.55 << endl;
```

The inserted objects `setw(10)` and `endl` are the manipulators. As a side effect, the manipulator `setw(10)` sets the stream's field width to `10`. Similarly, the manipulator `endl` inserts the end of line character and flushes the output.

As we mentioned previously, extensibility is a major advantage of iostreams. We've seen in the previous Section 11.3 how you can implement inserters and extractors for user-defined types that behave like the built-in input and output operations. Additionally, you can add user-defined manipulators that fit seamlessly into the iostreams framework. In this section, we show how to do this.

First of all, to be extracted or inserted, a manipulator must be an object of a type that we call `manipT`, for which overloaded versions of the shift operators exist. (Associated with the manipulator type `manipT`, there is usually a function called $f_{manipT}()$ that we explain in detail later.) Here's the pattern for the manipulator extractor:

```
template <class charT, class Traits>
basic istream<charT,Traits>&
operator>> (basic istream<charT,Traits>& istr
           ,const manipT& manip)
{   return f_manipT(istr, ...);   }
```

With this extractor defined, you can extract a manipulator `Manip`, which is an object of type `manipT`, by simply saying:

```
cin >> Manip;
```

This results in a call to the `operator>>()` sketched out above. The manipulator inserter is analogous.

A manipulator's side effect is often created by calling an associated function $f_{manipT}()$ that takes a stream and returns the stream. There are several ways to associate the manipulator type `manipT` to the function $f_{manipT}()$, which we explore in the following sections. The iostream framework does not specify a way to implement manipulators, but there is a base class called `smanip` that you can use for deriving your own manipulators. We explain this technique along with other useful approaches.

It turns out that there is a major difference between manipulators with parameters like `width(10)` and manipulators without parameters like `endl`. Let's start with the simpler case of manipulators without parameters.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Manipulators without Parameters

Manipulators that do not have any parameters, like `endl`, are the simplest form of manipulator. The manipulator type `manipT` is a function pointer type, the manipulator `Manip` is the function pointer, and the associated function $f_{manipT}()$ is the function pointed to.

In iostreams, the following function pointer types serve as manipulators:

```
(1) ios_base&                    (*pf)(ios_base&)
(2) basic_ios<charT,Traits>&     (*pf)(basic_ios<charT,Traits>)
(3) basic_istream<charT,Traits>& (*pf)(basic_istream<charT,Traits>)
(4) basic_ostream<charT,Traits>& (*pf)(basic_ostream<charT,Traits>)
```

The signature of a manipulator function is not limited to the examples above. If you have created your own stream types, you will certainly want to use additional signatures as manipulators.

For the four manipulator types listed above, the stream classes already offer the required overloaded inserters and member functions. For input streams, extractors take the following form:

```
template<class charT, class traits>
basic_istream<charT, traits>&
basic_istream<charT, traits>::operator>>
(basic_istream<charT,traits>& (*pf)(input_stream_type&) )
{  return (*pf)(*this);..}
```

where ***input_stream_type*** is one of the function pointer types (1)-(3).

Similarly, for output streams we have:

```
template<class charT, class traits>
basic_ostream<charT, traits>&
basic_ostream<charT, traits>::operator<<
(basic_ostream<charT, traits>& (*pf)(output_stream_type& ))
{  return (*pf)(*this);  }
```

where ***output_stream_type*** is one of the function pointer types (1), (2), or (4).

Examples of Manipulators without Parameters

Let's look at the manipulator `endl` as an example of a manipulator without parameters. The manipulator `endl`, which can be applied solely to output streams, is a pointer to the following function of type (4):

```
template<class charT, class traits>
inline basic_ostream<charT, traits>&
endl(basic_ostream<charT, traits>& os)
{
  os.put( os.widen('\n') );
  os.flush();

  return os;
}
```

Hence an expression like:

```
cout << endl;
```

results in a call to the inserter:

```
ostream& ostream::operator<< (ostream& (*pf)(ostream&))
```

with `endl` as the actual argument for `pf`. In other words, `cout << endl;` is equal to `cout.operator<<(endl);`

©Copyright 1999 Rogue Wave Software, Inc.

Here is another manipulator, `boolalpha`, that can be applied to input *and* output streams. The manipulator `boolalpha` is a pointer to a function of type `(1)`:

```
ios_base& boolalpha(ios_base& strm)
{
  strm.setf(ios_base::boolalpha);

  return strm;
}
```

NOTE: Every function that takes a reference to an ios_base, a basic_ios, a basic_ostream, or a basic_istream, and returns a reference to the same stream, can be used as a parameter-less manipulator.

Remark on the Manipulator endl

The manipulator `endl` is often used for inserting the end-of-line character into a stream. However, `endl` does additionally flush the output stream, as you can see from the implementation of `endl` shown above. Flushing a stream, a time-consuming operation that decreases performance, is unnecessary in most common situations. In the standard example:

```
cout << "Hello world" << endl;
```

flushing is not necessary because the standard output stream `cout` is tied to the standard input stream `cin`, so input and output to the standard streams are synchronized anyway. Since no flush is required, the intent is probably to insert the end-of-line character. If you consider typing `'\n'` more trouble than typing `endl`, you can easily add a simple manipulator `nl` that inserts the end-of-line character, but refrains from flushing the stream.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Manipulators with Parameters

Manipulators with parameters are more complex than those without because there are additional issues to consider. Before we explore these issues in detail and examine various techniques for implementing manipulators with parameters, let's take a look at one particular technique, the one that is used to implement standard manipulators such as `setprecision()`, `setw()`, etc.

The Standard Manipulators

Rogue Wave's implementation of the standard iostreams uses a certain technique for implementing most standard manipulators with parameters: the manipulator type `manipT` is a function pointer type; the manipulator object is the function pointed to; and the associated function f_{manipT} is a global function.

The C++ standard defines the manipulator type as `smanip`. The type itself is implementation-defined; all you know is that it is returned by some of the standard manipulators. In Rogue Wave's implementation, `smanip` is a class template:

```
template<class T>
class smanip {
  public:
    smanip(ios_base& (*pf)(ios_base&, T), T manarg);
};
```

A standard manipulator like `setprecision()` can be implemented as a global function returning an object of type `smanip<T>`:

```
inline smanip<int> setprecision(int n)
{ return smanip<int>(sprec, n); }
```

The associated function f_{manipT} is the global function `sprec`:

```
inline ios_base& sprec(ios_base& str, int n)
{
  str.precision(n);
  return str;
}
```

The Principle of Manipulators with Parameters

The previous section gave an example of a technique for implementing a manipulator with one parameter: the technique used to implement the standard manipulators in iostreams. However, this is not the only way to implement a manipulator with parameters. In this section, we examine other techniques. Although all explanations are in terms of manipulators with one parameter, it is easy to extend the techniques to manipulators with several parameters.

Let's start right at the beginning: what is a manipulator with a parameter?

A manipulator with a parameter is an object that can be inserted into or extracted from a stream in an expression like:

```
cout << Manip(x);
cin  >> Manip(x);
```

`Manip(x)` must be an object of type `manipT`, for which the shift operators are overloaded. Here's an example of the corresponding inserter:

```
template <class charT, class Traits>
basic ostream<charT,Traits>&
operator<< (basic ostream<charT,Traits>& ostr
           ,const manipT& manip)
{  // call the associated function f_manipT, for example
   (*manip.f_manipT )(ostr,manip.arg_f);
   return os;
}
```

©Copyright 1999 Rogue Wave Software, Inc.

With this inserter defined, the expression `cout << Manip(x);` is equal to a call to the shift operator sketched above; that is, `operator<<(cout, Manip(x));`

Assuming that a side effect is created by an associated function f_{manipT}, the manipulator must call the associated function with its respective argument(s). Hence it must store the associated function together with its argument(s) for a call from inside the shift operator.

The associated function f_{manipT} can be a static or a global function, or a member function of type `manipT`, for example.

In the inserter above, we've assumed that the associated function f_{manipT} is a static or a global function, and that it takes exactly one argument. Generally, the manipulator type `manipT` might look like this:

```
template <class FctPtr, class Arg1, class Arg2, ...>
class manipT
{
 public:
   manipT(FctPtr, Arg1, Arg2, ...);
 private:
   FctPtr fp_;
   Arg1   arg1_;
   Arg2   arg2_;
};
```

Note that this is only a suggested manipulator, however. In principle, you can define the manipulator type in any way that makes the associated side effect function and its arguments available for a call from inside the respective shift operators for the manipulator type. We show other examples of such manipulator types later in this chapter; for instance, a manipulator type called `smanip` defined by the C++ standard. It is an implementation-defined function type returned by the standard manipulators. See the *Class Reference* for details.

Returning now to the example above, the manipulator object provided as an argument to the overloaded shift operator is obtained by `Manip(x)`, which has three possible solutions:

1. `Manip(x)` is a function call. In this case, `Manip` would be the name of a function that takes an argument of type `x` and returns a manipulator object of type `manipT`; that is, `Manip` is a function with the following signature:

   ```
   manipT Manip (X x);
   ```

2. `Manip(x)` is a constructor call. In this case, `Manip` would be the name of a class with a constructor that takes an argument of type `X` and constructs a manipulator object of type `Manip`; that is, `Manip` and `manipT` would be identical:

   ```
   class Manip {
    public:
      Manip(X x);
   };
   ```

3. `Manip(x)` is a call to a function object. In this case, `Manip` would be an object of a class `M`, which defines a function call operator that takes an argument of type `x` and returns a manipulator object of type `manipT`:

   ```
   class M {
    public:
      manipT operator()(X x);
   } Manip;
   ```

Solutions **1.)** and **2.)** are semantically different from solution **3.)**. In solution **1.)**, `Manip` is a function and therefore need not be created by the user. In solution **2.)**, `Manip` is a class name and an unnamed temporary object serves as manipulator object. In solution **3.)**, however, the manipulator object `Manip` must be explicitly created by the user. Hence the user has to write:

```
manipT Manip;

cout << Manip(x);
```

which is somewhat inconvenient because it forces the user to know an additional name, the manipulator type `manipT`, and to create the manipulator object `Manip`.13 For these reasons, solution **3.)** is useful if the manipulator has *state*; that is, if it stores additional data like a manipulator, let's call it `lineno`, which provides the next line number each time it is inserted.

The problem with solution **2.)** is that several current compilers cannot handle unnamed objects.

©Copyright 1999 Rogue Wave Software, Inc.

For any of the three solutions just discussed, there is also a choice of associated functions. The associated function f_{manipT} can be either:

a. A static or a global function;

b. A static member function;

c. A virtual member function.

Among these choices, **b.)**, the use of a static member function, is the preferable in an object-oriented program because it permits encapsulation of the manipulator together with its associated function. This is particularly recommended if the manipulator has *state*, as in solution **3.)**, where the manipulator is a function object, and the associated function has to access the manipulator's state. Using **c.)**, a virtual member function, introduces the overhead of a virtual function call each time the manipulator is inserted or extracted. It is useful if the manipulator has state, and the state needs to be modified by the associated manipulator function. A static member function would only be able to access the manipulator's static data; a non-static member function, however, can access the object-specific data.

Examples of Manipulators with Parameters

In this section, let's look at some examples of manipulators with parameters. The examples here are arbitrary combinations of solutions **1.)** to **3.)** for the manipulator type, with **a.)** to for the associated function. We also use the standard manipulator `setprecision()` to demonstrate the various techniques.

Example 1: Function Pointer and Global Function. This example combines **1.)** and **c.)**, and so:

- `manipT` is a function pointer type.
- The manipulator object is the function pointed to.
- The associated function f_{manipT} is a global function.

Rogue Wave's implementation of the standard iostreams uses this technique for implementing most standard manipulators with parameters. See Section 7.3.2 for reference.

Example 2: Unnamed Object and Static Member Function. This example combines **2.)** and **b.)**, and thus:

- The manipulator object `Manip` is an unnamed object.
- The manipulator type `manipT` is a class.
- The associated function f_{manipT} is a static member function.

The manipulator type `manipT` can be derived from the manipulator type `smanip` defined by iostreams. Here is an alternative implementation of a manipulator like `setprecision()`:

```
class setprecision : public smanip<int> {
  public:
    setprecision(int n) : smanip<int>(sprec_, n) { }
  private:
    static ios_base& sprec_(ios_base& str, int n)
    { str.precision(n);
      return str;
    }
};
```

Example 3: Unnamed Object and Virtual Member Function. This example **2.)** and **c.)**, and therefore:

- The manipulator object `Manip` is an unnamed object.
- The manipulator type `manipT` is a class.
- The associated function f_{manipT} is a virtual member function of that class.

©Copyright 1999 Rogue Wave Software, Inc.

The idea here is that the associated function f_{manipT} is a non-static member function of the manipulator type `manipT`. In such a model, the manipulator does not store a pointer to the associated function f_{manipT}, but defines the associated function as a pure virtual member function. Consequently, the manipulator type `manipT` is an abstract class, and concrete manipulator types are derived from this abstract manipulator type. They are required to implement the virtual member function that represents the associated function.

Clearly, we need a new manipulator type because the standard manipulator type `smanip` is implementation-defined. In Rogue Wave's **Standard C++ Library**, it has no virtual member functions, but stores a pointer to the associated function. Here is the abstract manipulator type we need:

```
template <class Arg, class Ostream>
class virtsmanip
{
 public:
  typedef Arg argument_type;
  typedef Ostream ostream_type;
  virtsmanip (Arg a) : arg_(a) { }

 protected:
  virtual Ostream& fct_(Ostream&,Arg) const = 0;
  Arg arg_;

  friend Ostream&
  operator<< (Ostream& ostr
              ,const virtsmanip<Arg,Ostream>& manip);
};
```

This type `virtsmanip` differs from the standard type `smanip` in several ways:

- It defines the above-mentioned pure virtual member function `fct_()`.
- The argument `arg_` and the virtual function `fct_()` are protected members, and consequently the respective shift operator for the manipulator type has to be a friend function.
- It is a base class for output manipulators only.

The standard manipulator `smanip` expects a pointer to a function that takes an `ios_base` reference. In this way, a manipulator is always applicable to input *and* output streams, regardless of whether or not this is intended. With our new manipulator type `virtsmanip`, we can define manipulators that cannot inadvertently be applied to input streams.

Since we have a new manipulator type, we also need a new overloaded version of the manipulator inserter:

```
template <class Arg, class Ostream>
Ostream&
operator<< (Ostream& ostr, const virtsmanip<Arg,Ostream>& manip)
{
   manip.fct_(ostr,manip.arg_);
   return ostr;
}
```

After these preparations, we can now provide yet another alternative implementation of a manipulator like `setprecision()`. This time `setprecision()` is a manipulator for output streams only:

```
class setprecision : public virtsmanip<int,basic_ostream<char> >
{
public:
 setprecision(argument_type n)
 : virtsmanip<argument_type,ostream_type>(n) { }

protected:
 ostream_type& fct_(ostream_type& str, argument_type n) const
 {
   str.precision(n);
   return str;
 }
};
```

Example 4: Function Object and Static Member Function. The next example combines **3.)** and **b.)**, so here:

- The manipulator object `Manip` is an object of a type `M` that defines the function call operator.

©Copyright 1999 Rogue Wave Software, Inc.

- The manipulator type `manipT` is a class type that is returned by the overloaded function call operator of class `M`.
- The associated function f_{manipT} is a static member function of class `M`.

This solution, using a function object as a manipulator, is semantically different from the previous solution in that the manipulator object has *state*, that is, it can store data between subsequent uses.

Let us demonstrate this technique in terms of another example: an output manipulator that inserts a certain string that is maintained by the manipulator object. Such a manipulator could be used, for instance, to insert a prefix to each line:

```
Tag<char> change_mark("v1.2 >> ");

while ( new_text )
   ostr << change_mark << next_line;

change_mark("");
while ( old_text)
   ostr << change_mark << next_line;
```

We would like to derive the `Tag` manipulator here from the standard manipulator `smanip`. Unfortunately, `smanip` is restricted to associated functions that take an `ios_base` reference as a parameter. In our example, we want to insert the stored text to the stream, so we need the stream's inserter. However, `ios_base` does not have inserters or extractors. Consequently we need a new manipulator base type, similar to `smanip`, that allows associated functions that take a reference to an output stream:

```
template <class Ostream, class Arg>
class osmanip {
  public:
    typedef Ostream ostream_type;
    typedef Arg argument_type;

    osmanip(Ostream& (*pf)(Ostream&, Arg), Arg arg)
    : pf_(pf) , arg_(arg) { ; }

  protected:
    Ostream&     (*pf_)(Ostream&, Arg);
    Arg          arg_;

  friend Ostream&
    operator<<
    (Ostream& ostr, const osmanip<Ostream,Arg>& manip);
};
```

Then we need to define the inserter for the new manipulator type `osmanip`:

```
template <class Ostream, class Arg>
Ostream&
operator<< (Ostream& ostr,const osmanip<Ostream,Arg>& manip)
{
   (*manip.pf_)(ostr,manip.arg_);
   return ostr;
}
```

Now we define the function object type `M`, here called `Tag`:

```
template <class  charT>
class Tag
: public osmanip<basic_ostream<charT>, basic_string<charT> >
{
public:
 Tag(argument_type a = "")
  : osmanip<basic_ostream<charT>, basic_string<charT> >
    (fct_, a) { }

 osmanip<ostream_type,argument_type>&
  operator() (argument_type a)
  {
    arg_ = a;
    return *this;
  }

private:
 static ostream_type& fct_ (ostream_type& str, argument_type a)
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 {
   return str << a;
 }
};
```

Note that the semantics of this type of manipulator differ from the previous ones, and from the standard manipulator `setprecision`. The manipulator object has to be explicitly created before it can be used, as shown in the example below:

```
Tag<char> change_mark("v1.2 >> ");

while ( new_text )
   ostr << change_mark << next_line;

change_mark("");
while ( old_text)
   ostr << change_mark << next_line;
```

This kind of manipulator is more flexible. In the example above, you can see that the default text is set to `"v1.2 >>"` when the manipulator is created. Thereafter you can use the manipulator as a parameterless manipulator and it will remember this text. You can also use it as a manipulator taking an argument, and provide it with a different argument each time you insert it.

Example 5: Function Object and Virtual Member Function. In the previous example, a static member function is used as the associated function. This has the slight disadvantage that the associated function cannot modify the manipulator's state. Should modification be necessary, you might consider using a virtual member function instead.

Our final example here is a manipulator that stores additional data, the previously mentioned `lineno` manipulator. It adds the next line number each time it is inserted:

```
LineNo lineno;
while (!cout)
{
   cout << lineno << ...;
}
```

The manipulator is implemented following the **3.)** and **b.)** pattern, that is:

- The manipulator object `Manip` is an object of a type `M` that defines the function call operator.
- The manipulator type `manipT` is a class type that is returned by the overloaded function call operator of class `M`.
- The associated function f_{manipT} is a virtual member function of class `M`.

The manipulator object contains a line number that is initialized when the manipulator object is constructed. Each time the `lineno` manipulator is inserted, the line number is incremented.

For the manipulator base type, we use a slightly modified version of the manipulator type `osmanip` from Example 3. The changes are necessary because the associated function in this case may not be a constant member function:

```
template <class Arg, class Ostream>
class virtsmanip
{
 public:
  typedef Arg argument_type;
  typedef Ostream ostream_type;
  virtsmanip (Arg a) : arg_(a) { }

 protected:
  virtual Ostream& fct_(Ostream&,Arg) = 0;
  Arg arg_;

  friend Ostream&
  operator<< (Ostream& ostr
              ,virtsmanip<Arg,Ostream>& manip);
};

template <class Arg,class Ostream>
Ostream&
operator<< (Ostream& ostr
            ,virtsmanip<Arg,Ostream>& manip)
{
   manip.fct_(ostr,manip.arg_);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
   return ostr;
}
```

The line number manipulator could be implemented like this:

```
template <class Ostream>
class LineNo
 : public virtsmanip<int,Ostream >
{
public:
 LineNo(argument_type n=0)
  : virtsmanip<argument_type, ostream_type> (n)
  { }

 virtsmanip<argument_type,ostream_type>&
  operator() (argument_type arg)
  {
   arg_ = arg;
   return *this;
  }
protected:
 argument_type lno_;
 ostream_type& fct_ (ostream_type& str, argument_type n)
 {
   lno_ = (n>0) ? n : lno_;
   str << ++lno_;
   arg_ = -1;
   return str;
 }
};
```

Using a virtual member function as the associated manipulator function introduces the overhead of a virtual function call each time the manipulator is inserted. If it is necessary that a manipulator update its state after each insertion, a static member function does not suffice. A global function that is a friend of the manipulator type might do the trick. However, in an object-oriented program, you are usually advised against global functions that modify private or protected data members of a class with whom they are friends.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 13: Streams and Stream Buffers

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Streams as Objects

So far we have used streams as the source or target of input and output operations. But there is another aspect of streams: they are also objects in your C++ program that you might want to copy and pass around like other objects. However, streams cannot simply be copied and assigned. The following chapter shows what you must do instead.

Stream buffers play a special role. In numerous cases you will not want to create copies of a stream buffer object, but simply share a stream buffer among streams. Section 13.3 explores this option.

If you really want to work with copies of stream buffers, see Section 13.2 for hints and caveats.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Copying and Assigning Stream Objects

Stream objects cannot simply be copied and assigned. Let us consider a practical example to see what this means. A program writes data to a file if a file name is specified on program call, or to the standard output stream `cout` if no file name is specified. You should write to one output stream in your program; this stream can be either a file stream or the standard output stream. The most obvious way to do this is to declare an output file stream object and assign it to `cout`, or to use `cout` directly. However, you can't do it this way:

```
int main(int argc, char argv[])
{
  ofstream fil;
  if (argc > 1)
  {  fil.open(argv[1]);
     cout = fil;                              // can't do this !!!
  }
  // output to cout, for example
  cout << "Hello world!" << endl;
}
```

This solution is bad for at least three reasons. First, the predefined streams `cin`, `cout`, `cerr`, and `clog` have special properties and are treated differently from other streams. If you could reassign them, as done with `cout` in the example above, you would lose their special properties. Second, assignment and copying of streams is hazardous. Even if the assignment of the output stream `fil` compiles, your program is likely to crash afterwards.14 Finally, the base class for iostreams has private assignment and coy constructors to prevent you from doing this.

NOTE: Stream objects must never be copied or assigned to each other.

Copying a Stream's Data Members

To achieve the equivalent effect of a copy, you might consider copying each data member individually. This can be done as follows:

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)
     out.open(argv[1]);
  else
  {  out.copyfmt(cout);                                  //1
     out.clear(cout.rdstate());                          //2
     out.rdbuf(cout.rdbuf());                            //3
  }
  // output to out, for example
  out << "Hello world!" << endl;
}
```

//1 The `copyfmt()` function copies all data from the standard output stream `cout` to the output file stream `out`, except the error state and the stream buffer. There is a function `exceptions()` that allows you to copy the exception mask separately, as in `cout.exceptions(fil.exceptions())`, but you need not do this explicitly, since `copyfmt()` already copies the exception mask.

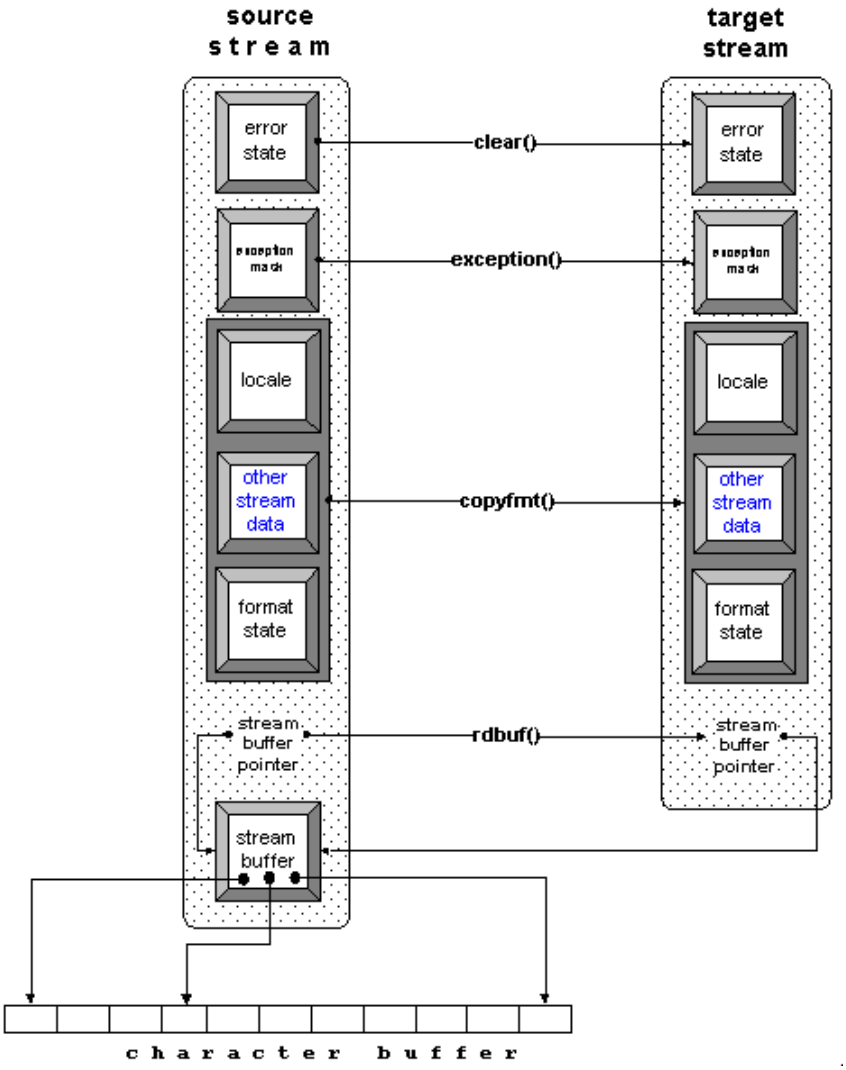
//2 Here the error state is copied.

//3 Here the stream buffer pointer is copied.

Please note the little snag here. After the call to `rdbuf()`, the buffer is shared between the two streams, as shown in Figure 30:

Figure 30 -- Copying a stream's internal data results in a shared buffer

©Copyright 1999 Rogue Wave Software, Inc.



Sharing Stream Buffers Inadvertently

Whether or not you intend to share a stream buffer among streams depends on your application. In any case, it is important that you realize the stream buffer is shared after a call to `rdbuf()`; in other words, you must monitor the lifetime of the stream buffer object and make sure it exceeds the lifetime of the stream. In our little example above, we use the standard output stream's buffer. Since the standard streams are static objects, their stream buffers have longer lifetimes that most other objects, so we are safe. However, whenever you share a stream buffer among other stream objects, you must carefully consider the stream buffer's lifetime.

The example above has another disadvantage we haven't considered yet, as shown in the following code:

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)
     out.open(argv[1]);
  else
  {  out.copyfmt(cout);                                  //1
     out.clear(cout.rdstate());                          //2
     out.rdbuf(cout.rdbuf());                            //3
  }
  // output to out, for example
  out << "Hello world!" << endl;
}
```

//1 Copy the values of member variables (other than the streambuffer and the iostate) in `cout` to `out`.

//2 Set state flags for `out` to the current state of `cout`.

//3 Replace `out`'s streambuffer with `cout`'s streambuffer.

As we copy the standard output stream's entire internal data, we also copy its special behavior. For instance, the standard output stream is synchronized with the standard input stream. (See Chapter 14 for further details.) If our output file stream `out` is a copy of `cout`, it is forced to synchronize its output operations with all input operations from `cin`. This

©Copyright 1999 Rogue Wave Software, Inc.

might not be desired, especially since synchronization is a time-consuming activity. Here is a more efficient approach using only the stream buffer of the standard output stream:

```
int main(int argc, char argv[])
{
  filebuf* fb = new filebuf;                                    //1
  ostream out((argc>1)?                                         //2
    fb->open(argv[1],ios_base::out|ios_base::trunc):            //3
    cout.rdbuf());                                              //4
  if (out.rdbuf() != fb)
    delete fb;
  out << "Hello world!" << endl;
}
```

//1 Instead of creating a file stream object, which already contains a file buffer object, we construct a separate file buffer object on the heap that we can hand over to an output stream object if needed. This way we can delete the file buffer object if not needed. In the original example, we constructed a file stream object with no chance of eliminating the file buffer object if not used.

//2 An output stream is constructed. The stream has either the standard output stream's buffer, or a file buffer connected to a file.

//3 If the program is provided with a file name, the file is opened and connected to the file buffer object. (Note that you must ensure that the lifetime of this stream buffer object exceeds the lifetime of the output stream that uses it.) The `open()` function returns a pointer to the file buffer object. This pointer is used to construct the output stream object.

//4 If no file name is provided, the standard output stream's buffer is used.

As in the original example, `out` inserts through the standard output stream's buffer, but lacks the special properties of a standard stream.

Here is an alternative solution that uses file descriptors, a nonstandard feature of Rogue Wave's implementation of the standard iostreams15:

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)     out.open(argv[1]);                          //1
  else              out.rdbuf()->open(1);                       //2
  out << "Hello world!" << endl;
}
```

//1 If the program is provided with a file name, the file is opened and connected to the file buffer object.

//2 Otherwise, the output stream's file buffer is connected to the standard input stream `stdout` whose file descriptor is `1`.

The effect is the same as in the previous solution, because the standard output stream `cout` is connected to the C standard input file `stdout`. This is the simplest of all solutions, because it doesn't involve reassigning or sharing stream buffers. The output file stream's buffer is simply connected to the right file. However, this is a nonstandard solution, and may decrease portability.

Using Pointers or References to Streams

If you do not want to deal with stream buffers at all, you can also use pointers or references to streams instead. Here is an example:

```
int main(int argc, char argv[])
{
  ostream* fp;                                                  //1
  if (argc > 1)
      fp = new ofstream(argv[1]);                               //2
  else
     fp = &cout                                                 //3

  // output to *fp, for example
  *fp << "Hello world!" << endl;                                //4
  if(fp!=&cout) delete fp;
}
```

//1 A pointer to an `ostream` is used. (Note that it cannot be a pointer to an `ofstream`, because the standard output stream `cout` is not a file stream, but a plain stream of type `ostream`.)

//2 A file stream for the named output file is created on the heap and assigned to the pointer, in case a file name is

©Copyright 1999 Rogue Wave Software, Inc.

provided.

//3 Otherwise, a pointer to `cout` is used.

//4 Output is written through the pointer to either `cout` or the named output file.

An alternative approach could use a reference instead of a pointer:

```
int main(int argc, char argv[])
{
  ostream& fr = (argc > 1) ? *(new ofstream(argv[1])) : cout;
  // output to *fr, for example
  fr << "Hello world!" << endl;
  if (&fr!=&cout) delete(&fr);
}
```

Working with pointers and references has a drawback: you must create an output file stream object on the heap and, in principle, you must worry about deleting the object again, which might lead you into other dire straits.

In summary, creating a copy of a stream is not trivial and should only be done if you really need a copy of a stream object. In many cases, it is more appropriate to use references or pointers to stream objects instead, or to share a stream buffer between two streams.

NOTE: Never create a copy of a stream object when a reference or a pointer to the stream object would suffice, or when a shared stream buffer would solve the problem.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Sharing a Stream Buffer Among Streams

Despite the previous caveats, there are situations where sharing a stream buffer among streams is useful and intended. Let us focus on these in this section.

Several Format Settings for the Same Stream

Imagine you need different formatting for different kinds of output to the same stream. Instead of switching the format settings between the different kinds of output, you can arrange for two separate streams to share a stream buffer. The streams would have different format settings, but write output to the same stream buffer. Here is an example:

```
ofstream file1("/tmp/x");
ostream file2(file1.rdbuf());                                   //1

file1.setf(ios_base::fixed, ios_base::floatfield);              //2
file1.precision(5);
file2.setf(ios_base::scientific, ios_base::floatfield);
file2.precision(3);

file1 << setw(10) << 47.11 << '\n';                             //3
file2 << setw(10) << 47.11 << '\n';                             //4
```

//1 The stream buffer of `file1` is replaced by the stream buffer of `file2`. Afterwards, both streams share the buffer.

//2 Create different format settings for both files.

//3 The output here is: 47.11000

//4 The output here is: 4.711e+01

Note that `file2` in the example above has to be an output stream rather than an output file stream. This is because file streams do not allow you to switch the file stream buffer.

Several Locales for the Same Stream

Similarly, you can use separate streams that share a stream buffer in order to avoid locale switches. This is useful when you must insert multilingual text into the same stream. Here is an example:

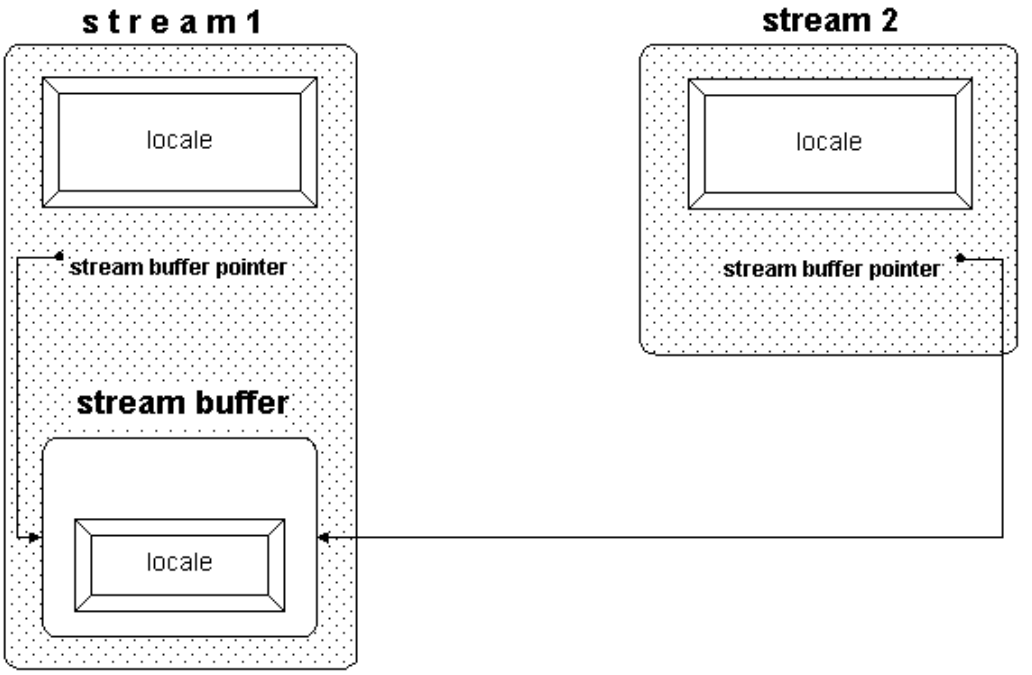
```
  ostringstream file1;
  ostream file2(file1.rdbuf());

  file1.imbue(locale("De_DE"));
  file2.imbue(locale("En_US"));

  file1 << 47.11 << '\t';
  file2 << 47.11 << '\n';

  cout << file1.str() << endl;                                  //1
```

//1 The output is: `47,11 47.11`

Again, there is a little snag. In Figure 31, note that a stream buffer has a locale object of its own, in addition to the stream's locale object.

Figure 31 -- Locale objects and shared stream buffers

©Copyright 1999 Rogue Wave Software, Inc.



Section 6.4.4 explained the role of those two locale objects. To recap, the stream delegates the handling of numeric entities to its locale's numeric facets. The stream buffer uses its locale's code conversion facet for character-wise transformation between the buffer content and characters transported to and from the external device.

Usually the stream's locale and the stream buffer's locale are identical. However, when you share a stream buffer between two streams with different locales, you must decide which locale the stream buffer will use.

You can set the stream buffer's locale by calling the `pubimbue()` function as follows:

```
file1.imbue(locale("De_DE"));
file2.imbue(locale("En_US"));
file1.rdbuf()->pubimbue(locale("De_DE"));
```

Input and Output to the Same Stream

You can also use a shared stream buffer in order to have read *and* write access to a stream:

```
filebuf fbuf;                                                    //1
  fbuf.open("/tmp/inout",ios_base::in|ios_base::out);            //2
  istream in(&fbuf);                                             //3
  ostream out(&fbuf);                                            //4

  cout << in.rdbuf();                                            //5
  out << "..." << '\n' ;                                         //6
```

//1 Create a file buffer.

//2 Connect the file buffer to a file. Note that you must open the file in input and output mode if you want to read *and* write to it.

//3 Create an input stream that works with the file buffer `fbuf`.

//4 Create an output stream that also uses the file buffer `fbuf`.

Read the entire content of the file and insert it into the standard output stream. Afterwards the file position is at the end of the file.

//5 The most efficient way to read a file's entire content is through the `rdbuf()` function, which returns a pointer to the underlying stream buffer object. There is an inserter available that takes a stream buffer pointer, so you can insert the buffer's content into another stream.

//6 Write output to the file. As the current file position is the end of the file, all output is inserted at the end.

Naturally, it is easier and less error-prone to use bidirectional streams when you must read and write to a file. The bidirectional equivalent to the example above would be:

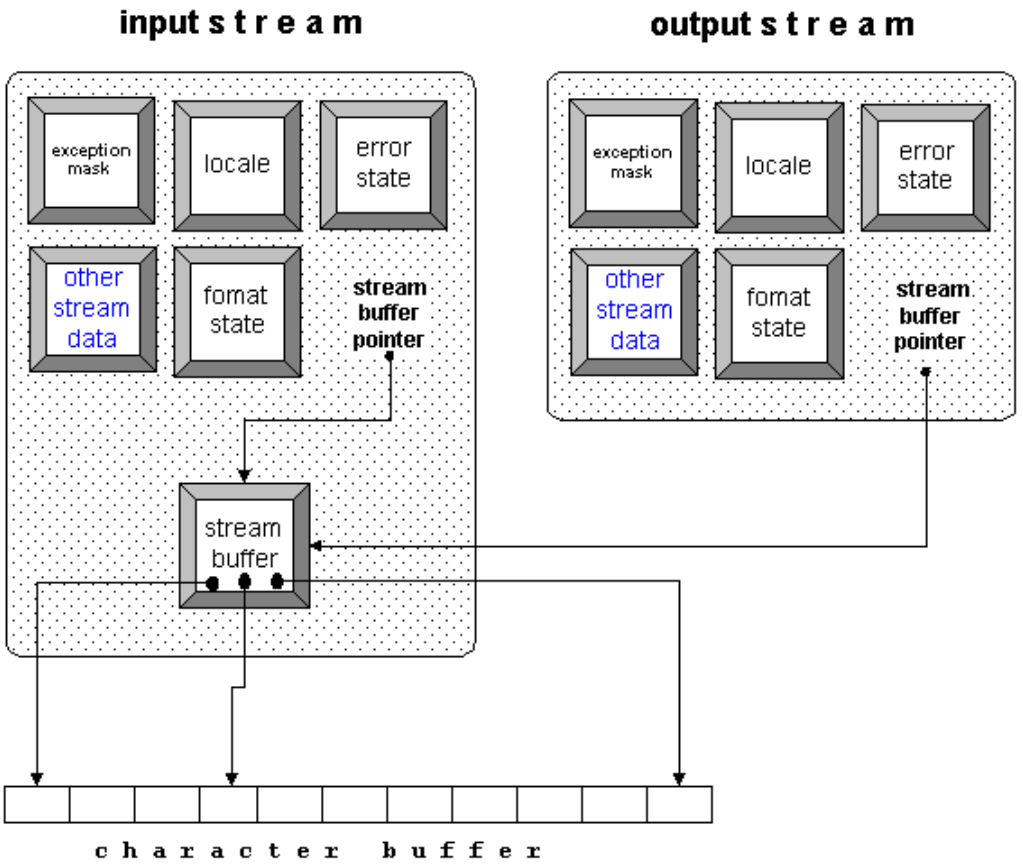
```
fstream of("/tmp/inout");
  cout << of.rdbuf();
```

©Copyright 1999 Rogue Wave Software, Inc.

```
of << "..." << '\n' ;
```

Notice that there is a difference between the solutions that you can see by comparing Figure 32 and Figure 33. An input and an output stream that share a stream buffer, as shown in Figure 32, can still have separate format settings, different locales, different exception masks, and so on.

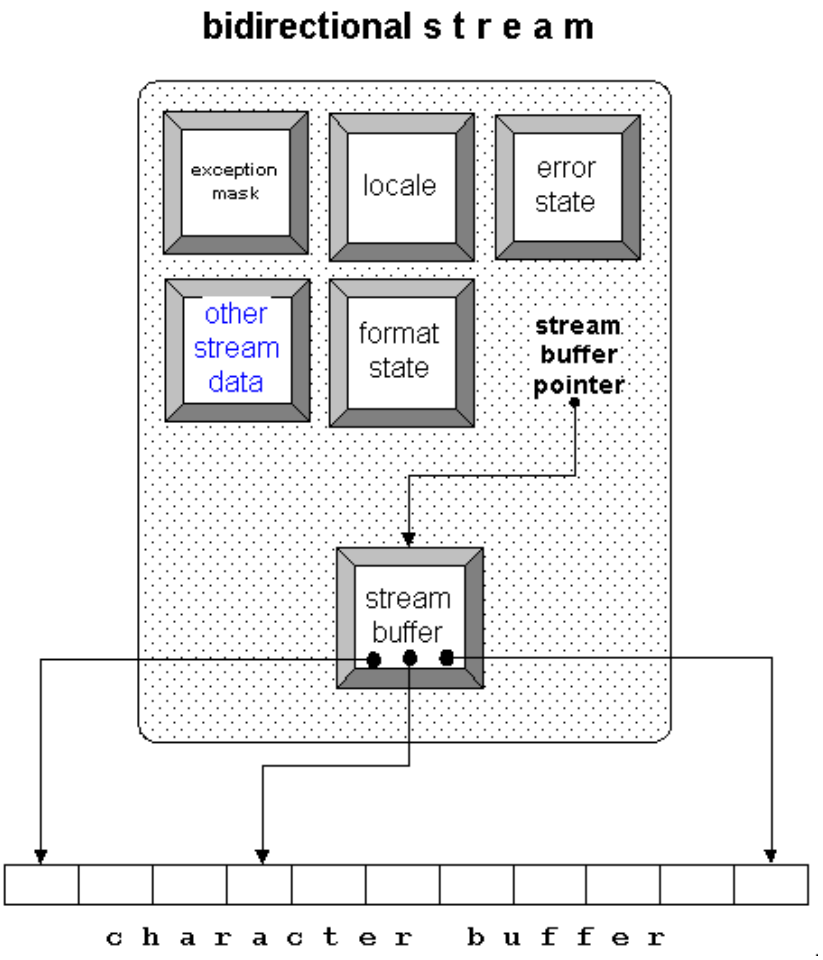
Figure 32 -- An input and an output stream sharing a stream buffer



In contrast, the bidirectional stream shown in Figure 33 can have only one format setting, one locale, and so on:

Figure 33 -- A bidirectional stream

©Copyright 1999 Rogue Wave Software, Inc.



It seems clear that you cannot have different settings for input and output operations when you use a bidirectional stream. Still, it is advisable to use bidirectional file or string streams if you need to read and write to a file or string, instead of creating an input and an output stream that share a stream buffer. The bidirectional stream is easier to declare, and you need not worry about the stream buffer object's lifetime.

NOTE: It's better to use one bidirectional file or string stream for reading and writing to a file or string, rather than two streams that share a stream buffer.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Copies of the Stream Buffer

The previous section showed how you can read the content of a file in its entirety by using the `rdbuf()` function. Let us now explore a variation of that example. Imagine another file containing some sort of header information that needs to be analyzed before we start appending the file. Instead of writing the current file content to the standard output stream, we want to process the content before we start adding to it. The easiest way to put the entire file content into a memory location for further processing is by creating a string stream and inserting the file's stream buffer into the string stream:

```
fstream fil("/tmp/inout");
  stringstream header_stream;                          //1
  header_stream << fil.rdbuf();                        //2

  // process the header, for example
  string word;
  header_stream >> word;                               //3
```

//1 The easiest way to put the entire file content into a memory location for further processing is by creating a string stream, and

//2 Inserting the file's stream buffer into the string stream.

//3 We now have the usual facilities of an input stream for reading and analyzing the header information; that is, `operator>>()`, `read()`, `get()`, and so on.

In cases where this procedure is insufficient, you should create a string that contains the header information and process the header by means of the string operations `find()`, `compare()`, etc.

```
fstream fil("/tmp/inout");
  header_stream << fil.rdbuf();
  string header_string = header_stream.str();

  // process the header, for example
  string::size_type pos = header_string.rfind('.');
```

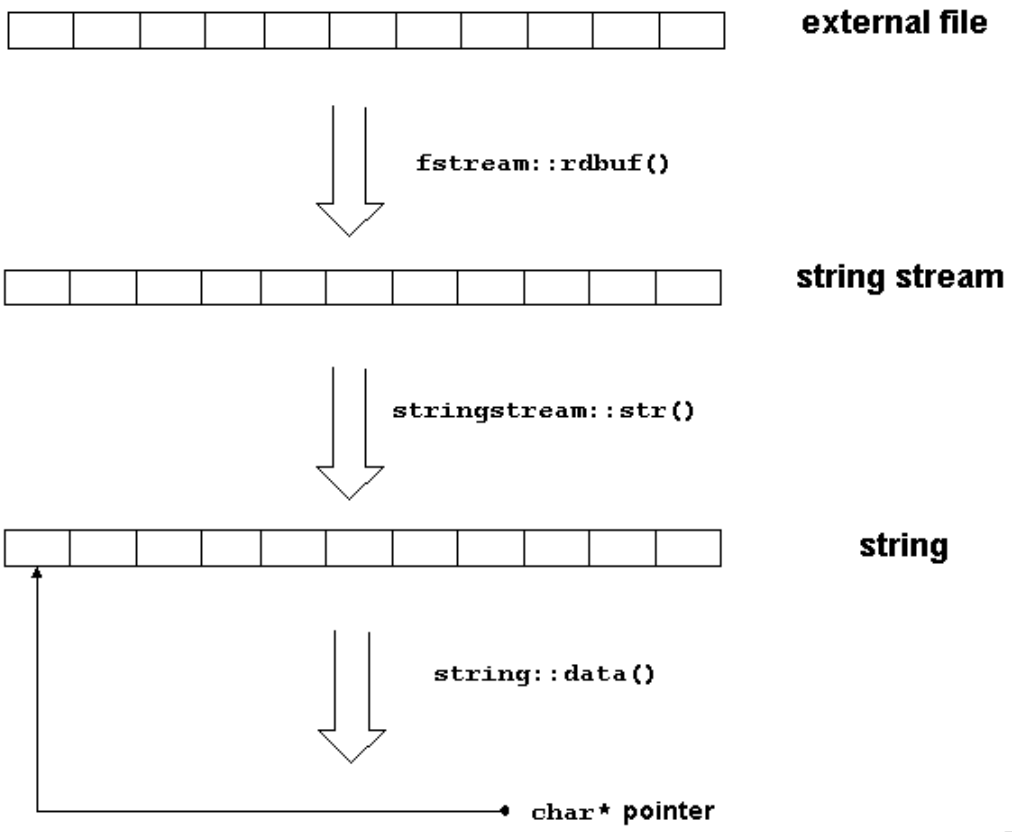
If the header contains binary data instead of text, even a string will probably not suffice. Here you would want to see the header as a plain byte sequence, that is, an ordinary `char*` buffer. But note that a code conversion might already have been performed, depending on the locale attached to the file stream. In cases where you want to process binary data, you must ensure that the attached locale has a non-converting code conversion facet:

```
fstream fil("/tmp/inout");
  header_stream << fil.rdbuf();
  string header_string = header_stream.str();
  const char* header_char_ptr = header_string.data();

  // process the header, for example
  int idx;
  memcpy((char*) &idx,header_char_ptr,sizeof(int));
```

A note on efficiency: if the header information is extensive, you must consider the number of copy operations performed in the previous example. Figure 34 shows how these copies are made:

Figure 34 -- Copies of the file content

©Copyright 1999 Rogue Wave Software, Inc.



The content of the file is copied into the string stream's buffer when the pointer obtained through `rdbuf()` is inserted to the string stream. A second copy is created when the string stream's function `str()` is called.16 The call to the string's function `data()` does not create yet another copy, but returns a pointer to the string's internal data.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 14: Synchronizing Streams

Top Contents Index

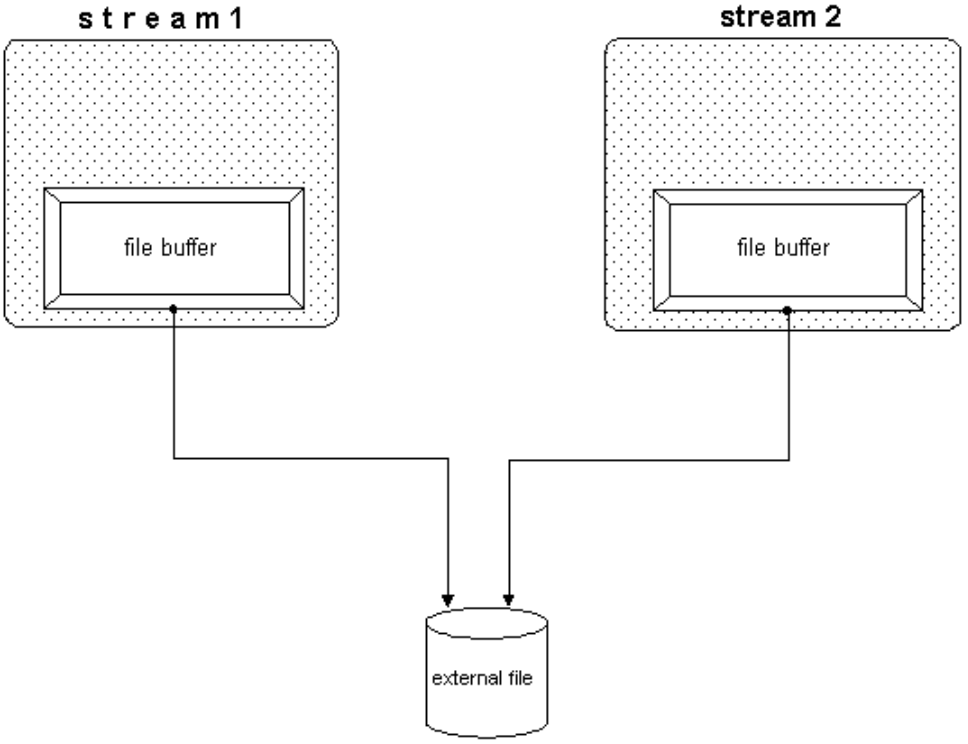
OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Sharing Files Among Streams

In the previous chapter, we saw how streams can share stream buffers. In this chapter, we show that streams can also share files, as when streams in different processes exchange data through a file. Figure 35 illustrates how streams share files:

Figure 35 -- Streams sharing a file



Because streams use a buffer, the content of the file might be different from the content of the buffer that is supposed to reflect the file's content. When data is extracted through a file stream, a certain part of the file's content is read into the buffer; subsequent extractions access the buffer instead of the file. Once the file content is modified, the buffer content becomes obsolete. Similarly, when data is written through a file stream, the output is stored in the buffer and not written to the file. The file is accessed only when the buffer is full. For this reason, output from one stream is not immediately available to the other stream.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Explicit Synchronization

You can force the stream to empty its buffer into an output file, or to refill its buffer from an input file. This is done through the stream buffer's function `pubsync()`. Typically, you call `pubsync()` indirectly through functions of the stream layer. Input streams and output streams have different member functions that implicitly call `pubsync()`.

Output Streams

Output streams have a `flush()` function that writes the buffer content to the file. In case of failure, `badbit` is set or an exception thrown, depending on the exception mask.

```
ofstream ofstr("/tmp/fil");
ofstr << "Hello ";                                    //1
ofstr << "World!\n";
ofstr.flush();                                        //2
```

//1 The attempt to extract anything from the file `/tmp/fil` after this insertion will probably fail, because the string `"Hello"` is buffered and not yet written to the external file.

//2 After the call to `flush()`, however, the file contains `"Hello World!\n"`. (Incidentally, the call to `ostr.flush()` can be replaced by the `flush` manipulator; that is, `ostr << flush;`)

Keep in mind that flushing is a time-consuming operation. The function `flush()` not only writes the buffer content to the file; it may also reread the buffer in order to maintain the current file position. For the sake of performance, you should avoid inadvertent flushing, as when the `endl` manipulator calls `flush()` on inserting the end-of-line character. (See Section 7.3.2.)

Input Streams

Input streams have a `sync()` function. It forces the stream to access the external device and refill its buffer, beginning with the current file position.[17] The example below demonstrates the principle theoretically. In real life, however, the two streams would belong to two separate processes and could use the shared file to communicate.

```
ofstream ofstr("/tmp/fil");
ifstream ifstr("/tmp/fil");
string s;

ofstr << "Hello "
ofstream::pos_type p = ofstr.tellp();
ofstr << "World!\n" << flush;
ifstr >> s;                                           //1

ofstr.seekp(p);
ofstr << "Peter!" << flush;                           //2
ifstr >> s;                                           //3

ofstr << " Happy Birthday!\n" << flush;               //4
ifstr >> s;                                           //5

ifstr.sync();                                         //6
ifstr >> s;
```

//1 Here the input stream extracts the first string from the shared file. In doing so, the input stream fills its buffer. It reads as many characters from the external file as needed to fill the internal buffer. For this reason, the number of characters to be extracted from the file is implementation-specific; it depends on the size of the internal stream buffer.

//2 The output stream overwrites part of the file content. Now the file content and the content of the input stream's buffer are inconsistent. The file contains `"Hello Peter!"`; the input stream's buffer still contains `"Hello World!"`.

//3 This extraction takes the string `"World!"` from the buffer instead of yielding `"Peter!"`, which is the current file content.

//4 More characters are appended to the external file. The file now contains `"Hello Peter! Happy Birthday!"`, whereas the input stream's buffer is still unchanged.

©Copyright 1999 Rogue Wave Software, Inc.

//5 This extraction yields nothing. The input stream filled its buffer with the entire content of the file because the file is so small in our toy example. Subsequent extractions made the input stream hit the end of its buffer, which is regarded as the end of the file as well. The extraction results in `eofbit` set, and nothing is extracted. There is no reason to ever access the external file again.

A call to `sync()` eventually forces the input stream to refill the buffer from the external device, beginning with the current file position. After the synchronization, the input stream's buffer contains `"Happy Birthday!\n"`. The next extraction yields `"Happy"`.

//6

As the standard specifies the behavior of `sync()` as implementation-defined, you can alternatively try repositioning the input stream to the current position instead; for example, `istr.seekg(ios_base::cur);`.

NOTE: If you must synchronize several streams that share a file, it is advisable to call the `sync()` function after each output operation and before each input operation.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Implicit Synchronization Using the unitbuf Format Flag

You can achieve a kind of automatic synchronization for output files by using the format flag `ios_base::unitbuf`. It causes an output stream to flush its buffer after each output operation as follows:

```
ofstream ostr("/tmp/fil");
ifstream istr("/tmp/fil");

ostr << unitbuf;                                          //1

while (some_condition)
{ ostr << "... some output...";                              //2
  // process the output
  istr >> s;
  // ...
}
```

//1 Set the `unitbuf` format flag.

//2 After each insertion into the shared file `/tmp/fil`, the buffer is automatically flushed, and the output is available to other streams that read from the same file.

Since it is not overly efficient to flush after every single token that is inserted, you might consider switching off the `unitbuf` flag for a lengthy output that is not supposed to be read partially.

```
ostr.unsetf(ios_base::unitbuf);                           //1
ostr << " ... some lengthy and complicated output ...";
ostr.flush().setf(ios_base::unitbuf);                     //2
```

//1 Switch off the `unitbuf` flag. Alternatively, using manipulators, you can say `ostr << nounitbuf;`

//2 Flush the buffer and switch on the `unitbuf` flag again. Alternatively, you can say `ostr << flush << unitbuf;`

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Implicit Synchronization by Tying Streams

Another mechanism for automatic synchronization in certain cases is tying a stream to an output stream, as demonstrated in the code below. All input or output operations flush the tied stream's buffer before they perform the actual operation.

```
ofstream ostr("/tmp/fil");
ifstream istr("/tmp/fil");
ostream* old_tie = istr.tie(&ostr);                          //1

while (some_condition)
{ ostr << " some output ";
  string s;
  while (istr >> s)                                          //2
     // process input ;
}

istr.tie(old_tie);                                           //3
```

//1 The input stream `istr` is tied to the output stream `ostr`. The `tie()` function returns a pointer to the previously tied output stream, or `0` if no output stream is tied.

//2 Before any input is done, the tied output stream's buffer is flushed so that the result of previous output operations to `ostr` is available in the external file `/tmp/fil`.

//3 The previous tie is restored.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Synchronizing the Predefined Standard Streams

The predefined streams `cin`, `cout`, `cerr`, and `clog` are examples of synchronized streams:

- `cin` is tied to `cout`; that is, before each input operation on `cin`, the output stream `cout` is forced to flush its buffer.
- `cerr` is synchronized using the `unitbuf` format flag; that is, after each output to `cerr`, its buffer is flushed.
- `clog` is connected to the same output channel and thus behaves like `cerr`, except that it is not synchronized with any of the other standard streams; that is, it does not have the `unitbuf` flag set.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Synchronization with the C Standard I/O

The predefined C++ streams `cin`, `cout`, `cerr`, and `clog` are associated with the standard C files `stdin`, `stdout`, and `stderr`, as we saw in Section 7.1. This means that insertions into `cout`, for instance, go to the same file as output to `stdout`. By default, input and output to the predefined streams are synchronized with read or write operations on the standard C files. The effect is that input and output operations are executed in the order of invocation, independently of whether the operations used the predefined C++ streams or the standard C files.

This synchronization is time-consuming and thus might not be desirable in all situations. You can switch it off by calling:

```
sync_with_stdio(false);
```

After such a call, the predefined streams operate independently of the C standard files, with possible performance improvements in your C++ stream operations. However, you should call `sync_with_stdio()` prior to any input or output operation on the predefined streams, because otherwise the effect of calling `sync_with_stdio()` is implementation-defined.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 15: Stream Storage for Private Use

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Adding Data to a Stream

As we showed in previous sections, a stream carries a number of data items: an error state, a format state, a locale, an exception mask, information about tied streams, and a stream buffer, to mention a few. Sometimes it is useful and necessary to store *additional* data in a stream.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

An Example: Storing a Date Format String

Consider the inserter and extractor we defined for a date class in Section 11.3. The input and output operations were internationalized and relayed the task of date formatting and parsing to the stream's locale. Here, however, the rules for formatting and parsing were fixed, making them much more restricted than the features available in the standard C library, for example.

In the standard C library, you can specify format strings, similar to those for `prinft()` and `scanf()`, that describe the rules for parsing and formatting dates.18 For example, the format string `"%A, %B %d, %Y"` stands for the rule that a date must consist of the name of the weekday, the name of the month, the day of the month, and the year-as in Friday, July 12, 1996.

Now imagine you want to improve the input and output operations for the date class by allowing specification of such format strings. How can you do this? Other format information is stored in the stream's format state; consequently, you may want to store the format string for dates somewhere in the stream as well. And indeed, you can.

Streams have an array for private use. An array element is of a `union` type that allows access as a `long` or as a pointer to `void`.19 The array is of unspecified size, and new memory is allocated as needed. In principle, you can think of it as infinitely long.

You can use this array to store in a stream whatever additional information you might need. In our example, we would want to store the format string.

The array can be accessed by two functions: `iword()` and `pword()`. Both functions take an index to an array element and return a reference to the respective element. The function `iword()` returns a reference to `long`; the function `pword()` allows access to the array element as a pointer to `void`.

Indices into the array are maintained by the `xalloc()` function, a static function in class `ios_base` that returns the next free index into the array.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Another Look at the Date Format String

We would like to store the date format string in the iostream storage through `iword()` and `pword()`. In this way, the input and output operations for `date` objects can access the format string for parsing and formatting. Format parameters are often set with manipulators (see Section 7.3.2), so we should add a manipulator that sets the date format string. It could be used like this:

```
date today;
ofstream ostr;
// ...
ostr << setfmt("%D") << today;
```

Here is a suggested implementation for such a manipulator:

```
class setfmt : public smanip<const char*>
{
 public:
  setfmt(const char* fmt)
  : smanip<const char*>(setfmt_,fmt) {}
 private:
  static const int datfmtIdx ;                                  //1

  static ios_base& setfmt_(ios_base& str, const char* fmt)
  {
    str.pword(datfmtIdx) = (void*) fmt;                         //2
    return str;
  }

  template<class charT, class Traits>
  friend basic_ostream<charT, Traits> &                         //3
  operator << (
  basic_ostream<charT, Traits >& os, const date& dat);
};

const int setfmt::datfmtIdx = ios_base::xalloc();               //4
```

The technique applied to implement the manipulator is described in detail in Example 2 of Section 12.3, so we won't repeat it here. But regarding this manipulator and the private use of iostream storage, there are other interesting details:

//1 The manipulator class owns the index of the element in the iostream storage where we want to store the format string. It is initialized in `//4` by a call to `xalloc()`.

//2 The manipulator accesses the array `pword()` using the index `datfmtIdx`, and stores the pointer to the date format string.[20] Note that the reference returned by `pword()` is only used for *storing* the pointer to the date format string. Generally, you should never store a reference returned by `iword()` or `pword()` in order to access the stored data through this reference later on. This is because these references can become invalid once the array is reallocated or copied. (See the *Class Reference* for more details.)

//3 The inserter for `date` objects needs to access the index into the array of pointers, so that it can read the format string and use it. Therefore, the inserter must be declared as a friend. In principle, the extractor must be a friend, too; however, the standard C++ locale falls short of supporting the use of format strings like the ones used by the standard C function `strptime()`. Hence, the implementation of a date extractor that supports date format strings would be a lot more complicated than the implementation for the inserter, which can use the stream's locale. We have omitted the extractor for the sake of brevity.

//4 Initializes the index of elements in istream storage where the format string is kept.

The inserter for `date` objects given below is almost identical to the one we described in Section 11.5.1:

```
template<class charT, class Traits>
basic_ostream<charT, Traits> &
operator << (basic_ostream<charT, Traits >& os, const date& dat)
{
 ios_base::iostate err = 0;
 char*  patt = 0;
```

©Copyright 1999 Rogue Wave Software, Inc.

```
int    len  = 0;
charT* fmt  = 0;

try {
 typename basic_ostream<charT, Traits>::sentry opfx(os);

 if(opfx)
 {
    patt = (char*) os.pword(setfmt.datfmtIdx);              //1
    len  = strlen(patt);
    fmt  = new charT[len];

    use_facet<ctype<charT> >(os.getloc()).
        widen(patt, patt+len, fmt);

    if (use_facet<time_put<charT
                ,ostreambuf_iterator<charT,Traits> > >
         (os.getloc())
         .put(os,os,os.fill(),&dat.tm_date,fmt,fmt+len)     //2
         .failed()
       )
         err = ios_base::badbit;
    os.width(0);
 }
} //try
catch(...)
{
   delete [] fmt;
   bool flag = FALSE;
   try {
          os.setstate(ios_base::failbit);
       }
   catch( ios_base::failure ) { flag= TRUE; }
   if ( flag ) throw;
}

delete [] fmt;
if ( err ) os.setstate(err);

return os;
}
```

The only change from the previous inserter is that the format string here is read from the iostream storage (in statement `//1`) instead of being the fixed string `"%x"`. The format string is then provided to the locale's time formatting facet (in statement `//2`).

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Caveat

Note that the solution suggested here has a pitfall.

The manipulator takes the format specification and stores it. The inserter retrieves it and uses it. In such a situation, the question arises: Who owns the format string? In other words, who is responsible for creating and deleting it and hence controlling its lifetime? Neither the manipulator nor the inserter can own it because they share it.

We solved the problem by requiring the format specification to be created and deleted by the iostream user. Only a pointer to the format string is handed over to the manipulator, and only this pointer is stored through `pword()`. Also, we do not copy the format string because it would not be clear who-the manipulator or the inserter-is responsible for deleting the copy. Hence the iostream user has to monitor the format string's lifetime, and ensure that the format string is valid for as long as it is accessed by the inserter.

This introduces a subtle lifetime problem in cases where the date format is a variable instead of a constant: the stream might be a static stream and hence live longer that the date format variable. This is a problem you always deal with when storing a pointer or reference to additional data instead of copying the data.

However, this subtle problem does not impose an undue burden on the user of our `setfmt` manipulator. If a static character sequence is provided, as in:

```
cout << setfmt("%A, %B %d, %Y") << today;
```

the `setfmt` manipulator can be used safely, even with static streams like `cout`.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 16: Registration of Callback Functions

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Defining Callback Functions

The Standard C++ Library iostreams facility uses callback functions to allow a program to respond to certain events. A program registers a callback by providing an ordinary function with an appropriate signature and calling `ios_base::register_callback` with a pointer to the function and an index into the streams parray. The callback function is then called any time one of the following occurs:

- Destruction of a stream; `ios_base::~ios_base()` is called.
- Imbuing a locale; `ios_base::imbue()` is called.
- Copying the stream state; `ios_base::cpyfmt()` is called.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

An Example

A program can register a callback function on a particular stream as follows:

```
// Callback function
void show_event(ios_base::event e, iosbase& io, int index)
{
  if (e == ios_base::imbue_event)
    cout << "imbue called" << endl;
  else if (e == ios_base::erase_event)
    cout << "stream destroyed" << endl;
  else
    cout << "cpyfmt called" << endl;
}

ostringstream s;
s.register_callback(my_callback,0);   //1
s.imbue(locale::global());            //2
```

//1 The function `show_event` is now called with either `ios_base::erase_event`, `ios_base::imbue_event`, or `ios_base::copyfmt_event` as the first argument, depending on whether the destruction of the stream, the imbuing of a new locale, or a call to `cpyfmt` initiated the callback.

//2 This causes `show_event` to be called. The first argument is `ios_base::imbue_event`; the second argument is a reference to the stream where the event occurred, which is `s` in this case; and the third argument is always the index provided in the call to `register_callback`, which is `0` in this case.

If more than one function is registered, functions are called in the opposite order of registration.

Please refer to Section 5.9.2 for an example using callback functions with a user-defined stream inserter.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 17: Creating New Stream Classes by Derivation

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Deriving a New Stream Type

Sometimes it is useful to derive a stream type from the standard iostreams. This is the case when you want to add data members or functions, or modify the behavior of a stream's I/O operations.

In Chapter 15 we learned that additional data can be added to a stream object by using `xalloc()`, `iword()`, and `pword()`. However, this solution has a certain weakness in that only a pointer to the additional data can be stored and someone else has to worry about the actual memory.

This weakness can be overcome by deriving a new stream type that stores the additional data as a data member. Let's consider again the example of the `date` inserter and the `setfmt` manipulator from Section 15.3. Here let's derive a new stream that has an additional data member for storing the format string together with a corresponding member function for setting the date format specification.[21] Again, we confine the example to the inserter of the `date` object and omit the extractor. Instead of inserting into an output stream, as we did before, we now use a new type of stream called `odatstream`:

```
date today;
odatstream ostr(cout);
// ...
ostr << setfmt("%D") << today;
```

In the next sections, we explore how we can implement such a derived stream type.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Choosing a Base Class

The first question is: Which of the standard stream classes shall be the base class? The answer fully depends on the kind of addition or modification you want to make. In our case, we want to add formatting information, which depends on the stream's character type since the format string is a sequence of tiny characters. As we show later on, the format string must be expanded into a sequence of the stream's character type for use with the stream's locale. Consequently, a good choice for a base class is class `basic_iostream <charT,Traits>`, and since we want the format string to impact only output operations, the best choice is class `basic_ostream <charT,Traits>`.

In general, you choose a base class by looking at the kind of addition or modification you want to make and comparing it with the characteristics of the stream classes.

- Choose `ios_base` if you add information and services that do not depend on the stream's character type.
- Choose `basic_ios<charT, Traits>` if the added information does depend on the character type, or requires other information not available in `ios_base`, such as the stream buffer.
- Derive from the stream classes `basic_istream <charT,Traits>`, `basic_ostream <charT,Traits>`, or `basic_iostream <charT, Traits>` if you want to add or change the input and output operations.
- Derive from the stream classes `basic_(i/o)fstream <charT,Traits>`, or `basic_(i/o)stringstream <charT, Traits, Allocator>` if you want to add or modify behavior that is file- or string-related, such as the way a file is opened.

Derivations from `basic_istream <charT,Traits>`, `basic_ostream <charT,Traits>`, or `basic_iostream <charT, Traits>` are the most common cases, because you typically want to modify or add input and output operations.

If you derive from `ios_base` or `basic_ios<charT, Traits>`, you do not inherit any input and output operations; you do this if you really want to reimplement all of them or intend to implement a completely different kind of input or output operation, such as unformatted binary input and output.

Derivations from file or string streams such as `basic_(i/o)fstream <charT,Traits>` or `basic_(i/o)stringstream <charT, Traits, Allocator>` are equally rare, because they make sense only if file- or string-related data or services must be added or modified.

NOTE: Choose basic_istream <charT,Traits>, basic_ostream <charT,Traits>, or basic_iostream <charT, Traits> as a base class when deriving new stream classes, unless you have good reason not to do so.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Construction and Initialization

All standard stream classes have class `basic_ios<charT,Traits>` as a virtual base class. In C++, a virtual base class is initialized by its most derived class; that is, our new `odatstream` class is responsible for initialization of its base class `basic_ios<charT,Traits>`. Now class `basic_ios<charT,Traits>` has only one public constructor, which takes a pointer to a stream buffer. This is because class `basic_ios<charT,Traits>` contains a pointer to the stream buffer, which has to be initialized when a `basic_ios` object is constructed. Consequently, we must figure out how to provide a stream buffer to our base class. Let's consider two options:

- Derivation from file stream or string stream classes; that is, class `(i/o)fstream<>` or class `(i/o)stringstream<>`, and
- Derivation from the stream classes `basic_(i/o)stream<>`.

Derivation from File Stream or String Stream Classes Like (i/o)fstream<> or (i/o)stringstream<>

The file and string stream classes contain a stream buffer data member and already monitor the initialization of their virtual base initialization by providing the pointer to their own stream buffer. If we derive from one of these classes, we do not provide another stream buffer pointer because it would be overwritten by the file or string stream's constructor anyway. (Remember that virtual base classes are constructed before non-virtual base classes regardless of where they appear in the hierarchy.) Consider:

```
template <class charT, class Traits=char_traits<charT> >
class MyOfstream : public basic_ofstream<charT,Traits> {
 public:
   MyOfstream(const char* name)
   : basic_ios<charT,Traits>(...streambufptr...)
   , basic_ofstream<charT,Traits>(name) {}
   // . . .
};
```

The order of construction would be:

```
basic_ios(basic_streambuf<charT,Traits>*)
basic_ofstream(const char*)
basic_ostream(basic_streambuf<charT,Traits>*)
ios_base()
```

In other words, the constructor of `basic_ofstream` overwrites the stream buffer pointer set by the constructor of `basic_ios`.

To avoid this dilemma, class `basic_ios<charT,Traits>` has a protected default constructor in addition to its public constructor. This default constructor, which requires a stream buffer pointer, doesn't do anything. Instead, there is a protected initialization function `basic_ios<charT,Traits>::init()` that can be called by any class derived from `basic_ios<charT,Traits>`. With this function, initialization of the `basic_ios<>` base class is handled by the stream class that actually provides the stream buffer-in our example, `basic_ofstream<charT,Traits>`. It calls the protected `init()` function:

```
template <class charT, class Traits=char_traits<charT> >
class MyOfstream : public basic_ofstream<charT,Traits> {
 public:
   MyOfstream(const char* name)
   : basic_ofstream<charT,Traits>(name) {}
   // . . .
};
```

The order of construction and initialization is:

```
basic_ios()
basic_ofstream(const char*)
basic_ostream()
```

©Copyright 1999 Rogue Wave Software, Inc.

```
which calls:
  basic_ios<charT,Traits>::init(basic_streambuf<charT,Traits>*)
ios_base()
```

Derivation from the Stream Classes basic_(i/o)stream<>

The scheme for deriving from the stream classes is slightly different in that you must always provide a pointer to a stream buffer. This is because the stream classes do not contain a stream buffer, as the file or string stream classes do. For example, a class derived from an output stream could look like this:

```
template <class charT, class Traits=char_traits<charT> >
class MyOstream : public basic_ostream<charT,Traits> {
 public:
   MyOstream(basic_streambuf<charT,Traits>* sb)
   : basic_ostream<charT,Traits>(sb) {}
   // . . .
};
```

There are several ways to provide the stream buffer required for constructing such a stream:

- **Create the stream buffer independently, before the stream is created.** Here is a simple example in which a file buffer is created as a separate object and used by the derived stream:

```
basic_filebuf<char> strbuf;
strbuf.open("/tmp/xxx");
MyOstream<char> mostr(&strbuf);
mostr << "Hello world\n";
```

- **Take the stream buffer from another stream.** In the example below, the stream buffer is "borrowed" from the standard error stream `cerr`:

```
MyOstream<char,char_traits<char> > mostr(cerr.rdbuf());
mostr << "Hello world\n";
```

Remember that the stream buffer is now shared between `mostr` and `cerr` (see Section 13.3 for details).

- **Contain the stream buffer in the derived stream, either as a data member or inherited**. It is typically preferred when a new stream buffer type is used along with the new stream type.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The Example

Let's return now to our example, in which we are creating a new stream class by derivation.

The Derived Stream Class

Let us derive a new stream type `odatstream` that has an additional data member `fmt_` for storing a date format string, together with a corresponding member function `fmt()` for setting the date format specification.

```
template <class charT, class Traits=char_traits<charT> >
class odatstream : public basic_ostream <charT,Traits>
{
public:
 odatstream(basic_ostream<charT,Traits>& ostr,
            const char* fmt = "%x")                          //1
 : basic_ostream<charT,Traits>(ostr.rdbuf())
 {
   fmt_=new charT[strlen(fmt)];
   use_facet<ctype<charT> >(ostr.getloc()).
       widen(fmt, fmt+strlen(fmt), fmt_);                    //2
 }

 basic_ostream<charT,Traits>& fmt(const char* f)             //3
 {
   delete[] fmt_;
   fmt_=new charT[strlen(f)];
   use_facet<ctype<charT> >(os.getloc()).
       widen(f, f+strlen(f), fmt_);
   return *this;
 }

 charT const* fmt() const                                   //4
 {
  charT * p = new charT[Traits::length(fmt_)];
  Traits::copy(p,fmt_,Traits::length(fmt_));
  return p;
 }
 ~odatstream()                                              //5
 { delete[] fmt_; }


private:
 charT* fmt_;                                               //6


 template <class charT, class Traits>                       //7
 friend basic_ostream<charT, Traits> &
 operator << (basic_ostream<charT, Traits >& os, const date& dat);
};
```

//1 A date output stream borrows the stream buffer of an already existing output stream, so that the two streams share the stream buffer.

The constructor also takes an optional argument, the date format string. This is always a sequence of tiny characters.

//2 The format string is widened or translated into the stream's character type `charT`. This is because the format string is provided to the time facet of the stream's locale, which expects an array of characters of type `charT`.

//3 This version of function `fmt()` allows you to set the format string.

//4 This version of function `fmt()` returns the current format string setting.

//5 The date stream class needs a destructor that deletes the format string.

//6 A pointer to the date format specification is stored as a private data member `fmt_`.

//7 The inserter for dates must access the date format specification. For this reason, we make it a friend of class `odatstream`.

©Copyright 1999 Rogue Wave Software, Inc.

The Date Inserter

We would like to be able to insert `date` objects into all kinds of output streams. Whenever the output stream is a date output stream of type `odatstream`, we would also like to take advantage of its ability to carry additional information for formatting date output. How can this be achieved?

It would be ideal if the inserter for `date` objects were a virtual member function of all output stream classes that we could implement differently for different types of output streams. For example, when a date object is inserted into an `odatstream`, the formatting would use the available date formatting string; when inserted into an arbitrary output stream, default formatting would be performed. Unfortunately, we cannot modify the existing output stream classes, since they are part of a library you will not want to modify.

This kind of problem is typically solved using dynamic casts. Since the stream classes have a virtual destructor, inherited from class `basic_ios`, we can use dynamic casts to achieve the desired virtual behavior.[22]

Here is the implementation of the date inserter:

```
template<class charT, class Traits>
basic_ostream<charT, Traits> &
operator << (basic_ostream<charT, Traits >& os, const date& dat)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
    charT* fmt;
    charT buf[3];

    try {                                                    //1
      odatstream<charT,Traits>*
      p = dynamic_cast<odatstream<charT,Traits>*>(&os);      //2
    }
    catch (bad_cast)                                         //3
    {
      char patt[3] = "%x";

      use_facet(os.getloc(),
        (ctype<charT>*)0).widen(patt,patt+3,buf);
    }
    fmt = (p) ? p->fmt_ : buf;                               //4

    if (use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >(os.getloc())
       .put(os,os,os.fill(),&dat.tm_date,fmt,fmt+Traits::length(fmt)).failed())
       err  = ios_base::badbit;
    os.width(0);
  }
 } //try
 catch(...)
 {
    bool flag = FALSE;
    try {
           os.setstate(ios_base::failbit);
     }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }

 if ( err ) os.setstate(err);

 return os;
}
```

//1 We perform a dynamic cast in statement `//2`. A dynamic cast throws an exception in case of mismatch. Naturally, we do not want to confront our user with `bad_cast` exceptions because the mismatch does not signify an error condition, but only that the default formatting is performed. For this reason, we try to catch the potential `bad_cast` exception.

//2 This is the dynamic cast to find out whether the stream is a date stream or any other kind of output stream.

//3 In case of mismatch, we prepare the default date format specification `"%x"`.

//4 If the stream is not of type `odatstream`, the default format specification prepared in the catch clause is used.

©Copyright 1999 Rogue Wave Software, Inc.

Otherwise, the format specification is taken from the private data member `fmt_`.

The Manipulator

The date output stream has a member function for setting the format specification. Analogous to the standard stream format functions, we would like to provide a manipulator for setting the format specification. This manipulator affects only output streams. Therefore, we must define a manipulator base class for output stream manipulators, `osmanip`, along with the necessary inserter for this manipulator. We do this in the code below. See Section 12.3.3 for a detailed discussion of the technique we are using here:

```
template <class Ostream, class Arg>
class osmanip {
  public:
    typedef Ostream ostream_type;
    typedef Arg argument_type;

    osmanip(Ostream& (*pf)(Ostream&, Arg), Arg arg)
    : pf_(pf) , arg_(arg) { ; }

  protected:
    Ostream&     (*pf_)(Ostream&, Arg);
    Arg          arg_;

  friend Ostream&
    operator<< (Ostream& ostr, const osmanip<Ostream,Arg>& manip);
};

template <class Ostream, class Arg>
Ostream& operator<< (Ostream& ostr,const osmanip<Ostream,Arg>& manip)
{
   (*manip.pf_)(ostr,manip.arg_);
   return ostr;
}
```

After these preliminaries, we can now implement the `setfmt` manipulator itself:

```
template <class charT, class Traits>
inline basic_ostream<charT,Traits>&
sfmt(basic_ostream<charT,Traits>& ostr, const char* f)        //1
{
 try {                                                        //2
 odatstream<charT,Traits>* p = dynamic_cast<odatstream<charT,Traits>*>(&ostr);
}
catch (bad_cast)                                              //3
{ return ostr; }

 p->fmt(f);                                                   //4
 return ostr;
}

template <class charT,class Traits>
inline osmanip<basic_ostream<charT,Traits>,const char*>
setfmt(const char* fmt)
{ return osmanip<basic_ostream<charT,Traits>,const char*>(sfmt,fmt); }  //5
```

//1 The function `sfmt()` is the function associated with the `setfmt` manipulator. Its task is to take a format specification and hand it over to the stream. This happens only if the stream is a date output stream; otherwise, nothing is done.

//2 We determine the stream's type through a dynamic cast. As it would be rather drastic to let a manipulator call result in an exception thrown, we catch the potential `bad_cast` exception.

//3 In case of mismatch, we don't do anything and simply return.

//4 In case the stream actually is a date output stream, we store the format specification by calling the stream's `fmt()` function.

//5 The manipulator itself is a function that creates an output manipulator object.

Remark on Performance

The solution suggested in Section 17.4.3 uses dynamic casts and exception handling to implement the date inserter and the date format manipulator. Although this technique is elegant and makes proper use of the C++ language, it might introduce some loss in runtime performance due to the use of exception handling. This is particularly true since the

©Copyright 1999 Rogue Wave Software, Inc.

dynamic cast expression, and the exception it raises, is used as a sort of branching statement. In other words, the "exceptional" case occurs relatively often and is not really an exception.

If optimal performance is important, you can choose an alternative approach: in the proposed solution that uses dynamic casts, extend the date inserter for arbitrary output streams `basic_ostream<charT,Traits>& operator<< (basic_ostream <charT,Traits>&, const date&)` so that it formats dates differently, depending on the type of output stream. Alternatively, you can leave the existing date inserter for output streams unchanged and implement an additional date inserter that works for output *date* streams only; its signature would be `odatstream<charT,Traits>& operator<< (odatstream<charT,Traits>&, const date&)`. Also, you would have two manipulator functions, one for arbitrary output streams and one for output date streams only, that is, `basic_ostream<charT,Traits>& sfmt (basic_ostream<charT,Traits>&, const char*)` and `odatstream<charT,Traits>& sfmt (odatstream<charT,Traits>&, const char*)`. In each of the functions for date streams, you would replace those operations that are specific for output *date* streams.

This technique has the drawback of duplicating most of the inserter's code, which in turn might introduce maintenance problems. The advantage is that the runtime performance is likely to be improved.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Using iword/pword for RTTI in Derived Streams

In the previous section, we discussed an example that used runtime-type identification (RTTI) to enable a given input or output operation to adapt its behavior based on the properties of the respective stream type.

Before RTTI was introduced into the C++ language in the form of the new style casts `dynamic_cast<>`, the problem was solved using `iword()`, `pword()`, and `xalloc()` as substitutes for RTTI. We describe this old-fashioned technique only briefly because, as the previous example suggests, the use of dynamic casts is clearly preferable over the RTTI substitute. Still, the traditional technique might be useful if your current compiler does not yet support the new-style casts.

The basic idea of the traditional technique is that the stream class and all functions and classes that need the runtime type information, like the inserter and the manipulator function, agree on two things:

- An index into the arrays for additional storage; in other words, *Where* do I find the RTTI?
- The content or type identification that all concerned parties expect to find there; in other words, *What* will I find?

In the following sketch, the derived stream class reserves an index into the additional storage. The index is a static data member of the derived stream class, and identifies all objects of that stream class. The content of that particular slot in the stream's additional storage, which is accessible through `pword()`, is expected to be the respective stream object's `this` pointer.

Here are the modifications to the derived class `odatstream`:

```
template <class charT, class Traits=char_traits<charT> >
class odatstream : public basic_ostream <charT,Traits>
{
public:
 static int xindex()                                          //1
 {
  static int inited = 0;
  static int value = 0;
  if (!inited)
  {
     value = xalloc();
     inited++;
  }
  return value;
 }

 odatstream(basic_ostream<charT,Traits>& ostr,const char* fmt = "%x")
 : basic_ostream<charT,Traits>(ostr.rdbuf())
 {
   pword(xindex()) = this;                                    //2

   fmt_=new charT[strlen(fmt)];
   use_facet<ctype<charT> >(ostr.getloc()).widen(fmt, fmt+strlen(fmt), fmt_);
 }

 // ... other member, as in the previous section ...
};
```

//1 The static function `xindex()` is concerned with reserving the index into the arrays for additional storage. It also serves as the access function to the index.

//2 The reserved slot in the arrays for additional storage is filled with the object's own address.

Here are the corresponding modifications to the manipulator:

```
template <class charT, class Traits>
inline basic_ostream<charT,Traits>&
sfmt(basic_ostream<charT,Traits>& ostr, const char* f)
{
 if (ostr.pword(odatstream<charT,Traits>::xindex()) == &ostr) //1
    ((odatstream<charT,Traits>&)ostr).fmt(f);
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 return ostr;
}
```

//1 The manipulator function checks whether the content of the reserved slot in the stream's storage is the stream's address. If it is, the stream is considered to be a date output stream.

Note that the technique described in this section is not safe. There is no way to ensure that date output streams and their related functions and classes are the only ones that access the reserved slot in a date output stream's additional storage. In principle, every stream object of any type can access all entries through `iword()` or `pword()`. It's up to your programming discipline to restrict access to the desired functions. It is unlikely, however, that all streams will make the same assumptions about the storage's content. Instead of agreeing on each stream object's address as the run-time-type identification, we also could have stored certain integers, pointers to certain strings, etc. Remember, it's the combination of reserved index *and* assumed content that represents the RTTI substitute.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 18: Stream Buffers

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

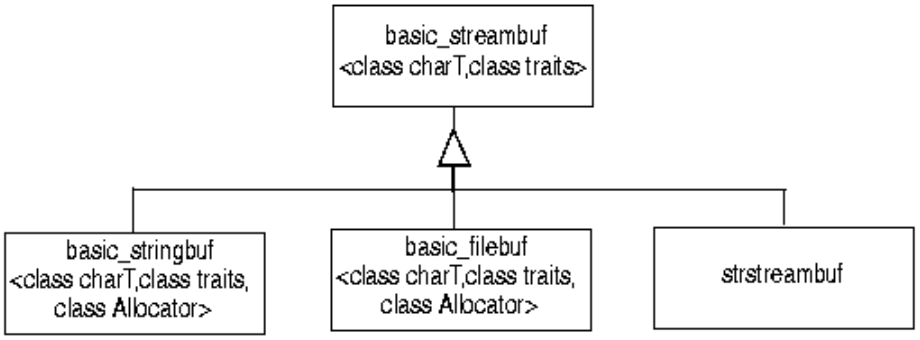
Class streambuf: the Sequence Abstraction

Stream buffers provide the transport and character code conversion capabilities of the Standard C++ Library iostreams. A stream buffer serves as a source and/or sink for the formatting layer represented by the streams themselves. The buffer in turn sends characters on to a file or stores them directly in a string, memory array, or some other destination; it also receives characters from strings, memory arrays, and files. A stream buffer need not handle both input and output, however; it may specialize in read operations or write operations. A stream buffer might also alter or manipulate data in some appropriate way, as we'll see in Section 18.2 when we derive a new stream buffer class.

The streambuf Hierarchy

As with streams, stream buffers are organized in a class hierarchy, albeit a much simpler one. To refresh your memory, Figure 36 below repeats the class hierarchy given previously in Figure 26:

Figure 36 -- The streambuf class hierarchy



Class ***basic_streambuf*** provides the base abstraction for a one-dimensional character array with theoretically unlimited capacity. All other stream buffers derive from this class. Class ***basic_streambuf*** also defines the basic interface between streams and stream buffers. Much of this interface delegates implementation to a set of protected virtual functions that allow derived classes to determine how things actually work. While these functions in ***basic_streambuf*** all have default behavior, a ***basic_streambuf*** object is not itself useful for a transport layer, since it doesn't provide a public constructor. Instead, a more specific kind of stream buffer is derived from the ***basic_streambuf*** class.

As shown in the class hierarchy, the iostreams facility defines three different derived stream buffers. These three are used respectively for controlling input/output to files, strings, and character arrays in memory:

- Class ***basic_filebuf*** provides a transport layer for interfacing streams with files.
- Class ***basic_stringbuf*** provides a transport layer for interacting directly with strings in memory.
- Class ***strstreambuf*** allows writing to and reading from character arrays.

Each of these derived stream buffers implements behavior specific to its task, and extends the ***streambuf*** interface to accommodate the specific needs of its source/destination. For instance, ***basic_filebuf*** provides `open` and `close` functions for performing those operations on an underlying file, and ***basic_stringbuf*** provides a `str()` function that returns a copy of the underlying string.

Every stream buffer implements a character array in memory that represents a *portion* of the data passing through the stream--the portion that is currently buffered. The buffer maintains both a `put` area, which contains buffered characters written to the stream, and a `get` area, which contains buffered characters available for reading. Either of these may be empty, depending on the type of stream (that is, read or write only), or the stream state (for example, at the end of a file).

©Copyright 1999 Rogue Wave Software, Inc.

When the put area becomes full, the characters in that area are written out using the protected virtual function overflow(). When the get area is emptied, a new set of characters is read in using the protected virtual function underflow(). In this way the actual reading and writing of characters is delegated to a derived class as necessary. For example, a ***filebuf*** has an implementation of overflow that writes characters out to a file, while a ***stringbuf*** simply copies characters into a string whenever overflow is called.

Seeking operations and the *sync operation* are handled in the same way. The sync operation ensures that the state of the stream buffer and the underlying source/sink are synchronized.

The streambuf Interface

Class ***basic_streambuf*** defines a public interface for reading, writing, seeking, querying, and localization. Most of the public functions that define this interface actually delegate to protected virtual functions so that specific behavior is implemented by derived classes. The *Class Reference* contains detailed descriptions of all of these functions. Meanwhile, the public interface, and the way in which functions delegate to virtual functions, is described below:

For reading:

in_avail() returns the number of characters currently in the buffer that are available for reading, or an estimate of the number of characters available in the underlying source if the buffer is currently empty. If an estimate cannot be obtained, as is the case with the default stream cin, then this function returns -1.

snextc() moves the current position forward in the buffer one character and returns the character it now points to, or returns ios_base::eof if at end of file.

sbumpc() returns the character currently pointed to in the buffer, then increments the current position by one.

sgetc returns the character at the current position. This function does not change the current position.

sgetn(char_type* s, streamsize n) copies up to n characters from the buffer to the character array pointed to by s. This function delegates to the protected virtual function xsgetn().

Note that the last four functions all use the protected virtual function underflow to fetch new characters if none is currently available in the buffer.

sungetc and dsputbackc(char_type) both move the current pointer back one step if possible. If it's not possible to back up, say, because we're at the beginning of a buffer attached to stdin, then both functions return the result of calling the protected virtual function pbackfail(). The function sputback also returns pbackfail() if the previous character in the buffer does not match the function's argument.

For writing:

sputc(char_type c) copies the character c into the buffer at the current position and increments the position. The protected virtual function overflow(c) is called if the write area is full.

sputn(const char_type* s, streamsize n) delegates to the protected virtual function xsputn. The effect is to copy up to n characters from s into the put area of the buffer and increment the write position that many times.

For positioning:

pubseekoff and pubseekpos delegate to their respective virtual functions, seekoff and seekpos. The behavior of these is highly dependent on the type of derived stream buffer and the type of code conversion needed. See Section 9.5 on file positioning for a description of these functions with regard to filebuf.

For locales:

pubimbue(const locale&) and getloc() set and get the character code conversion properties for a stream buffer; pubimbue actually delegates to the protected virtual function imbue. pubimbue returns the previous locale for the stream buffer, the same locale that would have been returned by getloc before a call to pubimbue.

Finally, ***streambuf*** provides a function for setting its internal character buffer and another for synchronizing the buffer and the underlying source or sink. Function pubsetbuf(char_type*,streamsize) delegates to the protected virtual function setbuf, and pubsync delegates to the protected virtual function sync.

In Section 18.2 we show how to create a new kind of stream buffer by deriving from one of the existing stream buffer classes. We re-implement one of the protected virtual functions declared by ***streambuf*** in order to modify the behavior of

©Copyright 1999 Rogue Wave Software, Inc.

a ***filebuf***.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Deriving New Stream Buffer Classes

Deriving a new ***streambuf*** class is not commonly necessary, but it can be extraordinarily useful when specialized behavior is needed. For example, consider a log book stream that starts writing at the beginning after reaching a certain size, so that the log file does not grow infinitely. In order to implement this new class, we first need to derive a new stream buffer type. The easiest way to do this is to derive from ***filebuf***, and then reimplement one of the protected virtual functions:

```
template <class charT, class traits>
class logbuffer : public std::basic_filebuf<charT,traits>
{

  std::streamsize max_size;

public:
  typedef charT            char_type;           // 1
  typedef traits::int_type int_type;
  typedef traits::off_type off_type;
  typedef traits::pos_type pos_type;
  typedef traits           traits_type;

  logbuffer(std::streamsize sz) : max_size(sz)   // 2
  { ; }

protected:
  int_type overflow(int_type c = traits::eof()); // 3

};
```

//1 All streams should have these types defined.

//2 This constructor takes a size parameter that we'll use to limit the size of the log file.

//3 The protected virtual function `overflow(char_type)` is called whenever a stream attempts to write to the stream buffer when the `put` area is full. We re-implement the function in order to reset the file pointer to the beginning whenever the file gets too large.

The `overflow` function is implemented as follows:

```
template <class charT, class traits>
logbuffer<charT,traits>::int_type
logbuffer<charT,traits>::overflow(logbuffer<charT,traits>::int_type c)
{
  using std::ios_base;
  using std::basic_filebuf;
  std::streamsize len = epptr() - pbase();                  // 1
  int_type ret = basic_filebuf<charT,traits>::overflow(c);  // 2

  if (seekoff(0,ios_base::cur) > max_size - len)            // 3
    seekoff(0,ios_base::beg);                               // 4
  return ret;                                               // 5
}
```

//1 First, retain the size of the `put` area of the buffer in a local variable. Both `epptr()` and `pbase()` are protected functions in ***basic_streambuf*** that return, respectively, the beginning and end of the `put` area. Other protected functions return the beginning and end of the `get` area, and also the current `get` and `put` positions. See the *Class Reference* for a full description of these functions.

//2 Next, call ***filebuf***'s `ovrflow` to write the `put` area of the buffer out to the file.

//3 Now use `seekoff` to get the current stream position and see if the distance between that and our maximum file size is less than the size of the `put` buffer. If it is, the next flush of the `put` area will exceed the maximum size we've set for the file.

//4 Seek back to the beginning of the `stream(file)` when the log is getting too large. Future writes to the stream will overwrite the contents of the log file starting at the beginning.

//5 Finally, return the value we got from ***filebuf***'s `overflow` function.

©Copyright 1999 Rogue Wave Software, Inc.

In order to use this new ***logbuf*** class, we'll also need a new ***logstream*** class:

```
template <class charT, class traits>
class logstream : public std::basic_iostream<charT,traits>
{
  logbuffer<charT,traits> buf;              // 1
public:
  typedef charT            char_type;       // 2
  typedef traits::int_type int_type;
  typedef traits::off_type off_type;
  typedef traits::pos_type pos_type;
  typedef traits           traits_type;

  logstream(std::streamsize sz, char* file); // 3
  logbuffer<charT,traits> *rdbuf() const     // 4
  {
    return (logbuffer<charT,traits>*)&buf;
  }
};
```

//1 This private member provides us with a ***logbuffer*** object.

//2 Again, we want to define the standard set of types.

//3 This constructor creates a stream with the given maximum size and the given file name.

//4 `rdbuf` returns a pointer to the `logbuffer`.

Finally, we implement our new log stream class as shown here:

```
template <class charT, class traits>
logstream<charT,traits>::logstream(std::streamsize sz, char* file)
    : buf(sz)
{
  using std::ios_base;

  init(&buf);                                          // 1
  if ( !buf.open(file, ios_base::out) )                // 2
    setstate(ios_base::failbit);
  buf.pubseekoff(0,ios_base::beg);                     // 3
}
```

//1 The `ios_base::init()` function initializes the base class. In part the initialization consists of installing a pointer to the ***logbuffer*** in the ***ios_base*** so base classes have access to the buffer.

//2 Open the file for writing.

//3 Always start out writing at the beginning.

We can use this new log buffer class as follows:

```
int main ()
{
  using std::char_traits;
  using std::endl;

  logstream<char,char_traits<char> > log(4000,"test.log"); // 1
  for (int i = 0; i < 1000; i++)                           // 2
    log << i << ' ';
  log.rdbuf()->pubseekoff(0,std::ios_base::beg);           // 3
  int in = 0;
  log >> in;                                               // 4
  return 0;
}
```

//1 Create a ***logstream*** object with a maximum size of 4000 characters and attach it to the file `test.log`.

//2 Write out the integers from 0 to 999 to the log file. The total number of characters required to represent these numbers along with the spaces between them will exceed the maximum size we've set.

//3 Seek to the beginning of the file.

//4 Look at the first value in the file. If we had used an ordinary ***fstream*** then this value would be `0`, the first value we wrote out. Instead, we used a logstream and wrote out more than the log can hold, so we'll get a different number since the log has wrapped around to the beginning.

©Copyright 1999 Rogue Wave Software, Inc.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Connecting iostream and streambuf Objects

In Section 17.3.2 we saw how to derive a new ***stream*** class. In Section 18.2 we saw how to derive a ***streambuf*** class, and an example of how to connect the two. In this section, we'll look a little more closely at the ways the two can be connected together safely.

This connection can be attempted in two different ways:

- By deriving from a descendent of ***ios*** which does not have a ***streambuf***, such as ***istream*** or ***ostream***, and providing it with an external ***streambuf*** which is itself a derived type.
- By deriving from a descendent of ***ios*** which provides a ***streambuf***, such as ***ifstream*** or ***ofstream***.

In the first case where the stream does not have a buffer, the ANSI/ISO standard mandates that no parent class constructors or destructors (***ios***, ***istream***, or ***ostream***) access the stream buffer. This restriction is important, since a derivation such as the following is otherwise unsafe:

```
class DerivedStreamBuf : public streambuf
{
      // ....
};

class DerivedOutputStream : public ostream
{
     public:
           DerivedOutputStream():ostream(&dsb):ios(0){} // 1
            // ....
     private:
           DerivedStreamBuf &dsb;
          // ....
};
```

The ***DerivedOutputStream*** constructor calls its parent constructors in the following order:

```
   ios()

//1 ostream(&dsb)

   DerivedStreamBuf ()

   DerivedOutputStream()
```

Looking at this order, we can see that ***ios*** and ***ostream*** were constructed before ***DerivedStreamBuf()*** was executed. Therefore the pointer `(&dsb)` passed through the ***ostream*** constructor is essentially an invalid pointer, and accessing it would be catastrophic. In the case where the derived ***stream*** has a buffer, only the descendent who provides the buffer can access it during construction or destruction. In both cases, explicitly preventing access to the stream buffer by the base class during the construction and destruction phases prevents catastrophic consequences.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 19: Defining A Code Conversion Facet

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Overview

File stream buffers are responsible for the transport of characters to and from an external device. In many cases, the character encoding used internally by your program and externally on the device will differ. Hence the file stream buffer must convert characters from one encoding to another each time it reads from or writes to the external device. (Section 2.3 gives a detailed discussion of character encodings and explains some typical code conversions. If you are not familiar with code conversions, we recommend that you read about them before delving into the details of implementing one in this section.)

A code conversion is not performed by the file stream buffer itself. This task is encapsulated in a code conversion facet. Each time the file stream buffer has to convert characters, it consults its locale's code conversion facet for the actual conversion. For this reason, file stream buffers and code conversion facets must work together closely, and the file stream buffer depends on its locale's code conversion facet.

This clear separation of responsibilities enables you to change a file stream's behavior substantially, without touching the file stream class itself. All you must do is provide a special code conversion facet. In doing so, you turn an ordinary file stream into one that converts, say, EBCDIC files on a mainframe's file system into a stream of ASCII characters for internal processing.

However, the task of implementing a code conversion facet requires a thorough understanding of the way file stream buffers and code conversion facets interact. In this section, we use two examples to explain the principles of this interaction.

Before we move on to the examples, let's get an overview of the different kinds of code conversions. As we show later on, different types of code conversions require different kinds of implementations.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Categories of Code Conversions

Code conversions fall into various categories depending on the properties of the character encodings involved. There are:

- Constant-size conversions
- Multibyte conversions, which again fall into the categories of:
    - State-independent conversions
    - State-dependent conversions

Constant-size conversions are between character encodings where all characters are of equal size. All single- or wide-character encodings are examples of such character encodings. Each single character stands for itself and can be recognized and translated independently of its context. Conversions between ASCII and EBCDIC, or Unicode and ISO10646, are examples of constant-size conversions.

Multibyte conversions involve multibyte encodings. In multibyte encodings, characters have varying size. Some mulitbyte characters consist of two or more bytes, while others are represented by just one byte.

There is a substantial difference between code conversions involving state-dependent character encodings, and conversions between state-independent encodings. (Again, see Section 2.3.)

State-dependent multibyte conversions involve one character encoding that is state-dependent. In state-dependent character encodings, character sequences can have different meanings depending on the current context. State-dependent encodings typically have *modes* and escape sequences that allow switching between modes. An example of a state-dependent character conversion is the conversion between the state-dependent JIS encoding for Japanese characters and the Unicode wide-character encoding.

State-independent multibyte conversions do not have modes. A sequence of characters can always be interpreted independently of its context. An example of a state-independent mulitbyte conversion is the conversion between EUC, which a state-independent multibyte encoding, and Unicode.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example 1: Defining a Tiny Character Code Conversion (ASCII <-> EBCDIC)

As an example of how file stream buffers and code conversion facets collaborate, we would now like to implement a code conversion facet that can translate text files encoded in EBCDIC into character streams encoded in ASCII. The conversion between ASCII characters and EBCDIC characters is a constant-size code conversion where each character is represented by one byte. Hence the conversion can be done on a character-by-character basis.

To implement and use an ASCII-EBCDIC code conversion facet, we:

1. Derive a new facet type from the standard code conversion facet type `codecvt`.
2. Specialize the new facet type for the character type `char`.
3. Implement the member functions that are used by the file buffer.
4. Imbue a file stream's buffer with a locale that carries an ASCII-EBCDIC code conversion facet.

The following sections explain these steps in detail.

Derive a New Facet Type

Here is the new code conversion facet type `AsciiEbcdicConversion`:

```
template <class internT, class externT, class stateT>
class AsciiEbcdicConversion
: public codecvt<internT, externT, stateT>
{
};
```

It is empty because we will specialize the class template for the character type `char`.

Specialize the New Facet Type and Implement the Member Functions

Each code conversion facet has two main member functions, `in()` and `out()`:

- Function `in()` is responsible for the conversion done on reading from the external device.
- Function `out()` is responsible for the conversion necessary for writing to the external device.

The other member functions of a code conversion facet used by a file stream buffer are:

- The function `always_noconv()`, which returns `TRUE` if no conversion is performed by the facet. This is because file stream buffers might want to bypass the code conversion facet when no conversion is necessary; for example, when the external encoding is identical to the internal. Our facet obviously will perform a conversion and does not want to be bypassed, so `always_noconv()` returns `FALSE` in our example.
- The function `encoding()`, which provides information about the type of conversion; that is, whether it is state-dependent or constant-size, etc. In our example, the conversion is constant-size. The function `encoding()` is supposed to return the size of the internal characters, which is 1 because the file buffer uses an ASCII encoding internally.

All public member functions of a facet call the respective, protected virtual member function, named `do_...()`. Here is the declaration of the specialized facet type:

```
class AsciiEbcdicConversion<char, char, mbstate_t>
: public codecvt<char, char, mbstate_t>
{
protected:
```

©Copyright 1999 Rogue Wave Software, Inc.

```
 result do_in(mbstate_t& state
              ,const char* from, const char* from_end, const char*& from_next
              ,char* to        , char* to_limit      , char*& to_next) const;

 result do_out(mbstate_t& state
              ,const char* from, const char* from_end, const char*& from_next
              ,char* to        , char* to_limit      , char*& to_next) const;

 bool do_always_noconv() const thow()
 { return false; };

 int do_encoding() const throw();
 { return  1; }

};
```

For the sake of brevity, we implement only those functions used by Rogue Wave's implementation of file stream buffers. If you want to provide a code conversion facet that is more widely usable, you must also implement the functions `do_length()` and `do_max_length()`.

The implementation of the functions `do_in()` and `do_out()` is straightforward. Each of the functions translates a sequence of characters in the range `[from,from_end)` into the corresponding sequence `[to,to_end)`. The pointers `from_next` and `to_next` point one beyond the last character successfully converted. In principle, you can do whatever you want, or whatever it takes, in these functions. However, for effective communication with the file stream buffer, it is important to indicate success or failure properly.

Use the New Code Conversion Facet

Here is an example of how the new code conversion facet can be used:

```
fstream inout("/tmp/fil");                                  //1
AsciiEbcdicConversion<char,char,mbstate_t> cvtfac;
locale cvtloc(locale(),&cvtfac);
inout.rdbuf()->pubimbue(cvtloc)                             //2
cout << inout.rdbuf();                                      //3
```

//1 When a file is created, a snapshot of the current global locale is attached as the default locale. Remember that a stream has two locale objects: one used for formatting numeric items, and a second used by the stream's buffer for code conversions.

//2 Here the stream buffer's locale is replaced by a copy of the global locale that has an ASCII-EBCDIC code conversion facet.

//3 The content of the EBCDIC file `"/tmp/fil"` is read, automatically converted to ASCII, and written to `cout`.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Error Indication in Code Conversion Facets

Since file stream buffers depend on their locale's code conversion facet, it is important to understand how they communicate. On writing to the external device, the file stream buffer hands over the content of its internal character buffer, partially or entirely, to the code conversion facet; that is, to its `out()` function. It expects to receive a converted character sequence that it can write to the external device. The reverse takes place, using the `in()` function, on reading from the external file.

In order to make the file stream buffer and the code conversion facet work together effectively, it is necessary that the two main functions `in()` and `out()` indicate error situations the way the file stream buffer expects them to do it.

There are four possible return codes for the functions `in()` and `out()`:

- `ok`, which should obviously be returned when the conversion went fine.
- `partial`, which should be returned when the code conversion reaches the end of the input sequence `[from,from_end)` before a new character can be created. The file stream buffer's reaction to `partial` is to provide more characters and call the code conversion facet again, in order to successfully complete the conversion.[23]
- `error`, which indicates a violation of the conversion rules; that is, the character sequence to be converted does not obey the expected rules and thus cannot be recognized and converted. In this situation, the file stream buffer stops doing anything, and the file stream eventually sets its state to `badbit` and throws an exception if appropriate.
- `noconv`, which is returned if no conversion was needed.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Example 2: Defining a Multibyte Character Code Conversion (JIS <-> Unicode)

Let us consider the example of a state-dependent code conversion. As mentioned previously, this type of conversion would occur between JIS, which is a state-dependent multibyte encoding for Japanese characters, and Unicode, which is a wide-character encoding. As usual, we assume that the external device uses multibyte encoding, and the internal processing uses wide-character encoding.

Here is what you must do to implement and use a state-dependent code conversion facet:

1. Define a new conversion state type if necessary.
2. Define a new character traits type if necessary, or instantiate the character traits template with the new state type.
3. Define the code conversion facet.
4. Instantiate new stream types using the new character traits type.
5. Imbue a file stream's buffer with a locale that carries the new code conversion facet.

These steps are explained in detail in the following sections.

Define a New Conversion State Type

While parsing or creating a sequence of multibytes in a state-dependent multibyte encoding, the code conversion facet has to maintain a conversion state. This state is by default of type `mbstate_t`, which is the implementation-dependent state type defined by the C library. If this type does not suffice to keep track of the conversion state, you must provide your own conversion state type. We show how this is done in the code below, but please note first that the new state type must have the following member functions:

- A constructor, since the argument `0` has the special meaning of creating a conversion state object that represents the initial conversion state
- Copy constructor and assignment
- Comparison for equality and inequality

Now here is the sketch of a new conversion state type:

```
class JISstate_t {
public:
                   JISstate_t( int state=0 )
                   : state_(state) { ; }

                   JISstate_t(const JISstate_t& state)
                   : state_(state.state_) { ; }

                   JISstate_t& operator=(const JISstate_t& state)
                    {
                       if ( &state != this )
                         state_= state.state_;
                       return *this;
                    }

                   JISstate_t& operator=(const int state)
                    {
                       state_= state;
                       return *this;
                    }

                   bool operator==(const JISstate_t& state) const
                    {
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                    return ( state_ == state.state_ );
                  }

                  bool operator!=(const JISstate_t& state) const
                  {
                    return ( !(state_ == state.state_) );
                  }

private:
                  int state_;

                };
```

Define a New Character Traits Type

The conversion state type is part of the character traits. Hence, with a new conversion state type, you need a new character traits type.

If you do not want to rely on a nonstandard and thus non-portable feature of the library, you must define a new character traits type and redefine the necessary types:

```
struct JIS_char_traits: public char_traits<wchar_t>
{
        typedef JISstate_t               state_type;
        typedef fpos<state_type>         pos_type;
        typedef wstreamoff               off_type;
};
```

Define the Code Conversion Facet

Just as in the first example, you must define the actual code conversion facet. The steps are basically the same as before, too: define a new class template for the new code conversion type and specialize it. The code would look like this:

```
template <class internT, class externT, class stateT>
class UnicodeJISConversion
: public codecvt<internT, externT, stateT>
{
};

class UnicodeJISConversion<wchar_t, char, JISstate_t>
: public codecvt<wchar_t, char, JISstate_t>
{
protected:

 result do_in(JISstate_t& state,
              const char*  from,
              const char*  from_end,
              const char*& from_next,
              wchar_t*     to,
              wchar_t*     to_limit,
              wchar_t*&    to_next) const;

 result do_out(JISstate_t& state,
               const wchar_t*  from,
               const wchar_t*  from_end,
               const wchar_t*& from_next,
               char*           to,
               char*           to_limit,
               char*&          to_next) const;

 bool do_always_noconv() const throw()
 { return false; };

 int do_encoding() const throw();
 { return -1; }

};
```

In this case, the function `do_encoding()` has to return `-1`, which identifies the code conversion as state-dependent. Again, the functions `in()` and `out()` must conform to the error indication policy explained under class ***codecvt*** in the *Class Reference.*

©Copyright 1999 Rogue Wave Software, Inc.

The distinguishing characteristic of a state-independent conversion is that the conversion state argument to `in()` and `out()` is used for communication between the file stream buffer and the code conversion facet. The file stream buffer is responsible for creating, maintaining, and deleting the conversion state. At the beginning, the file stream buffer creates a conversion state object that represents the initial conversion state and hands it over to the code conversion facet. The facet modifies it according to the conversion it performs. The file stream buffer receives it and stores it between two subsequent code conversions.

Use the New Code Conversion Facet

Here is an example of how the new code conversion facet can be used:

```
typedef basic_fstream<wchar_t,JIS_char_traits> JIS_fstream;   //1
JIS_fstream inout("/tmp/fil");
UnicodeJISConversion<wchar_t,char,JISstate_t> cvtfac;
locale cvtloc(locale(),&cvtfac);
inout.rdbuf()->pubimbue(cvtloc)                               //2
wcout << inout.rdbuf();                                       //3
```

//1 Our Unicode-JIS code conversion needs a conversion state type different from the default type `mbstate_t`. Since the conversion state type is contained in the character traits, we must create a new file type. Instead of `JIS_char_traits`, we could have taken advantage of the nonstandard extension to the character traits template and have used `char_traits<wchar_t,JISstate_t>`.

//2 Here the stream buffer's locale is replaced by a copy of the global locale that has a Unicode-JIS code conversion facet.

//3 The content of the JIS encoded file `"/tmp/fil"` is read, automatically converted to Unicode, and written to `wcout`.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 20: Defining Your Own Character Types

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

User-Defined Character Types

One of the fundamental differences between the new, templated iostreams and the traditional iostreams is the ability of the new iostreams to accommodate user-defined character types. It's now possible to use the C++ iostreams interface with arbitrary character-like types, as well as with ordinary `char`s.

Of course, this flexibility comes at a price. Any user-defined character type must satisfy rather severe requirements. The type really must act like an ordinary `char` in most circumstances. In addition, several classes must be defined to support the type in the iostreams environment.

Requirements for User-Defined Character Types

In general, user-defined character types must meet the following requirements:

- They must have a *public default constructor*.
- They must have a *public copy constructor*.
- They must have a *public destructor*.
- They must be *convertible to int*.
- They must be *assignable*.
- They must be *comparable for equality and ordering*.

In addition, it must be possible to convert an object of the user-defined type to an object of type `char` and vice versa. These particular conversions need not be part of the type itself. They are defined as part of the new character classification facet that must accompany the new type. We'll examine these conversions a little later on.

The following type satisfies the requirements for a user-defined type:

```
struct Echar
{
  Echar():c(0),i(0) {;}
  Echar(char cc):c(cc),i(0) {;}
  Echar(char cc,char ii):c(cc),i(ii) {;}
  operator int() { return c; }
  char c;
  char i;
};

bool operator==(const Echar& lhs, const Echar& rhs)
{ return lhs.c == rhs.c; }

bool operator!=(const Echar& lhs, const Echar& rhs)
{ return !(lhs.c == rhs.c); }

bool operator<(const Echar& lhs, const Echar& rhs)
{ return lhs.c < rhs.c; }
```


OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Defining Traits and Facets for User-Defined Types

A user-defined type requires its own traits class, its own code conversion facet, and its own character classification facet. The traits class defines a conversion `state_type` and also defines operations such as copying arrays of the type and comparing arrays. The code conversion facet provides conversion to and from simple `chars`. The `ctype` facet provides character classification and manipulation routines, including the method for converting the new type to and from simple `chars`.

The following code shows a traits class declaration for `Echar`:

```
struct Etraits
{
  typedef Echar                char_type;
  typedef long                 int_type;

  typedef long                 off_type;
  typedef mbstate_t            state_type;
  typedef fpos<state_type>     pos_type;

  static void assign (char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1,const char_type& c2);
  static bool lt (const char_type& c1, const char_type& c2);
  static int compare (const char_type* s1, const char_type* s2,
                      size_t n);
  static size_t length(const char_type *s);
  static const char_type*
      find (const char_type* s, int n, const char_type& a);
  static char_type* move (char_type* s1, const char_type* s2,
                          size_t n);
  static char_type* copy(char_type *dst,const char_type *src,
                          size_t n);
  static char_type* assign (char_type* s, size_t n,
                            const char_type& a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,const int_type& c2);
  static state_type get_state(pos_type pos);
  static int_type eof();
};
```

See the *Class Reference* section on `char_traits` for a complete description of the member functions in a traits class.

To create a new code conversion facet, you must inherit from the `codecvt` template and then provide implementations for all the protected virtual functions. Iostreams calls the protected functions through the public interface defined for `codecvt`. You must also provide a constructor taking a single `size_t` argument, and initialize `codecvt` with that argument.

A code conversion facet for `Echar` has a declaration that looks like this:

```
class Ecodecvt : public codecvt<Echar,char,Estate>
{
  public:
       _EXPLICIT Ecodecvt (size_t refs = 0) :
           codecvt<Echar,char,Estate>(refs) {;}

  protected:
          virtual ~Ecodecvt() {;}
          virtual result do_out(Estate& state, const Echar* from,
                                const Echar* from_end,
                                const Echar*& from_next,
                                char* to, char* to_limit,
                                char*& to_next) const;
         virtual result do_in(Estate& state,
                              const char* from,
                              const char* from_end,
                              const char*& from_next,
```

©Copyright 1999 Rogue Wave Software, Inc.

```
                              Echar* to, Echar* to_limit,
                              Echar*& to_next) const;
        virtual result do_unshift(Estate& state,
                                  char* to, char* to_limit,
                                  char*& to_next) const;
        virtual int do_encoding() const throw();
        virtual bool do_always_noconv() const throw();
        virtual int do_length(const Estate&, const char* from,
                              const char* end, size_t maxm) const;
        virtual int do_max_length() const throw();
  };
```

See Chapter 19 on defining a code conversion facet for more details. Also see the *Class Reference* section on `codecvt` for a complete description of member functions.

To create a character classification facet, you must inherit from the `ctype` template and provide implementations for all protected virtual functions. You must also provide a constructor taking a single `size_t` argument, and initialize `ctype` with that argument.

Note that the `widen` functions define conversions from simple `chars` to the user-defined character type, and the narrow function provides conversions from the user-defined type to simple `chars`.

A character classification facet for `Echar` has a declaration that looks like this:

```
class Ectype : public ctype<Echar>
{
     public:
       typedef Echar char_type;
       _EXPLICIT Ectype (size_t refs = 0) :
           ctype<Echar>(refs) {;}         // must pass refs onto
                                          // the ctype constructor

     protected:
       inline virtual ~Ectype ();
       virtual bool do_is(mask m, Echar c) const;
       virtual const Echar* do_is(
                                  const Echar* low,
                                  const Echar* high,
                                  mask* vec) const;
       virtual const Echar* do_scan_is(
                                       mask m, const Echar* low,
                                       const Echar* high) const;
       virtual const Echar* do_scan_not(
                                        mask m, const Echar* low,
                                        const Echar* high) const;
       virtual Echar        do_toupper(Echar e)  const;
       virtual const Echar* do_toupper(Echar* low,
                                       const Echar* high) const;
       virtual Echar        do_tolower(Echar)  const;
       virtual const Echar* do_tolower(Echar* low,
                                       const Echar* high) const;
       virtual Echar        do_widen(char) const;
       virtual const char*  do_widen(const char* lo,
                                     const char* hi,
                                     Echar* dest) const;
       virtual char         do_narrow(Echar, char dfault) const;
       virtual const Echar* do_narrow(const Echar* lo,
                                      const Echar* hi,char dfault,
                                      char* dest) const;

};
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Creating and Using Streams Instantiated on User-Defined Types

Once all the support work is finished, you can instantiate a stream class template on your type and begin to use it--well, not quite. Because the new stream requires the new facets you've defined, you must first create a new locale containing these facets and then imbue that locale on your stream. You can do this as follows:

```
typedef basic_fstream<Echar,Etraits> Estream;                // 1
locale Eloc(locale(locale(),new Ecodecvt),new Ectype);       // 2
Estream foo("foo.txt");                                      // 3
foo.imbue(Eloc);                                             // 4
```

//1 Typedef the special stream type to something convenient.

//2 Construct a new locale object and replace the `codecvt` and `ctype` facets with the ones we've defined for `Echar`. We use the constructor that takes an existing locale and a new facet to construct a locale with a new `codecvt` facet, and then use it again to get a locale with both new facets.

//3 Construct an `Estream`.

//4 Imbue the new locale containing our replacement facets onto the `Estream`.

Now we're ready to insert `Echars` into the stream and read them back in:

```
Echar e[10];
Estream::pos_type pos = foo.tellp();                          //1
foo << 10;                                                    //2
foo.seekg(pos);                                               //3
foo >> e;                                                     //4
```

//1 Get current position.

//2 Write out the integer `10`.

//3 Seek back.

//4 Read in the string `10` as `Echars`.

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 21: Locales

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Locales and Iostreams

Within the Standard C++ Library iostreams, ***locale*** is a class. Class ***locale*** is used for two distinct purposes:

- The formatting layer of iostreams uses ***locale*** facilities for numeric formatting and character classification.
- The transport layer uses ***locale*** facilities for code conversion.

In the standard iostream classes, the formatting (stream) layer uses only the ***ctype***, ***num_put***, and ***num_get*** facets, while the transport (stream buffer) layer uses only the ***codecvt*** facet. For more information on locales and facets, see Chapter 3 and the sections of the *Class Reference* devoted to the ***locale*** class and the various facets.

Each layer contains its own ***locale***, so the code conversion facility can be changed independent of numeric formatting. We simply change the ***locale*** associated with a stream buffer without altering the ***locale*** contained by the stream using that buffer. On the other hand, changing a stream's ***locale*** also changes the ***locale*** contained by the associated stream buffer.

The base class for all streams, ***ios_base***, contains a ***locale***. This ***locale*** can be accessed with the `getloc()` function and changed with the `imbue()` function. Similarly, the ***basic_streambuf*** class also contains a ***locale*** and it has a `getloc()` function and `pubimbue()` function. A stream buffer has a protected `imbue()` function. In both cases the `imbue()` function is virtual in order to allow derived classes to change how localization is implemented.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

When to Imbue a New Locale

A program cannot necessarily imbue a new ***locale*** on a stream or stream buffer at any particular time. Problems are likely if a new ***locale*** is imbued on a stream buffer with a state-dependent code conversion, since the buffer might very well be in the middle of a state-dependent conversion. In such a situation, the code conversion facet contained in the new ***locale*** would likely be at loose ends indeed. To avoid problems, follow these guidelines for imbuing locales:

- A new ***locale*** can always be imbued immediately after construction of a stream or stream buffer, in other words, before any other operation is attempted.
- A new ***locale*** can always be imbued after an operation that flushes the streambuffer.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

An Example

An example of when you would want to imbue a new locale on a stream might be the case where a file has a header that requires a different code conversion facet. The following code imbues the JIS to UNICODE conversion facet from Chapter 19 onto a stream, reads a single line, then imbues the default locale to continue processing the file:

```
wstring header;
wstring body;
typedef basic_ifstream<wchar_t,JIS_char_traits> JIS_ifstream;
JIS_fstream in("special.txt");
UnicodeJISConversion<wchar_t,char,JISstate_t> cvtfac;
locale cvtloc(locale(),&cvtfac);
in.rdbuf()->pubimbue(cvtloc);
getline(in,header);
in.rdbuf()->pubimbue(locale());
in >> body;
```


OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 22: Stream Iterators

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Definition

Stream iterators are used to access an existing input or output stream using iterators. In the Standard C++ Library, the template classes ***istream_iterator*** and ***ostream_iterator*** are commonly referred to as the stream iterators. Class ***istream_iterator*** reads successive characters from the stream for which it was constructed; ***ostream_iterator*** writes to its respective stream.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Differences between Stream Iterators and Container Iterators

In contrast to regular container iterators, stream iterators can provide only non-modifiable read-only access or write-only access to their elements. The ***istream_iterator*** is a read-only iterator that can access but not modify elements by assignment. The ***ostream_iterator***,on the other hand, provides write-only access to streams.

Each time the ***istream_iterator*** invokes `operator++`, a new value from the stream is read and stored in a `const T` value, where `T` is the type of the element being written. The `const` value, which by definition cannot be overwritten by assignment, is then available through the use of the dereferencing operator `*`. This behavior of storing elements in the iterator is unique to ***istream_iterator***s.

Also unlike container iterators, stream iterators can access elements only once, and only in the forward-moving direction, so that they can work only with one-pass algorithms. If the contents of the stream are to be read more than once, separate iterators must be created for each pass.

The ***istream_iterator*** has a template argument `Distance`, which is defined as `ptrdiff_t` but not used in the implementation. There is no distance type template argument for ***ostream_iterator***, so it does not have the notion of distance!

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Error Indication by Stream Iterators

The ***istream*** and ***ostream*** iterators by themselves have no means of indicating the success or failure of an operation. The stream being operated on should be probed for good(), eof(), fail(), or bad() conditions. This is convenient, as stream state can be checked for exception conditions that were set in
basic_ios (void exceptions(iostate except_mask)) after each input or output operation.

```
void  print()
{
      ostream_iterator<int, char, char_traits<char> > os(cout);
       try
          {
              *os = 3;  //output `3'
                os++;  // receive next output
           }
            catch(ios_base::failure & emsg)
            {
                  cerr<<emsg.what();
            }

}
```

The ***streambuf*** iterators provide a member function, `failed()`, to indicate success or failure in a preceding operation on the stream buffer.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Several Iterators on One Stream

Stream iterators can be used only with one-pass algorithms. If a stream has to be read multiple times, a new iterator must be constructed for each pass.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 23: Iostreams and Multithreading

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Multithread-Safe: Level 2

This Rogue Wave implementation of iostreams and locales is Multithread-Safe: Level 2. All public and protected functions are reentrant, and there is protection against multiple threads trying to modify static and global data. The preferred method of protection is mutex locks; the library also locks an object before writing to it. The developer is not required to explicitly lock or unlock a class object, whether static, global, or local, in order to perform a single operation on the object.

This means that iostream objects, with the exception of stream buffers, can be shared between threads of execution using a simple mutex object without explicit locking. The locking mechanism is enforced at the stream level. Therefore, all operations carried out on the stream are multithread safe, including the following:

Thread 1:	**Thread 2:**
cout << "Thread 1" << endl;	cout << " Thread 2" << endl;

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

The Locking Mechanism

The locking mechanism has been designed for maximum efficiency, with no reentrant locks needed. There are two different mutex objects involved in this scheme:

- The first mutex object is located in the class ***ios_base***. It enforces multithread safety for all formatting operations performed on the stream, for imbuing the stream with a new locale object, and for accessing the raw storage mechanism (pword, iword). All functions performing these operations lock the mutex object beforehand and release it afterwards. These operations are not time-critical and do not occur often in comparison to buffer operations like inserting a character. They are located in classes ***ios_base*** or ***basic_ios***.

- The second mutex object, located in the ***basic_streambuf*** class, protects the buffer. The locking and unlocking of this mutex object is critical, since buffer operations are on the direct path of performance issues.

It is easy to see that locking and unlocking the buffer after each independent buffer operation would be disastrous. For example, when inserting a `char*` sequence of characters, a call to an inline ***basic_streambuf*** function is made for each character inserted; therefore, the locking mechanism is carried out at a higher level. For all formatted and unformatted stream functions, the locking is performed in the ***sentry*** object constructor, and the release in the ***sentry*** object destructor. If the function does not make use of the ***sentry*** class, the lock is directly performed inside the function. This is the case with basic_istream::seekg and basic_ostream::seekg.

Consider the following example:

Thread 1:	**Thread 2:**
`cout << "Hello Thread 1" << endl;`	`cout << "Hello Thread 2" << endl;`

If **Thread 1** is the first thread locking the buffer, the sequence of characters `"Hello Thread 1"` is output to `stdout` and the lock is released; **Thread 2** then acquires the lock, outputs its sequence of characters, and releases it.

Protecting the Buffer

Notice that only one lock occurs on the ***basic_streambuf*** mutex object for each stream operation. The advantage of this scheme is obviously high performance, but the drawback is that while buffer functionality is directly accessed, the buffer is left unprotected. We therefore provide functions and manipulators to solve this problem. The following example explains how they work:

Thread 1:	**Thread 2:**
	`cout<< __lock;`
	do {
`cout << "Hello Thread 1" << endl;`	cout.rdbuf()->sputc(*t);
	} while (*t++!=0)
	cout << __unlock;

In this scheme, if **Thread 2** is the first one to execute, when it gets to the statement `cout<< __lock;`, it locks the ***basic_streambuf*** object pointed at by `cout.rdbuf()`. **Thread 1** cannot output its sequence of characters until **Thread 2** reaches the statement `cout << __unlock;`, which releases the lock. This technique is easy to use and allows high performance for both stream and buffer operations.

Locking Several Stream Operations

There is also a way to lock several stream operations within one thread in order to preserve the global order of operations carried out by several threads running concurrently. The following example illustrates this technique:

Thread 1:	**Thread 2:**
`cout << __lock;`	`cout << "Thread 2 begin" << endl;`
`cout << "Thread 1 begin" << endl;`	

©Copyright 1999 Rogue Wave Software, Inc.

```
cout << "Thread 1 completion" << endl;
cout << __unlock;
```

In this example, if **Thread 1** is the first thread locking the ***basic_streambuf*** object attached to cout, **Thread 2** cannot output its sequence of characters until **Thread 1** reaches the statement cout << __unlock;. The result is to preserve the order of the output, which is:

Thread 1 locking first:

```
Thread 1 begin
Thread 1 completion
Thread 2 begin
```

Thread 2 locking first:

```
Thread 2 begin
Thread 1 begin
Thread 1 completion
```

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The Location of Locks

Stream locks are obtained and released in public member functions of ***basic_ios***, ***basic_istream***, ***basic_ostream***, ***basic_ifstream***, ***basic_ofstream***, ***basic_istringstream***, ***basic_ostringstream***, and ***strstream***. This means that all calls to public member functions of these classes are safe.

Stream buffer locks are obtained in the constructor of the ***sentry*** object contained in both ***basic_istream*** and ***basic_ostream***, and released in the destructor. Calls to member functions of stream buffers are not safe. The buffer must be locked explicitly before the function is called, and unlocked after the function returns.

A stream buffer lock can be obtained by explicitly calling the member function `_RW_lock_buffer()` provided by ***basic_streambuf***. The lock is released by calling `_RW_release_buffer()`, or you can use the lock and unlock manipulators as shown above. These functions are provided as extensions to the Standard C++ Library definition, but are contained in the `std` namespace.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Chapter 24: Standard vs. Traditional Iostreams

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

The Character Type

You may already have used iostreams in the past-the traditional iostreams. The iostreams included in the Standard C++ Library are mostly compatible, yet slightly different from what you know. The most apparent change is that the new iostream classes are templates, taking the type of the character as a template parameter.

The traditional iostreams were of limited use. They could handle only byte streams; in other words, they read files byte per byte, and worked internally with a buffer of bytes. They had problems with languages that have alphabets containing thousands of characters. These alphabets are encoded as multibytes for storage on external devices like files, and represented as wide characters internally. They required a code conversion with each input and output.

The new templatized iostreams can handle large alphabets. These iostreams can be instantiated for one-byte skinny characters of type `char`, and for wide characters of type `wchar_t`. In fact, you can instantiate iostream classes for any user-defined character type. Chapter 20 describes in detail how this can be done.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Internationalization

Another new feature of the standard iostreams is internationalization. Traditional iostreams were incapable of adjusting to local conventions. Output of numerical items was always done following the US English conventions for number formatting. The new iostreams are internationalized to allow for local conventions. They use the standard locales described in the section on locales.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

File Streams

Connecting Files and Streams

The traditional iostreams supported a file stream constructor, taking a file descriptor that allowed connection of a file stream to an already open file. This is no longer available in the standard iostreams.

The functions `attach()` and `detach()` do not exist anymore.

The File Buffer

Due to changes in the iostream architecture, the file buffer is now contained as a data member in the file stream classes. In some old implementations, the buffer was inherited from a base class called `fstreambase`.

The old file streams had a destructor; the new file streams don't need one. Flushing the buffer and closing the file is now done by the file buffer's destructor.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

String Streams

Output string streams are always dynamic. The `str()` function does not have the functionality of freezing the string stream anymore. Instead, the string provided through `str()` is copied into the internal buffer; it is not used *as* the internal buffer. Accordingly, the string returned through `str()` is always a copy of the internal buffer. If you need to influence a string stream's internal buffering, you must do it through `pubsetbuf()`.

The classes ***strstream***, ***istrstream***, ***ostrstream***, and ***strstreambuf*** are deprecated features in the standard iostreams. They are still provided by this implementation of the standard iostreams, but will be omitted in the future.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Streams with Assign

The classes `ostream_withassign`, `istream_withassign`, and `iostream_withassign` do not exist in the standard iostreams anymore. You can only assign the data components of a stream to another stream. This is done through the functions `copyfmt()` and `rdbuf()` in class `basic_ios`.

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Top Contents Index

Chapter 25: Standard vs. Rogue Wave Iostreams

This section describes how the Rogue Wave implementation of the standard iostreams differs from the ISO/ANSI Standard C++ Library specification. You must be aware that whenever you use one of the features described here, the portability of your program is impaired. It will not conform to the standard.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Extensions

Rogue Wave's implementation of the standard iostreams has several extensions that we describe briefly in the sections below.

File Descriptors

The traditional iostreams allowed a file stream to connect to a file using a *file descriptor*. File descriptors are used by functions like `open()`, `close()`, `read()`, and `write()` that are part of most C libraries, especially on UNIX-based platforms. However, the ISO/ANSI standard for the programming language C and its library does not include these functions, nor does it mention file descriptors. In this sense, the use of file descriptors introduces platform and operating system dependencies into your program. This is exactly why the standard iostreams does not use file descriptors.

Now you might already have programs that use the file descriptor features of the traditional iostreams. And you may need to access system-specific files like pipes, which are accessible only through file descriptors. To address these concerns, Rogue Wave's implementation offers additional constructors and member functions in the file stream and file buffer classes that enable you to work with file descriptors.

The main additions are:

- Constructors that take a file descriptor rather than a file name
- An additional third parameter that allows specification of file access rights. This parameter, available on several constructors and the `open()` member functions, is not available with the standard interface. The parameter has a default, so that you usually need not worry about file protection.

Multithreaded Environments

See Chapter 23 on multithreading. Note that all multithreading features are an extension of the standard and are therefore not portable to other implementations of the library.

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Restrictions

Rogue Wave's implementation of the standard iostreams has several restrictions, most of which correspond to the limited capabilities of current compilers in handling Standard C++. These restrictions include:

- Member templates
- Explicit template argument specification (`use_facet` and `has_facet` in locale)

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Deprecated Features

- The `strstream` classes

Top Contents Index

OEM Edition, OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Appendix A: Implementation Notes

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents Index

Implementation-Dependent Behavior

The ANSI/ISO-approved standard for the C++ language and library does not specify how `pword()` and `iword()` indicate failure, so this feature is dependent on the particular implementation of the library you are using. This Rogue Wave implementation uses operator `new` for allocating these arrays, which means that `bad_alloc` will be thrown.

The standard also doesn't specify what happens if `iword()` or `pword()` are provided with an index that was not returned by a previous call to `xalloc()`. This Rogue Wave implementation allocates as much memory as necessary to provide the requested array entry.

Top Contents Index

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

Topic Index

Click on one of the letters below to jump immediately to that section of the index. If you get no response, that letter has no entries.

A B C D E F G H I J K L M N O P

Q R S T U V W X Y Z

a

b

©Copyright 1999 Rogue Wave Software, Inc.

c

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

d

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

f

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

g

h

i

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

j

l

©Copyright 1999 Rogue Wave Software, Inc.

m

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

n

o

©Copyright 1999 Rogue Wave Software, Inc.

p

©Copyright 1999 Rogue Wave Software, Inc.

r

©Copyright 1999 Rogue Wave Software, Inc.

s

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

t

u

©Copyright 1999 Rogue Wave Software, Inc.

©Copyright 1999 Rogue Wave Software, Inc.

Top Contents

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.

Endnotes

1

This example assumes that you have created a class, ***moneytype***, to represent monetary values, and that you have written iostreams insertion << and extraction >> operators for the class. Further, it assumes that these operators format and parse values using the `money_put` and `money_get` facets of the locales imbued on the streams they're operating on. See Chapter 5 for a complete example of this technique, using phone numbers rather than monetary values. The ***moneytype*** class is not part of the Standard C++ Library.

Return

1

The shift operators for the character types, like char and wchar_t, are an exception to this rule; they are global functions in the standard library namespace ::std.

Return

2

Iostreams does not prevent you from setting other invalid combinations of these flags, however.

Return

3

The standard does not specify whether the grouping information, that is contained in a stream's locale's numpunct facet, is ignored or taken into account if present. In any case, there are no manipulators that allow to switch on and off the grouping.

Return

4

The classification of a character as a white space character depends on the character set used. The extractor takes the information from the locale's ctype facet.

Return

5

The stream buffer can be created as the stream's responsibility, or the buffer can be provided from outside the stream, so inadvertently the buffer could have 0 size.

Return

6

The streams layer catches bad_alloc exceptions thrown during allocation of its internal resources, iword and pword. It would then set badbit or failbit. An exception would be thrown only if the respective bit in the exception mask asks for it. The exception is ios_failure.

Return

7

Note that each change of either the stream state or the exception mask can result in an exception thrown. This is because the functions setstate() and exception() raise an exception in case the exception mask requires it.

Return

8

See Bjarne Stroustrup, The C++ Programming Language, 3rd Edition, p.366.

Return

©Copyright 1999 Rogue Wave Software, Inc.

9

The traditional iostreams supported a constructor, taking a file descriptor, that allowed connection of a file stream to an already open file. This is not available in the standard iostreams. However, Rogue Wave's implementation of the standard iostreams provides a corresponding extension (see the *Class Reference* for file streams.).

Return

10

For output file streams the open mode out is equivalent to out|trunc, that is, you can omit the trunc flag. For bidirectional file streams, however, trunc must always be explicitly specified.

Return

11

Basically the binary mode flag is passed on to the respective operating system's service function, which means that in principle all system-specific conversions are suppressed, not only the carriage return / linefeed handling.

Return

12

This was different in the old iostreams, where you could have dynamic and static output streams. See Section 24.4 for further details.

Return

13

An alternative could be to provide Manip as a static or a global object at the user's convenience. Unfortunately, this approach would introduce the well-known order-of-initialization problems for global and static objects.

Return

14

Traditional iostreams had classes called ostream_withassign that explicitly allowed copying and assignment of stream objects.

Return

15

This feature was available in the traditional iostreams, but is not offered by the standard iostreams. Rogue Wave's implementation of the standard iostreams retains the old feature for backward compatibility with the traditional iostreams, but it is a nonstandard feature. Using it might make your application non-portable to other standard iostream libraries.

Return

16

The traditional iostreams' strstream allows you to obtain a pointer to the stream's internal buffer. Different from the standard iostreams' stringstream, it does not create a copy of the internal data. Hence, using the deprecated strstream instead of the standard stringstream spares you the overhead of creating a second copy of the data.

Return

17

In the case of input streams, the behavior of sync() is implementation-defined, that is, not standardized. The traditional iostreams had a sync() function that did the expected synchronization, that is, refilling the buffer beginning with the current file position.

Return

18

See functions strftime(), strptime(), and wcsftime() in X/Open for reference.

Return

19

©Copyright 1999 Rogue Wave Software, Inc.

According to the standard, they are two separate arrays. However, the Rogue Wave implementation uses the old technique involving only one array, since that part of the document is suggested only.

Return

20

For brevity, error handling is omitted in the example. If allocation fails, then badbit is set.

Return

21

This, of course, is only an example. You would probably never derive a new class for adding only one data member. However, it keeps the example simple and allows us to demonstrate the principle of deriving new stream classes.

Return

22

For a more detailed discussion of the problem and its solution, see Section 14.2, p. 306ff, of Bjarne Stroustrup, "The Design and Evolution of C++," Addison-Wesley 1994.

Return

23

In our example of a conversion between ASCII and EBCDIC, we have no reason to ever return `partial`, because this is a conversion of single byte characters. Either a character can be recognized and converted, or the conversion fails; that is, `error` is returned. The `partial` return code only makes sense in wide-character and multibyte conversions.

Return

OEM Edition, ©Copyright 1999, Rogue Wave Software, Inc.
Contact Rogue Wave about documentation or support issues.