Using the Standard Library with Borland C++

The Standard Template Library (STL) documentation describes an implementation of the STL that is consistent with the ANSI/ISO C++ working paper. To provide a completely flexible library, the working paper specifies the use of two template features that are not yet supported in the current version of Borland C++. The template features which are not yet supported are

- Member function templates
- Use of template parameters to define default types

Although the documentation includes information about STL features that are not supported, you don't need to take any special actions to start using the library. The header file for each container defines alternate forms which Borland C++ automatically inserts in your code. You must include the necessary header files as described in this topic.

Member function templates

Member function templates are used in all containers provided by the Standard Template Library. An example of this is the constructor for ***deque<T>*** that takes two templated iterators:

```
template <class InputIterator>
 deque (InputIterator, InputIterator);
```

deque also has an insert function of this type. Borland C++ does not support the use of functions that would allow you to use any type of input iterator as arguments. The header file for each container provides substitute functions that let you 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 that's in the container.

For example, to avoid 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 you cannot construct a ***deque*** this way:

```
deque<long>
 long_deque(first_deque.begin(),first_deque.end());
```

because the `long_deque` and `first_deque` are not the same type.

A container can have other member function templates besides the constructor. In general, the header file for each container provides an alternate non-template function prototype.

Template parameters

A template function can use template parameters that are initialized with a default value. The following topics describe the extent of Borland C++ support and how you should use the STL.

Default template arguments

Borland C++ supports the following form of default template arguments:

```
template < class T = int > class Array;
```

This syntax supports the construction of ***Array*** objects which, by default, are containers for **int** types. It's possible to use any type in place of **int** including other user-defined types.

Using template parameters to define default types

Borland C++ does not support functions with template parameters which are used to specify default types. Therefore, you must always supply all template arguments that would otherwise use one of their parameters to generate a default type.

For example, there is a version of the ***stack*** container that uses a template parameter to define a default type for another parameter. In the following declaration, the generic type ***T*** is used to instantiate a ***deque*** object. But ***deque*** is a generic type that depends on a generic type ***T***.

The declaration is as follows.

```
template <class T, class Container = deque<T> >
 class stack;  // This form is not supported
```

The stack.h header file provides an alternate form which is supported by Borland C++. This class declaration does not extend the scope of template parameters to define other parameters. The declaration is as follows.

```
template <class T, class Container> class stack;
```

To construct a ***stack*** type, you must always supply all arguments. You must instantiate your ***stack*** type by writing something like this:

```
stack<double, deque<double> > MyStack;
```

Using the STL header files

For the STL implementation to work correctly, you must always include files as specified in this document. For example, to use the STL ***string*** implementation, include the following in your code:

```
#include <string>
```

Similarly, to use the STL generic algorithms, include the following in your code:

```
#include <algorithm>
```

Rogue Wave

Standard C++ Library

User's Guide,
Tutorial, and
Class Reference

Rogue Wave Software
Corvallis, Oregon USA

The enclosed software, including, but not limited to, one or more of the following: source code, object code, dynamic link libraries, shared libraries, static libraries, header files, utility programs and scripts, together with the accompanying documentation (collectively known as the "Software") is owned by Rogue Wave or its suppliers and is protected by U.S. copyright laws and international treaties. It is intended for use by a C++ software programmer who has experience using C++ class libraries. The Software is <u>not</u> intended for use by consumers for domestic/household use.

Rogue Wave grants to you (one software programmer) the limited right to use only one copy of the Software on a single terminal connected to a single computer (typically one workstation) on the terms and conditions set forth in this agreement. Each Software Programmer must have his or her license to use the Software. You may (a) make one backup copy of the Software solely for backup purposes, or (b) transfer the Software to a hard disk and keep the original copy solely for backup purposes.

Subject to the restrictions contained in this agreement, you may incorporate the dynamic link libraries, statically linked libraries and shared libraries into C++ software application products that you develop. You may also modify the dynamic link libraries, statically linked libraries and shared libraries and incorporate the modified dynamic link libraries, statically linked libraries and shared libraries into C++ software application products that you develop. You may make and distribute copies of the dynamic link libraries, statically linked libraries and shared libraries of the Software as incorporated into C++ software application products that you develop provided that the Software or other Rogue Wave products do not constitute a major portion of the value of your product.

Notwithstanding any provisions in this agreement to the contrary, you may NOT (a) distribute in any manner any of the header files, source code, object modules, template based classes, or independent static libraries of the Software; or (b) distribute any portion of the Software or any derivative of any portion of the Software in a software utility product or otherwise in competition with Rogue Wave's distribution of the Software. You may not use, copy, modify, merge or compile all or any portion of the source code or object code of the Software except as expressly provided in this agreement. You may NOT (a) decompile, disassemble or reverse engineer any object code form of any portion of the Software; or (b) export from the United States any portion of the Software without obtaining the prior written consent of Rogue Wave and all applicable export licenses and governmental permits; or (c) rent or lease the Software; or (d) disclose any source codes of the Software to any person or entity; or (e) copy the documentation. The source codes of the Software are valuable assets of Rogue Wave. You agree to keep all source codes of the Software in confidence.

You may not transfer or assign the Software or your rights under this agreement.

IN NO EVENT SHALL ROGUE WAVE BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING LOSS OF PROFITS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF ROGUE WAVE WAS ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT RESTRICTED RIGHTS

The SOFTWARE and documentation are 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. Contractor/manufacturer is Rogue Wave Software, Inc., P.O. Box 2328, Corvallis, Oregon 97339.

This agreement is governed by the laws of the State of Oregon.

Rogue Wave Standard C++ Library User's Guide and Tutorial

for

Rogue Wave's implementation of the Standard C++ Library.

Based on ANSI's Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++. April 28, 1995.

User's Guide and Tutorial Author: Timothy A. Budd
Class Reference Authors: Wendi Minne, Tom Pearson, and Randy Smithey

Product Team:
Development: Anna Dahan, Donald Fowler, Mike Lijewski, Randy Smithey
Quality Engineering: Dale Oberg, Randall Robinson, Chun Zhang
Manuals: Mike Lijewski, Wendi Minne, Kristi Moore, Randy Smithey
Support: North Krimsley

Significant contributions by: Josh Gray , Rodney Mishima

Copyright © 1995 Rogue Wave Software, Inc. All rights reserved.

Introduction

1.1 What is the Standard C++ Library?
1.2 Does the Standard C++ Library differ from other libraries?
1.3 What are the effects of non-object-oriented oesign?
1.4 How should I use the Standard C++ Library?
1.5 Reading this manual
1.6 Conventions
1.7 Using the Standard Library
1.8 Running the tutorial programs

What is the Standard C++ Library?

The International Standards Organization (ISO) and the American National Standards Institute (ANSI) are completing the process of standardizing the C++ programming language. A major result of this standardization process is the ÔÔStandard C++ Library,ÕÕ a large and comprehensive collection of classes and functions. This product is *Rogue Wave*'s implementation of the ANSI/ISO Standard Library.

The ANSI/ISO Standard C++ Library includes the following parts:

- A 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 class for representing complex numbers.
- A uniform framework for describing the execution environment, through the use of a template class named ***numeric_limits*** and specializations for each fundamental data type.
- Memory management features.
- Language support features.
- Exception handling features.

This version of the *Rogue Wave Standard C++ Library* includes the data structures and algorithms libraries (STL), and the ***string***, ***complex*** and ***numeric_limits*** classes.

Does the Standard C++ Library differ from other libraries?

A major portion of the Standard C++ Library is comprised of a collection of class definitions for standard data structures and a collection of algorithms commonly used to manipulate such structures. This part of the library was formerly known as the Standard Template Library or STL. The organization and design of the STL differs in almost all respects from the design of most other C++ libraries, *because it avoids encapsulation and uses almost no inheritance.*

An emphasis on encapsulation is a key hallmark of object-oriented programming. The emphasis on combining data and functionality into an object is a powerful organization principle in software development; indeed it is *the* primary organizational technique. Through the proper use of encapsulation, even exceedingly complex software systems can be divided into manageable units and assigned to various members of a team of programmers for development.

Inheritance is a powerful technique for permitting code sharing and software reuse, but it is most applicable when two or more classes share a common set of basic features. For example, in a graphical user interface, two types of windows may inherit from a common base window class, and the individual subclasses will provide any required unique features. In another use of inheritance, object-oriented container classes may ensure common behavior and support code reuse by inheriting from a more general class, and factoring out common member functions.

The designers of the STL decided against using an entirely object-oriented approach, and separated the tasks to be performed using common data structures from the representation of the structures themselves. This is why the STL is properly viewed as a collection of algorithms and, separate from these, a collection of data structures that can be manipulated using the algorithms.

What are the effects of non-object-oriented design?

The STL portion of the Standard C++ Library was purposely designed with an architecture that is not object-oriented. This design has some side effects, some advantageous, and some not, that developers must be aware of as they investigate how to most effectively use the library. We'll discuss a few of them here.

Smaller source code
There are approximately fifty different algorithms in the STL, and about a dozen major data structures. This separation of has the effect of reducing the size of source code, and decreasing some of the risk that similar activities will have dissimilar interfaces. Were it not for this separation, for example, each of the algorithms would have to be re-implemented in each of the different data structures, requiring several hundred more member functions than are found in the present scheme.

Flexibility
One advantage of the separation of algorithms from data structures is that such algorithms can be used with conventional C++ pointers and arrays. Because C++ arrays are not objects, algorithms encapsulated within a class hierarchy seldom have this ability.

Efficiency
The STL in particular, and the Standard C++ Library in general, provide a low-level, "nuts and bolts" approach to developing C++ applications. This low-level approach can be useful when specific programs require an emphasis on efficient coding and speed of execution.

Iterators: Mismatches and invalidations
The Standard C++ Library data structures use pointer-like objects called iterators to describe the contents of a container. (These are described in detail in Section 2.) Given the library's architecture, it is not possible to verify that these iterator elements are matched; i.e., that they are derived from the same container. Using (either intentionally or by accident) a beginning iterator from one container with an ending iterator from another is a recipe for certain disaster.

It is very important to know that iterators can become invalidated as a result of a subsequent insertion or deletion from the underlying container class. This invalidation is not checked, and use of an invalid iterator can produce unexpected results.

Familiarity with the Standard C++ Library will help reduce the number of errors related to iterators.

Templates: Errors and "Code Bloat"
The flexibility and power of templatized algorithms is, with most compilers, purchased at a loss of precision in diagnostics. Errors in the parameter lists to generic algorithms will sometimes be manifest only as obscure compiler errors for internal functions that are defined many levels deep in template expansions. Again, familiarity with the algorithms and their requirements is a key to successful use of the standard library.

Because of its heavy reliance on templates, the STL can cause programs to grow larger than expected. You can minimize this problem by learning to recognize the cost of instantiating a particular template class, and by making appropriate design decisions. Be aware that as compilers become more and more fluent in templates, this will become less of a problem.

Multithreading problems
The Standard C++ Library must be used carefully in a multithreaded environment. Iterators, because they exist independently of the containers they operate on, cannot be safely passed between threads. Since iterators can be used to modify a non `const` container, there is no way to protect such a container if it spawns iterators in multiple threads. Use "thread-safe" wrappers, such as those provided by *Tools.h++,* if you need to access a container from multiple threads.

How should I use the Standard C++ Library?

Within a few years the Standard C++ Library will be the standard set of classes and libraries delivered with all ANSI-conforming C++ compilers. We have noted that the design of a large portion of the Standard C++ Library is in many ways not object-oriented. On the other hand, C++, excels as a language for manipulating objects. How do we integrate the Standard Library's non-object-oriented architecture with C++'s strengths as a language for manipulating objects?

The key is to use the right tool for each task. Object-oriented design methods and programming techniques are almost without peer as guideposts in the development of large complex software. For the large majority of programming tasks, object-oriented techniques will remain the preferred approach. And, products such as Rogue Wave's *Tools.h++ 7.0*, which will encapsulate the Standard C++ Library with a familiar object-oriented interface, will provide you with the power of the Library and the advantages of object-orientation.

Use Standard C++ Library components directly when you need flexibility and/or highly efficient code. Use the more traditional approaches to object-oriented design, such as encapsulation and inheritance, when you need to model larger problem domains, and knit all the pieces into a full solution. When you need to devise an architecture for your application, *always* consider the use of encapsulation and inheritance to compartmentalize the problem. But if you discover that you need an efficient data structure or algorithm for a compact problem, such as data stream manipulation in drivers (the kind of problem that often resolves to a single class), look to the Standard C++ Library. The Standard C++ Library excels in the creation of reusable classes, where low-level constructs are needed, while traditional OOP techniques really shine when those classes are combined to solve a larger problem.

In the future, most libraries will use the Standard C++ Library as their foundation. By using the Standard C++ Library, either directly or through an encapsulation such as *Tools.h++ 7.0*, you help insure interoperability. This is especially important in large projects that may rely on communication between several libraries. A good rule of thumb is to use the highest encapsulation level available to you, but make sure that the Standard C++ Library is available as the base for interlibrary communication and operation.

The C++ language supports a wide range of programming approaches because the problems we need to solve require that range. The language, and now the Standard C++ library that supports it, are designed to give you the power to approach each unique problem from the best possible angle. The Standard C++ Library, when combined with more traditional OOP techniques, puts a very flexible tool into the hands of anyone building a collection of C++ classes, whether those classes are intended to stand alone as a library or are tailored to a specific task.

Reading this manual

This manual is an introduction to the Rogue Wave implementation of the *Standard C++ Library*. It assumes that you are already familiar with the basics features of the C++ programming language. If you are new to C++ you may wish to examine an introductory text, such as the book *The C++ Programming Language*, by Bjarne Stroustrup (Addison-Wesley, 1991).

There is a classic “chicken-and-egg” problem associated with the container class portion of the standard library. The heart of the container class library is the definition of the containers themselves, but you can't really appreciate the utility of these structures without an understanding of 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.

Conventions

We have presented both class_names and `function_names()` in a distinctive font the first time they are introduced. In addition, when we wish to refer to a function name or algorithm name but not draw attention to the arguments, we will follow the function name with an empty pair of parenthesis. We do this even when the actual function invocation requires additional arguments. We have used the term *algorithm* to refer to the functions in the generic algorithms portion of the standard library, so as to avoid confusion with member functions, argument functions, and functions defined by the programmer. Note that both class names and function names in the standard library follow the convention of using an underline character as a separator. Throughout the text, examples and file names are printed in the same `courier font` used for function names.

In the text, it is common to omit printing the class name in the distinctive font after it has been introduced. This is intended to make the appearance of the text less visually disruptive. However, we return to the distinctive font to make a distinction between several different possibilities, as for example between the classes vector and list used as containers in constructing a stack.

Using the Standard Library

Because the Standard C++ Library consists largely of template declarations, on most platforms it is only necessary to include in your programs the appropriate header files. These header files will be noted in the text that describes how to use each algorithm or class.

Running the tutorial programs

All the tutorial programs described in this text have been gathered together and are available as part of the distribution package. You can compile and run these programs, and use them as models for your own programming problems. Many of these example programs have been extended with additional output commands that are not reproduced here in the text. The expected output from each program is also included as part of the distribution.

Iterators

2.1 Introduction to Iterators
2.2 Varieties of Iterators
2.3 Stream Iterators
2.4 Insert Iterators
2.5 Iterator Operations

Introduction to iterators

Note: *Iterators* are pointer-like objects, used to cycle through the elements stored in a container.

Fundamental to the use of the container classes and the associated algorithms provided by the standard library is the concept of an *iterator*. Abstractly, an iterator is simply a pointer-like object used to cycle through all the elements stored in a container. Because different algorithms need to traverse containers in variety of fashions, there are different forms of iterator. Each container class in the standard 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 library can be used with which container classes.

Note: A *range* is a sequence of values held in a container. The range is described by a pair of iterators, which define the beginning and end of the sequence.

Just as pointers can be used in a variety of ways in traditional programming, iterators are also 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. On the other hand, a *pair* of iterators can be used to describe a *range* of values, in a manner analogous to the way in which two pointers can be used to describe a contiguous region of memory. In the case of iterators, however, the values being described are not necessarily physically in sequence, but are rather logically 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*, that is, 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 is *not* considered to be part of the region. For example, an array named `x` of length ten 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 Ð the element that is the next value *after* the end of the range being described. Iterators are used to describe a range in the same manner. The second value is not considered to be 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 (operator `++`). When the increment operator is applied to an iterator that denotes the final value in a sequence, it will be 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`.

Note: When iterators are used to describe a range of values in a container, it is assumed (but not verified) that the second iterator is reachable from the first. Errors will occur if this is not true.

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 subsequences within a single container, by employing two iterators to specific values. Whenever two iterators are used to describe a range it is assumed, but not verified, that the second iterator is reachable from the first. Errors can occur if this expectation is not satisfied.

In the remainder of this section we will describe the different forms of iterators used by the standard library, as well as various other iterator-related functions.

Varieties of iterators

There are five basic forms of iterators used in the standard library:

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.

The following table summarizes specific ways that various categories of iterators are generated by the containers in the standard 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`

Input iterators

Input iterators are the simplest form of iterator. To understand their capabilities, consider an example program. The `find()` generic algorithm (to be described in more detail in Searching operations), 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, here called `first` and `last`. While `first` is not equal to `last` the element denoted by `first` is compared to the test value. If equal, the iterator, which now denotes the located element, is returned. If 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.

```
template <class InputIterator, class T>
InputIterator
  find (InputIterator first, InputIterator last, const T& value)
{
  while (first != last && *first != value)
     ++first;
  return first;
}
```

This algorithm illustrates three requirements for an input iterator:

- An iterator can be compared for equality to another iterator. They are equal when they point to the same position, and are otherwise not equal.
- An iterator can be dereferenced using the `*` operator, to obtain the value being denoted by the iterator.
- An iterator can be incremented, so that it refers to the next element in sequence, using the operator `++`.

Notice that these characteristics 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. It is because of this overloading that iterators are possible. There are three main varieties of input 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 deference and increment operators have their conventional meanings. For example, the following searches for the value 7 in an array of integers:

Note: Because ordinary pointers have the same functionality as random access iterators, most of the generic algorithms in the standard library can be used with conventional C++ arrays, as well as with the containers provided by the standard library.

```
int data[100];
  ...
int * where = find(data, data+100, 7);
```

Note that constant pointers, 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);
```

Container iterators

All of the iterators constructed for the various containers provided by the standard 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 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 within the class declaration with the name `iterator`. Using this, 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 library provides a mechanism to operate on an input *stream* using an input iterator. This ability is provided by the class `istream_iterator`, and will be described in more detail in Input stream iterators.

Output iterators

An output iterator has the opposite functionality from 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
  (InputIterator first, InputIterator last, OutputIterator result)
{
   while (first != last)
     *result++ = *first++;
   return result;
}
```

Note: 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.

Two ranges are being manipulated here; 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. Indeed, output iterators can use the dereference operator only in this fashion, and cannot be used to return or access the elements they denote.

As we noted earlier, ordinary pointers, as well as all the iterators constructed by containers in the standard library, can be used as examples of output iterators. (Ordinary pointers are random access iterators, which are a superset of output iterators.) So, for example, the following code fragment copies elements from an ordinary C-style array into an standard library vector:

```
int data[100];
vector<int> newdata(100);
  ...
copy (data, data+100, newdata.begin());
```

Just as the `istream_iterator` provided a way to operate on an input stream using the input iterator mechanism, the standard library provides a data type `ostream_iterator`, that permits values to be written to an output stream in an iterator-like fashion. These will be described in Output stream iterators.

Yet another form of output iterator is an *insert iterator*. 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 lists and sets.

Forward iterators

A forward iterator combines the features of an input iterator and an output iterator. It permits values to both be 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, as well as any of the iterators produced by containers in the standard library, can be used as forward iterators. The following, for example, replaces instances of the value 7 with the value 11 in a vector of integers.

```
replace (aVec.begin(), aVec.end(), 7, 11);
```

Bidirectional iterators

A bidirectional iterator is similar to a forward iterator, except that bidirectional iterators support the decrement operator (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,
           BidirectionalIterator last,
           OutputIterator result)
{
   while (first != last)
    *result++ = *--last;
   return result;
}
```

As always, the value initially denoted by the `last` argument is not considered to be 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 the ability to access 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.

When using 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 the physical addresses being described may be different.

Algorithms that use random access iterators include generic operations such as 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 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 is used in a number of the example programs presented in later sections.

The program will cycle as long as `first` is denoting 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 given by increasing index values. For a set it is the increasing order of the elements held in the container. For a list the order is explicitly derived from the fashion in which values are inserted.

A *reverse iterator* will yield values in exactly the reverse order of those given by the standard iterators. That is, for a vector or a list, a reverse iterator will generate the last element first, and the first element last. For a set it will generate the largest element first, and the smallest element last. Strictly speaking, reverse iterators are not themselves a new category of iterator. Rather, there are reverse bidirectional iterators, reverse random access iterators, and so on.

The list, set and map data types 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 data types 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.

Stream iterators

Stream iterators are used to access an existing input or output stream using iterator operations.

Input stream iterators

Note: An input stream iterator permits an input stream to be read using iterator operations. An output stream iterator similarly writes to an output stream when iterator operations are executed.

As we noted in the discussion of input iterators, the standard library provides a mechanism to turn an input stream into an input iterator. This ability is provided by the class `istream_iterator`. When declared, the two template arguments are the element type, and a type that measures the distance between elements. Almost always the latter is the standard type `ptrdiff_t`. 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 (operator `*`). The value constructed by `istream_iterator` when no arguments are provided to the constructor can be used as an ending iterator value. The following, for example, finds the first value 7 in a file of integer values.

```
istream_iterator<int, ptrdiff_t> intstream(cin), eof;
istream_iterator<int, ptrdiff_t>::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, they cannot be 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 will be written on the associated output stream, using the >> operator. To create an output stream iterator you must specify, as an argument with the constructor, the associated output stream. Values written to the output stream must recognize the stream >> operation. An optional second argument to the constructor is a string that will be used as a separator between each pair of values. The following, for example, 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> (cout, " "));
```

Simple file transformation algorithms can be created by combining input and output stream iterators and the various algorithms provided by the standard 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, ptrdiff_t> input (cin), eof;
  ostream_iterator<int> output (cout, "\n");

  remove_copy (input, eof, output, 7);
}
```

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. The following assumes that the list named `c` has 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. The following, for example, 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 lists, deques, and even vectors, but not with sets or maps.

The third, and most general form, 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 they are inserted each value 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
  copy (aList.begin(), aList.end(),
         ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Observe 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.

Iterator operations

The standard 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 iterator. 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:

```
void distance (InputIterator first, InputIterator last,
        Distance &n);
```

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.

Functions and Predicates

3.1 Functions
3.2 Predicates
3.3 Function Objects
3.4 Negators and Binders

Functions

A number of algorithms provided in the standard library require functions as arguments. A simple example is the algorithm `for_each()`, which invokes a function, passed as argument, on each value held in a container. The following, for example, 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 could perform this easily using the function `transform()` from the standard 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`.

Predicates

A *predicate* is simply a function that returns either a boolean (true/false) value or an integer value. Following the normal C convention, an integer value is assumed to be true if nonzero, and false otherwise. An example function might be the following, which takes as argument an integer and returns true if the number represents a leap year, and false otherwise:

```
bool isLeapYear (int year)
  // return true if year is leap year
{
     // millenniums are leap years
  if (0 == year % 1000) return true;
     // centuries are not
  if (0 == year % 100) return false;
     // 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);
```

Function objects

A *function object* is an instance of a class that defines the parenthesis operator as a member function. There are a number of situations where it is convenient to substitute function objects in place of functions. When a function object is used as a function, the parenthesis operator is invoked whenever the function is called.

To illustrate, 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 will be invoked. The next step is to generalize this class, by adding a constructor and a constant data field, which is set by the constructor.

```
class biggerThan {
  public:
     biggerThan (int x) : testValue(x) { }
     const int testValue;

     bool operator () (int val)
        { return val > testValue; }
};
```

The result is a general “bigger than X” 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 will find the first value in a list that is larger than 12:

```
list<int>::iterator firstBig =
  find_if (aList.begin(), aList.end(), biggerThan(12));
```

Three of the most common reasons to use function objects in place of ordinary functions are when an existing function object provided by the standard library can be employed instead of a new function, to improve execution by inlining function calls, or when the function object must either access or set state information being held by an object. We will give examples of each.

The following table illustrates the function objects provided by the standard library.

Name	Implemented operations
arithmetic functions	
`plus`	
`minus`	
`times`	
`divides`	
`modulus`	
`negate`	addition `x + y`
subtraction `x - y`	
multiplication `x * y`	
division `x / y`	
remainder `x % y`	
negation `- x`	
comparison functions	
`equal_to`	

`not_equal_to`

`greater`

`less`

`greater_equal`

`less_equal` equality test `x == y`

inequality test `x != y`

greater comparison `x > y`

less-than comparison `x < y`

greater than or equal comparison `x >= y`

less than or equal comparison `x <= y`

logical functions

`logical_and`

`logical_or`

`logical_not` logical conjunction `x && y`

logical disjunction `x || y`

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:

```
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>() );
```

Note: The class definitions for unary_function and binary_function can be incorporated by #including `functional`.

The base classes used by the standard library in the definition of the functions shown in preceding table are available for the creation of new unary and binary function objects. These base classes are defined as follows:

```
template <class ArgType, class ResultType>
class unary_function {
  typedef ArgType argument_type;
  typedef ResultType result_type;
};

template <class ArgType1, class ArgType2, class ResultType>
struct binary_function {
  typedef ArgType1 first_argument_type;
  typedef ArgType2 second_argument_type;
  typedef ResultType result_type;
};
```

An example of the use of these functions is found in Example programs. Here 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; }
};
```

A second reason to consider using function objects instead of functions is faster code. In many cases an invocation of a function object, such as the examples given in the calls on `transform()` presented earlier, can be expanded in-line, thereby eliminating the overhead of a function call.

Note: A more complex illustration of the use of a function object occurs in the radix sorting example program given as an illustration of the use of the list data type in Example program: radix sort. 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.

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:

```
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 library function `generate()` will initialize a vector of 20 elements with the values 1 through 20:

```
vector<int> aVec(20);
generate (aVec.begin(), aVec.end(), iotaGen(1));
```

Negators and binders

Negators and binders are function adaptors that are used to build new function objects out of existing function objects. Almost always, these 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 will yield the complement of the original. For example, using the widget tester function object defined in the previous section, the function object:

```
not2(WidgetTester())
```

yields a binary predicate which takes exactly the same arguments as the widget tester, and which is true when the corresponding widget tester would be false, and false otherwise. Negators work only with function objects defined as subclasses of the classes `unary_function` and `binary_function`, given earlier.

Note: The idea here described by the term binder is in other contexts often described by the term *curry*. This is not, as some people think, because it is a hot idea. Instead, 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 1930Õs. Curry himself attributed the idea to Moses Schšnfinkel, leaving one to wonder why we donÕt instead refer to binders as “Schšnfinkels.”

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 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 tie the widget number of a call on 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));
```

Container Classes

4.1 Container classes overview
4.2 Selecting a Container
4.3 Memory Management Issues
4.4 Container Types not Found in the Standard Library

Container classes overview

The standard library provides no fewer than ten alternative forms of container. In this section we will briefly describe the varieties, considering the characteristics of each, and discuss how you might go about selecting which container to use in solving a particular problem. Subsequent sections will then go over each of the different containers in more detail.

The following chart shows the ten container types provided by the standard library, and gives a short description of the most significant characteristic for each.

Name	Characteristic
`vector`	random access to elements, efficient insertions at end
`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
`map`	access to values via keys, efficient insertion and removal
`multimap`	map permitting duplicate keys
`stack`	insertions and removals only from top
`queue`	insertion at back, removal from front
`priority queue`	efficient access and removal of largest value

Selecting a container

The following series of questions can help you determine which type of container is best suited for solving a particular problem.

How are values going to be accessed?

If random access is important, than 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 in which 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. On the other hand, 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, then 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 true, then 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 will 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 preallocate 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, then a vector or deque should be employed. If, on the other hand, the key values are some other ordered data type (such as characters, strings, or a user-defined type), then the map container can be used.

Can values be related to each other?

All values stored in any container provided by the standard 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, then they cannot be stored in a set or a map.

Is finding and removing the largest value from the collection a frequent operation?

If this is true, then 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, then a deque or a list is the preferred choice. If values are inserted or removed only at the end, then a stack or queue may be a logical choice.

Is a frequent operation the merging of two or more sequences into one?

If true then a set or a list would seem to be the best choice, depending upon whether or not 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 data type

is to be preferred, since this operation is not provided in the other containers.

In many situations any number of different containers may be applicable to a given problem. In such cases one possibility is to compare actual execution timings to determine which alternative is best.

Memory management issues

Containers in the standard library can maintain a variety of different types of elements. These include the fundamental data types (`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, with little interaction by the programmer.

Values are placed into a container using the copy constructor. For most container classes, the element type held by the container must also define a default constructor. Generic algorithms that copy into a container (such as `copy()`) use the assignment operator.

When an entire container is duplicated (for example, through invoking a copy constructor or as the result of an assignment), every value is copied into the new structure using (depending on the structure) either the assignment operator or a copy constructor. Whether such a result is a "deep copy" or a "shallow copy" 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 will be invoked when values are removed from a container. When an entire collection is destroyed, the destructor will be 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 Application: event-driven simulation.

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 the memory values are properly allocated (usually by invoking the `new` operator), 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.

Container types not found in the standard library

There are a number of “classic” container types that are not found in the standard library. In most cases, the reason is that the containers that have been provided can easily be adapted to a wide variety of uses, including those traditionally solved by these alternative collections.

There is no *tree* collection that is described as such. However, the set data type is internally implemented using a form of binary search tree. For most problems that would be solved using trees, the set data type is an adequate substitute.

The set data type is specifically ordered, and there is no provision for performing set operations (union, intersection, and so on) on a collection of values that cannot be ordered (for example, a set of complex numbers). In such cases 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 in this case.

There are no *multidimensional arrays*. However, vectors can hold other vectors as elements, so such structures can be easily constructed.

There are no *graphs*. However, 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 Example program: graphs.

There are no *sparse arrays*. A novel solution to this problem is to use the graph representation discussed in Example program: graphs.

There are no *hash tables*. A hash table provides amortized constant time access, insertion and removal of elements, by converting access and removal operations into indexing operations. However, hash tables can be easily constructed as a vector (or deque) that holds lists (or even sets) as elements. A similar structure is described in the radix sort sample problem discussed in Example program: radix sort, although this example does not include invoking the hash function to convert a value into an index.

In short, while not providing every conceivable container type, the containers in the standard library represent those used in the solution of most problems, and a solid foundation from which further structures can be constructed.

Vector

5.1 The vector data abstraction
5.2 Vector operations
5.3 Boolean vectors
5.4 Example Program – Sieve of Eratosthenes

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 may be different 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 (The list data abstraction). If many insertion operations are to be performed, the list container should be used instead of the vector data type.

The vector container class in the standard library should be compared and contrasted to the deque container class we will describe in more detail in Deque data abstraction. 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, depending upon the particular set of operations being performed, use of a deque may result in a slightly faster program.

Vector include files

Whenever you use a vector, you must include the `vector` header file.

```
# include <vector>
```

Vector operations

The following chart summarizes the member functions provided by the vector data type. Each 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 Generic algorithms overview and 0

Result	Name	Arguments
	vector	()
	vector	(size)
	vector	(size, value_type)
	vector	template<class Iterator> (Iterator, Iterator)
	vector	(const vector)
	vector	template<class Iterator>assign (Iterator, Iterator)
	vector	template<class Size, class T> assign (Size, T)
reference	at	(size_type)
value_type	back	()
RandomAccessIterator	begin	()
size_type	capacity	()
bool	empty	()
RandomAccessIterator	end	()
void	erase	(iterator)
void	erase	(iterator, iterator)
value_type	front	()
void	insert	(iterator, size_type, value_type)
iterator	insert	(iterator, value_type)
void	insert	template <class Iterator> (iterator,Iterator, Iterator)
size_type	max_size	()
void	pop_back	()
void	push_back	(value_type)
RandomAccessIterator	rbegin	()
RandomAccessIterator	rend	()
void	reserve	(size_type)
void	resize	(size_type, value_type)
size_type	size	()
void	swap	(vector)

```
reference              operator[]  (size_type)
vector                 operator =  (vector)
```

Declaration and initialization of vectors

Note: Elements that are held by a vector must define a default constructor (constructor with no arguments), as well as 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 (operator ==) or the relational less-than operator (operator <).

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 (such as integer or double), a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a default constructor, as this constructor is used to initialize newly created elements. A copy constructor, either explicitly or implicitly defined, must also exist for the container element type. Like an array, a vector is most commonly declared with an integer argument that describes the number of elements the vector will hold:

```
vector<int> vec_one(10);
```

The constructor used to create the vector in this situation 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 vectors. In addition to a size, the constructor can provide a constant value that will be used to initialize each new vector location. 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 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. In the mean time, while compiler technology catches up with the standard library definition, the Rogue Wave version of the standard 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 subsequence of an existing container. The values from this subsequence then become the new elements in the receiver. The second version of `assign()` takes a count and an optional value of the container element type. After the call the container will hold 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 will be called for each value removed from the collection.

Finally, two vectors can exchange their entire contents by means of the `swap()` operation. The argument container will take on the values of the receiver, while the receiver will assume those of the argument. A swap is very efficient, and should be used, where appropriate, in preference to an explicit

element-by-element transfer.

```
vec_three.swap(vec_four);
```

Type definitions

The class vector includes a number of type definitions. These are 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`, the following types are defined:

`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 to distances between iterators.

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. And, like arrays, there currently are no attempts to verify the validity of the index values (although this may change in future releases). Indexing a constant vector yields a constant reference. Attempts to index a vector outside the range of legal values will generate 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.

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 vectors.

```
cout << vec_five.front() << " ... " << vec_five.back() << endl;
```

Extent and size-changing operations

There are, in general, three different “sizes” associated with any vector. The first is the number of elements currently being held by the vector. The second is the maximum size to which the vector can be expanded without requiring that new storage be allocated. The third is 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 data type `size_type`. The current size and capacity are more difficult to characterize. As we will note in the next section, 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.

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 are possible with a vector.

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 data type 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, then a reallocation occurs and the argument value becomes the new capacity. (It may subsequently grow even larger; the value given as argument need not be a bound, just a guess.) If the capacity is already in excess of the argument, then 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. If a destructor is defined for the element type, the destructor will be called for any values that are removed from the collection.

```
       // become size 12, adding values of 17 if necessary
vec_five.resize (12, 17);
```

Inserting and removing elements

As we noted earlier, the class vector differs from an ordinary array in that a vector can, in certain circumstances, increase or decrease in size. 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, then 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, then 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
```

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 will be 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. These functions are described in Deque data abstraction, which discusses deques in more detail.)

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 subsequence from another container. The range of values described by the sequence is inserted into the vector. Again, because at most a single allocation is performed, using this function is preferable to using a sequence of individual insertions.

```
vec_five.insert (where, vec_three.begin(), vec_three.end());
```

Note: Once more, it is important to remember that should reallocation occur as a result of an insertion, all references, pointers, and iterators that denoted a location in the now-deleted memory block that held the values before reallocation 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()`. There are two forms; the first takes a single iterator and removes an individual value, while 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 will be 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, however 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.

Vector 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`() (Find an element satisfying a condition and Count the number of elements that satisfy a condition) can be used for this purpose. The following statement, for example, tests to see whether an integer vector contains the element 17.

Note: Note that 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.

```
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;
```

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()` (Sorting algorithms). 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 Ordered collection algorithms overview can be applied to a vector holding an ordered collection. For example, two vectors can be merged using the generic algorithm `merge()` (Merge ordered sequences).

```
  // merge two vectors, printing output
merge (vecOne.begin(), vecOne.end(), vecTwo.begin(), vecTwo.end(),
  ostream_iterator<int> (cout, " "));
```

Sorting a vector also lets us use the more efficient binary search algorithms (Binary search), instead of a linear traversal algorithm such as `find()`.

Useful generic algorithms

Most of the algorithms described in Generic algorithms overview can be used with vectors. The following table summarizes a few of the more useful of these. 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;
```

Purpose	Name
Fill a vector with a given initial value	`fill`
Copy one sequence into another	`copy`
Copy values from a generator into a vector	`generate`
Find an element that matches a condition	`find`
Find consecutive duplicate elements	`adjacent_find`
Find a subsequence within a vector	`search`
Locate maximum or minimum element	`max_element`, `min_element`
Reverse order of elements	`reverse`
Replace elements with new values	`replace`
Rotate elements around a midpoint	`rotate`
Partition elements into two groups	`partition`
Generate permutations	`next_permutation`
Inplace merge within a vector	`Inplace_merge`
Randomly shuffle elements in vector	`random_shuffle`
Count number of elements that satisfy condition	`count`
Reduce vector to a single value	`accumulate`
Inner product of two vectors	`inner_product`
Test two vectors for pairwise equality	`equal`
Lexical comparison	`lexicographical_compare`
Apply transformation to a vector	`transform`
Partial sums of values `partial_sum`	
Adjacent differences of value	`adjacent_difference`
Execute function on each element	`for_each`

Boolean vectors

Vectors of bit values (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.

One new member function added to the boolean vector data type is `flip()`. When invoked, this function inverts all the bits of the vector. Boolean vectors 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
```

vector<bool> also supports an additional `swap()` member function.

Example program: sieve of Eratosthenes

Note: Source for this program is found in the file `sieve.cpp`.

An example program that illustrates the use of vectors is the classic algorithm, called the *sieve of Eratosthenes*, used to discover prime numbers. 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 those values that cannot be primes; thus all the remaining values will be the prime numbers. To do this, a loop examines each value in turn, and for those that are set to one (and thus have not yet been excluded from the set of candidate primes) strikes out all multiples of the number. When the outermost loop is finished, all remaining prime values have been discovered. The program is as follows:

```
void main() {
    // create a sieve of integers, initially set
  const int sievesize = 100;
  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])
      cout << j << " ";
  cout << endl;
}
```

List

6.1 The List Data Abstraction
6.2 List Operations
6.3 Example Programs

The list data abstraction

The vector data structure is a container of relatively fixed size. While the standard 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. In such cases an alternative data structure should be employed. In this section we will examine an alternative data structure that can be used in these circumstances, the list data type.

A list corresponds to the intuitive idea of holding elements in a linear (although not necessarily ordered) sequence. New values can be added or removed either to or from the front of the list, or to or from the back. By using an iterator to denote a position, elements can also be added or removed to or 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. Finally, a list is a linear structure. The contents of the list cannot be accessed by subscript, and, in general, elements can only be accessed by a linear traversal of all values.

List include files

Whenever you use a list, you must include the `list` header file.

```
# include <list>
```

List operations

The following chart summarizes the member functions provided by the list data type. Each 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 Generic algorithms overview and Ordered collection algorithms overview.

Result	Name	Arguments
	list	()
	list	(size)
	list	(size, value_type)
	list	template <class Iterator> (Iterator, Iterator)
	list	(const list)
	list	template<class Iterator> assign (Iterator, Iterator)
	list	template<class Size,class T> assign (Size, T)
value_type	back	()
BidirectionalIterator	begin	()
bool	empty	()
BidirectionalIterator	end	()
void	erase	(iterator)
void	erase	(iterator, iterator)
value_type	front	()
iterator	insert	(iterator, size_type, value_type)
iterator	insert	(iterator, value_type)
void	insert	template <class Iterator> (iterator, Iterator, Iterator)
size_type	max_size	()
void	merge	(list)
void	pop_back	()
void	pop_front	()
void	push_back	(value_type)
void	push_front	(value_type)
BidirectionalIterator	rbegin	()

void	remove	(value_type)
void	remove_if	(predicate)
BidirectionalIterator	rend	()
void	reverse	()
size_type	size	()
void	sort	()
void	splice	(iterator, list)
void	splice	(iterator, list, iterator)
void	splice	(iterator, list, iterator iterator)
void	swap	(list)
void	unique	()
void	unique	(predicate)
list	operator =	(list)

In subsequent sections we will illustrate the basic operations that can be performed with lists.

Declaration and initialization of lists

Note: Note that 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, for example, automatically free memory for these objects when an item is erased from the container.

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 `integer` or `double`), a pointer type, or a user-defined type. In the latter case, the user-defined type *must* implement a default constructor (a constructor with no arguments), as this constructor is in some cases used to initialize newly created elements. A collection declared in this fashion will initially not contain any elements.

```
list <int> list_one;
list <Widget *> list_two;
list <Widget> list_three;
```

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 lists. The constructor for this form takes two arguments, a size and an initial value. The second argument is optional. If only the number of initial elements to be created is given, these values will be initialized with the default constructor; otherwise the elements will be 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)
```

Lists can also be initialized using elements from another collection, using 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 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 (see Insert iterators). 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, to be described in Placing elements into a list, can also be used to place values denoted by an iterator into a list. Insert iterators can be used to initialize a list with a sequence of values produced by a *generator* (see Initialize a sequence with generated values). 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.

```
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 will be invoked for the elements being removed. There are two forms of `assign()`. The first takes two iterator arguments that specify a subsequence of an existing container. The values from this subsequence then become the new elements in the receiver. The second version of assign takes a count and an optional value of the container element type. After the call the container will hold the number of elements specified by the count, which will be equal to either the default value for the container type or the initial value specified.

```
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 lists can exchange their entire contents by means of the operation `swap()`. The argument container will take on the values of the receiver, while the receiver will assume those of the argument. A swap is very efficient, and should be used, where appropriate, in preference to an explicit element-by-element transfer.

```
list_ten.swap(list_nine);     // exchange lists nine and ten
```

Type definitions

The class list includes a number of type definitions. The most common use for these is in declaration statements. For example, an iterator for a list of integers can be declared in the following fashion:

```
list<int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`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 to distances between iterators.

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 a previous discussion (Insert iterators) we noted how, with the aid of an insert iterator and the `copy()` or `generate()` generic algorithm, values can be placed into a list at a location denoted by an iterator. 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.

```
         // insert default widget 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));
```

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.

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 Searching operations, such as an invocation of the `find()` generic algorithm. 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.

```
        //  find the location of the first 5 value 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);
```

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 twelveÕs
```

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());
```

There are a variety of ways to *splice* one list into another list. 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 subsequence 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 lists 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 lists. As with the generic algorithm of the same name (Merge ordered sequences), two forms are supported. The second form uses the binary function supplied as argument to order values. 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

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 will be 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 subsequence, 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 5 and the following 7
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 the use of either of these is to use the `remove()` or `remove_if()` generic algorithms (Remove unwanted elements). 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()` will erase 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 were 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 (see Remove runs of similar values). In the following example the binary function is the greater-than operator, which will have the effect of removing 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 operations

The member function `size()` will return the number of elements being held by a container. The function `empty()` will return 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.

```
         // 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) or through the use of iterators.

There are two 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 data type `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.

List test for inclusion

The list data types do not directly provide any method that can be used to determine if a specific value is contained in the collection. However, either the generic algorithms `find()` or `count()` (Find an element satisfying a condition and Count the number of elements that satisfy a condition) can be used for this purpose. The following statements, for example, tests to see whether an integer list contains the element 17.

```
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 lists. These are described in detail in Ordered collection algorithms overview.

Searching operations

The various forms of searching functions described in Searching operations, 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 library will 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 lists 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 in Generic algorithms overview.

For a list, the member function `reverse()` reverses the order of elements in the list.

```
  list_ten.reverse();   // elements are now reversed
```

The generic algorithm `transform()` (Transform one or two sequences) can be used to modify every value in a container, by simply using the same container as both input and as result for the operation. The following, for example, increments each element of a list by one. 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 in a similar fashion.

```
transform(list_ten.begin(), list_ten.end(),
    list_ten.begin(), bind1st(plus<int>(), 1));
```

In an analogous manner, the functions `replace()` and `replace_if()` (Replace certain elements with fixed value) can be used to replace elements of a list with specific values. Rotations (Rotate elements around a midpoint) and partitions (Partition a sequence into two groups), can also be performed with lists.

```
  // find the location of the 5 value, 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()` (Generate permutations in sequence) can be used to generate the next permutation (or previous permutation) of a collection of values.

```
next_permutation (list_ten.begin(), list_ten.end());
```

Other operations

The algorithm for_each() (Apply a function to all elements in a collection) will apply a function to every element of a collection. An illustration of this use will be given in the radix sort example program in the section on the deque data structure.

The accumulate() generic algorithm reduces a collection to a scalar value (see Reduce sequence to a single value). This can be used, for example, to compute the sum of a list of numbers. A more unusual use of accumulate() will be illustrated in the radix sort example.

```
cout << "Sum of list is: " <<
    accumulate(list_ten.begin(), list_ten.end(), 0) << endl;
```

Two lists 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 (see Lexical comparison).

Example program: an inventory system

Note: The executable version of the widget works program is contained in file `widwork.cpp` on the distribution disk.

We will 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. 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; }
  bool operator == (const Widget& rhs)
     { return lhs.id == rhs.id; }
  bool operator < (const Widget& rhs)
     { return lhs.id < rhs.id; }
};
```

The state of the inventory is represented by two lists. One list represents the stock of widgets on hand, while the second represents the type of widgets that customers have backordered. The first is a list of widgets, while the second is a list of widget identification types. To handle our inventory we have two commands; the first, `order()` processes orders, while the second, `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 can do this 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
     { 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);
     }
}
```

Deque

7.1 The Deque Data Abstraction
7.2 Deque Operations
7.3 An Example Program -- Radix Sort

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, as well as 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. (A capability not provided by the list class).
- Like a list, values can be efficiently added either to the front or to the back of a deque. (A capability 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 both in situations that require a vector and in those that call for a list. Often, the use of a deque in place of either a vector or a list will result in faster programs. To determine which data structure should be used, you can refer to the set of questions described in Selecting a container.

Deque include files

The `deque` header file must appear in all programs that use the deque data type.

```
# include <deque>
```

Deque operations

The following table summarizes the member functions provided by the deque data type. You will note the close similarity between this chart and the ones provided earlier for the vector and list data types. No further discussion will be provided for those operations which match either the vector or list member functions discussed earlier.

Result	Name	Arguments
	deque	()
	deque	(size_type)
	deque	(size_type, value_type)
	deque	template <class Iterator> (Iterator, Iterator)
	deque	template <class Iterator> assign (Iterator, Iterator)
	deque	template <class Size,class T> assign (Size, T)
	deque	(const deque)
reference	at	(size_type)
reference	back	()
RandomAccessIterator	begin	()
bool	empty	()
RandomAccessIterator	end	()
void	erase	(iterator)
void	erase	(iterator,iterator)
reference	front	()
iterator	insert	(iterator, value_type)
void	insert	(iterator, size_type, value_type)
void	insert	template <class Iterator> (iterator, Iterator, Iterator)
size_type	max_size	()
void	pop_back	()
void	pop_front	()
void	push_back	(value_type)
void	push_front	(value_type)
RandomAccessIterator	rbegin	()
RandomAccessIterator	rend	()
void	resize	(size_type,

		value_type)
size_type	size	()
void	swap	(deque)
reference	operator []	size_type
deque	operator =	deque

A deque is declared in the same fashion as a vector, and includes within the class the same type definitions as vector.

Notice that the `begin()` and `end()` member functions return random access iterator, rather than bidirectional iterators, as they do for lists.

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 data type, this is a much more restrictive condition than insertions into a list.

If the underlying element type provides a destructor, then the destructor will be invoked when a value is erased from a deque.

Since the deque data type provides random access iterators, all the generic algorithms that operate with vectors can also be used with deques.

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 will permit a deque to hold many more elements than a vector.

As values are inserted, the index associated with any particular element in the collection will change. For example, if a value is inserted into position 3, then the value formerly indexed by 3 will now be found at index location 4, the value formerly at 4 will be found at index location 5, and so on.

Example program: radix sort

The radix sort algorithm is a good illustration of how lists and deques can be combined with other containers. In the case of radix sort, a vector of deques is manipulated, much like a hash table.

Note: The complete radix sort program is found in the file `radix.cpp` in the tutorial distribution disk.

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 have been examined, the list must be sorted.

The following table 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 one's place digits are ordered. During pass 2 the ten's place digits are ordered, retaining the relative positions of values set by the earlier pass. On pass 3 the hundred's place digits are ordered, again retaining the previous relative ordering. After three passes the result is an ordered list.

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 deques 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:

```
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 would be 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);
       }
};
```

Set and Multiset

8.1 The Set Data Abstraction
8.2 Set and Multiset Operations
8.3 Example Program Ð A Spelling Checker
8.4 The class_bit set

The set data abstraction

Note: Although the abstract concept of a set does not necessarily imply an ordered collection, the set data type is always ordered. If necessary, a collection of values that cannot be ordered can be maintained in, for example, a list.

A *set* is a collection of values. Because the container used to implement the set data structure maintains values in an ordered representation, sets 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, sets 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 sets provided by the standard library. In the set container, every element is unique. 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.

Set include files

Whenever you use a set or a multiset, you must include the set header file.

```
# include <set>
```

Set and multiset operations

Note: In other programming languages, a multiset is sometimes referred to as a *bag*.

The following chart summarizes the member functions provided by the set and multiset data types. Each will shortly be described in more detail. 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 Generic algorithms overview and Ordered collection algorithms overview.

Result	Name	Arguments
	set	()
	multiset	()
	set	(Compare)
	multiset	(Compare)
	set	template <class Iterator> (Iterator, Iterator)
	multiset	template <class Iterator> (Iterator, Iterator)
	set	(const set)
	multiset	(const multiset)
BidirectionalIterator	begin	()
size_type	count	(value_type)
bool	empty	()
BidirectionalIterator	end	()
pair<iterator, iterator>	equal_range	(value_type)
void	erase	(iterator)
size_type	erase	(value_type)
void	erase	(iterator, iterator)
iterator	find	(value_type)
pair<iterator, bool>	insert	(value_type)
iterator	insert	(iterator, value_type)
void	insert	template <class Iterator> (Iterator, Iterator)
iterator	lower_bound	(value_type)
size_type	max_size	()
BidirectionalIterator	rbegin	()
BidirectionalIterator	rend	()
size_type	size	()
void	swap	(set)
iterator	upper_bound	(key_type)
Function	value_comp	()

```
set                    operator =    (set)
```

Creation and initialization

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 it is not provided, the less than operator for the key type will be 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 (operator ==) and the less than comparison operator (operator <).

Note: As we noted in the earlier discussion on vectors and lists, 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 then using the `copy()` generic algorithm to copy values into the set.

Sets can be declared with no initial elements, or they can be initialized from another container by providing a pair of iterators. An optional argument in both cases 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. 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> set_three(greater<int>());

set <gadget, less<gadget>() > gset;
set <gadget> gset(less<gadget>())

set <int> set_four (aList.begin(), aList.end());
set <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 library containers).

```
set_one = set_five;
set_six.swap(set_two);
```

Type definitions

The classes set and multiset include a number of type definitions. The most common use for these is in a declaration statement. For example, an iterator for a set of integers can be declared in the following fashion:

```
set<int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`value_type`	The type associated with the elements the set 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 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.

Insertion

Note: See the discussion of maps in The map data abstraction for a description of the pair data type.

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. 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());
```

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
stesttype::iterator five = set_three.find(5);
set_three.erase(five);

  // erase all values between seven and eleven
stesttype::iterator seven = set_three.find(7);
stesttype::iterator eleven = set_three.find(11);
set_three.erase (seven, eleven);
```

If the underlying element type provides a destructor, then the destructor will be invoked prior to removing the element from the collection.

Searching and counting

The member function `size()` will yield the number of elements held by a container. The member function `empty()` will return 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 (the 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.

```
list<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 multisets, as with sets 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

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.

The member functions `begin()` and `end()` produce iterators for both sets and multisets. The iterators produced by these functions are constant to insure that the ordering relation for the set is not inadvertently or intentionally destroyed by assigning a new value to a set element. Elements are generated by the iterators 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.

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 will work with any ordered structure. These functions are described in more detail in Set operations. The following summary describes 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. In the case of multisets 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 is a subset" << endl;
```

The less than operator (operator <) will be 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 sets.

Set union or intersection

The function `set_union()` can be used to construct a union of two sets. The two sets 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. (See Insert iterators for a discussion of insert iterators.) 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 sets.

As with the `includes()` function, the less than operator (operator <) is used to compare elements in the two argument sets, regardless of the operator provided in the declaration of the sets. 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 multisets should be distinguished from the operation of merging two sets. 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 (see Merge ordered sequences). 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 sets are ordered and have constant iterators, a number of the generic functions described in Generic algorithms overview and Ordered collection algorithms overview either are not applicable to sets or are not particularly useful. However, the following table gives a few of the functions that can be usefully used in conjunction with the set data type.

Purpose	Name
Copy one sequence into another Copy one sequence onto another sequence	`copy`
Find an element that matches a condition Find an element satisfying a condition	`find_if`
Find a subsequence within a set Find a subsequence within a sequence	`search`
Count number of elements that satisfy condition Count the number of elements that satisfy a condition	`count_if`
Reduce set to a single value Reduce sequence to a single value	`accumulate`
Execute function on each element Apply a function to all elements in a collection	`for_each`

Example program: a spelling checker

Note: This program can be found in the file `spell.cpp` in the tutorial distribution.

A simple example program that uses a set is a spelling checker. The checker takes as arguments two input streams; the first representing a stream of correctly spelled words (that is, a dictionary), and the second a text file. First, the dictionary is read into a set. This is performed using a `copy()` 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, then 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 will 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.

```
void findMisspell(stringset & words, string & word)
{
  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]);
    }
}
```

The class bit_set

A ***bit_set*** 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 ***bit_set*** does not provide any iterators for accessing elements.

Initialization and creation

A ***bit_set*** is a template class abstraction. The template argument is not, however, a type, but an integer value. The value represents the number of bits the set will contains.

```
bit_set<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 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 will be generated. The actual size, however, will be determined by the constructor.

```
bit_set<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 bit_set is created that has as many elements as are characters in the string, and is initialized with the values from the string.

```
bit_set<126> small_set("10101010"); // this set has 8 elements
```

Accessing and testing elements

An individual bit in the bit_set can be accessing using the subscript operation. Whether the bit is one or not can be determined using the member function `test()`. Whether any bit in the bit_set is ÔÔonÕÕ 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;
```

The function `set()` can be used to set a specific bit. `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 (sets 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 bit_set, while the member function `count()` yields the number of bits that are set.

Set operations

Set operations on bit_sets are implemented using the bit-wise operators, in a manner analogous to the way in which the same operators act on integer arguments.

The negation operator (operator `~`) applied to a bit_set returns a new bit_set containing the inverse of elements in the argument set.

The intersection of two bit_sets is formed using the `and` operator (operator `&`). The assignment form of the operator can 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 (operator `|`). The exclusive-or is formed using the bit-wise exclusive or operator (operator `^`).

The left and right shift operators (operator `<<` and `>>`) can be used to shift a bit_set left or right, in a manner analogous to the use of these operators 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 ***bit_set*** into an `unsigned long`. It is an error to perform this operation on a bit_set containing more elements than will fit into this representation.

The member function `to_string()` converts a ***bit_set*** into an object of type ***string.*** The string will have as many characters as the bit_set. Each zero bit will correspond to the character `0`, while each one bit will be represented by the character `1`.

Map and Multimap

9.1 The Map Data Abstraction
9.2 Map and Multimap Operations
9.3 Example Programs

The map data abstraction

Note: In other programming languages, a map-like data structure is sometimes referred to as a *dictionary*, a *table*, or an *associative array*.

A map is an indexed data structure, similar to a vector or a deque. However, maps differ from vectors or deques in two important respects. First, in a map, unlike a vector or deque, the index values (called the *key values*) need not be integer, but can be any ordered data type. For example, maps can be indexed by real numbers, or by strings. Any data type 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 use of the subscript operator (although there are other techniques). The second important difference is that a map is an ordered data structure. This means that elements are maintained in sequence, the ordering being determined by key values. Because they maintain values in order, maps 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.

There are two varieties of maps provided by the standard library. The map data structure demands unique keys. That is, there is a one-to-one association between key elements and their corresponding value. In a map, the insertion of a new value that uses an existing key is ignored. A multimap, on the other hand, permits multiple different entries to be indexed by the same key. Both data structures provide relatively fast (logarithmic time) insertion, deletion, and access operations.

Note: If you want to use the pair data type without using maps, you should include the header file named `utility`.

In large part, a map can simply be considered to be a ***set*** that maintains a collection of pairs. 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.

Map include files

Whenever you use a map or a multimap, you must include the `map` or `multimap` header file.

```
# include <map>
```

Map and multimap operations

The following chart summarizes the member functions provided by the map and multimap data types. Each 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 Generic algorithms overview and Ordered collection algorithms overview.

Result	Name	Arguments
	map	()
	multimap	()
	map	(Compare)
	multimap	(Compare)
	map	template <class Iterator> (Iterator, Iterator)
	multimap	template <class Iterator> (Iterator, Iterator)
	map	(const map)
	multimap	(const multimap)
BidirectionalIterator	begin	()
size_type	count	(key_type)
bool	empty	()
BidirectionalIterator	end	()
pair<iterator, iterator>	equal_range	(key_type)
void	erase	(iterator)
size_type	erase	(key_type)
void	erase	(iterator, iterator)
iterator	find	(key_type)
Function	key_comp	()
pair<iterator, bool>	insert	(value_type)
iterator	insert	(iterator, value_type)
void	insert	template <class Iterator> (Iterator, Iterator)
iterator	lower_bound	(key_type)
size_type	max_size	()
BidirectionalIterator	rbegin	()
BidirectionalIterator	rend	()
size_type	size	()

`void`	`swap`	`(map)`
`iterator`	`upper_bound`	`(key_type)`
`Function`	`value_comp`	`()`
`reference`	`operator []`	`key_type` (map only)
`map`	`operator =`	`(map)`

Creation and initialization

The declaration of a map follows the pattern we have seen repeatedly in the standard 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. If your compiler supports default template types (a relatively new feature in C++ not yet supported by all vendors), then the last of these is optional, and if not provided, the less than operator for key type will be assumed. Maps 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 maps to be created as copies of other maps.

```
  // 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 maps can exchange their values using the `swap()` operation (in a manner analogous to other standard library containers).

Type definitions

The classes map and multimap include a number of type definitions. These are most commonly used in declaration statements. For example, an iterator for a map of strings to integers can be declared in the following fashion:

```
map<string, int>::iterator location;
```

In addition to `iterator`, the following types are defined:

`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.
`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 to the distances between iterators.

Insertion and access

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 data type 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 Ð 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; as, for example, the iterator yielded by a `find()` operation. A pair of iterators can be used to erase an entire range of elements.

```
  // erase element 4
map_three.erase(4);
  // erase element five
mtesttype::iterator five = map_three.find(5);
map_three.erase(five);

  // erase all values between seven and eleven
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, then the destructor will be invoked prior to removing the key and value pair from the collection.

Iterators

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.

The member functions `begin()` and `end()` produce bidirectional iterators for both maps and multimaps. Dereferencing a iterator for either a map or a multimap will yield a pair of key/value elements. The fields 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 will be 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.

Searching and counting

The member function `size()` will yield the number of elements held by a container. The member function `empty()` will return 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. In the case of multimaps, 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_end.end())
  cout << "contains a four" << 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 will be presented later in this section.

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 four" << endl;
```

Element comparisons

The member functions `key_comp()` and `value_comp()`, which take no arguments, return a function 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 will have any effect on the container.

```
if (map_two.key_comp()(i, j))
  cout << "element i is less than j" << endl;
```

Other map operations

Because maps and multimaps are ordered collections, and because the iterators for maps return pairs, many of the functions described Generic algorithms overview and Ordered collection algorithms overview are meaningless or difficult to use. However, there are a few notable exceptions. The functions `for_each()` (Apply a function to all elements in a collection), `adjacent_find()` (Find consecutive duplicate elements), and `accumulate()` (Reduce sequence to a single value) 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.

Example programs

We present three example programs that illustrate the use of maps and multimaps. These are a telephone database, graphs, and a concordance.

Example program: a telephone database

Note: The complete example program is included in file `tele.cpp` in the distribution.

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 will use a pair of slightly more complex operations to illustrate how a few of the algorithms described in Generic algorithms overview can be used with maps. Suppose one wanted to display all the phone numbers with a certain three digit initial prefixapologize to international readers for this obviously North-American-centric example. . We will use the `find_if()` function (which is different from the `find()` member function in class map) to locate the first entry. Starting from this location, subsequent calls on `find_if()` will 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;
}
```

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 technique that is commonly used with the standard library, defining the predicate function as an instance of a class, and storing the test predicate as an instance variable in the class, initialized when the class is constructed. The desired function is then defined as the function call operator for the class:

```
int prefix(const pair<string, long> entry)
  { return entry.second / 10000; }

class checkPrefix {
public:
  checkPrefix (int p) : testPrefix(p) { }
  const int testPrefix;
  bool operator () const (const entry_type & entry)
     { return prefix(entry) == testPrefix; }
};
```

Our final example will be to display the directory sorted by prefix. It is not possible to alter the way in which maps are themselves ordered. So instead, we create a new map with the element types reversed, then copy the values into the new map, which will have the effect of ordering the values by prefix. Once the new map is created, it is then 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);
}
```

The function used to print the sorted entries is the following:

```
void printSortedEntry (const sorted_entry_type & entry)
     { cout << entry.first << ":" << entry.second << endl; }
```

Example program: graphs

Note: The executable version of this program is found in the file `graph.cpp` on the distribution disk.

A map who's elements are themselves maps 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 in the following fashion:

```
typedef map<string, int> stringVector;
typedef map<string, stringVector> graph;

string pendleton("Pendleton");      // define strings for city names
string pensacola("Pensacola");
string peoria("Peoria");
string phoenix("Phoenix");
string pierre("Pierre");
string pittsburgh("Pittsburgh");
string princeton("Princeton");
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;
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 pairs is then constructed, and initialized with the distance from the starting city to itself (namely, zero). The definition for the distance pair data type is as follows:

```
struct DistancePair {
  unsigned first;
  string second;
  DistancePair() : first(0) { }
  DistancePair(unsigned int f, const string & s)
     : first(f), second(s) { }
bool operator < (const DistancePair & rhs) const
  { 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 by examining the graph we can compute the distance from this city to each of its adjacent cities. 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<DinstancePair> > 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(DinstancePair(distance + (*start).second,
             (*start).first));
       }
    }
}
```

Notice that this relatively simple algorithm makes use of vectors, maps, strings and priority queues.

Example program: a concordance

Note: An executable version of the concordance program is found on the distribution file under the name `concord.cpp`.

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 will be maintained in the concordance by a multimap, indexed by strings (the words) and will hold integers (the line numbers). A multimap is employed because the same word will often appear on multiple different lines; indeed, discovering such connections is one of the primary purposes of a concordance. An alternative possibility would have been to use a map and use a set of integer elements as the associated values.

```
class concordance {
  typedef multimap<string, int> wordDictType;
public:
  void addWord (string, int);
  void readText (istream &);
  void printConcordance (ostream &);

private:
  wordDictType wordMap;
};
```

The creation of the concordance is divided into two steps: first the program generates the concordance (by reading lines from an input stream), and then the program 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, Ô\nÕ); 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 Example function: split a line into words. 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 values are not duplicated in the word map should the same word occur 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. If, for example, 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:

```
best: 1
it: 1 2
of: 1 2
the: 1 2
times: 1 2
was: 1 2
worst: 1
```

Stack and Queue

10.1 Stack and queue overview
10.2 The Stack Data Abstraction
10.3 The Queue Data Abstraction

Stack and queue 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 it is the item on the top that 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.

Note: A stack is sometimes 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.

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.

In the standard library, both stacks and queues are *adaptors*, built on top of other containers which are used to actually hold the values. A stack can be built out of either a vector 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 Generic algorithms overview and Ordered collection algorithms overview cannot be used with either data structure.

The stack data abstraction

As a data abstraction, a stack is traditionally defined as any object that implements the following operations:

`empty()`	return true if the collection is empty
`size()`	return number of elements in collection
`top()`	return (but do not remove) the topmost element in the stack
`push(newElement)`	push a new element onto the stack
`pop()`	remove (but do not return) the topmost element from the stack

Note that accessing the front element and removing the front element are separate operations. Programs that utilize the stack data abstraction should include the file `stack`, as well as the include file for the container type (e.g., `vector`).

```
# include <stack>
# include <vector>
```

Note: Note that on most compilers it is important to leave a space between the two right angle brackets in the declaration of a stack; otherwise they are interpreted by the compiler as a right shift operator.

A declaration for a stack must specify two arguments; the underlying element type, and the container that will hold the elements. For a stack, the most common container is a vector or a deque, however a list 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, vector<int> > stackOne;
stack< double, deque<double> > stackTwo;
stack< Part *, list<Part * > > stackThree;
stack< Customer, list<Customer> > stackFour;
```

The last example creates a stack of a programmer-defined type named Customer.

Example program: a RPN calculator

A classic application of a stack is in the implementation of calculator. Input to the calculator consists of a text string that represents an expression written in reverse polish notation (RPN). Operands, that is, 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.

Note: This program is found in the file `calc.cpp` in the distribution package.

We can divide the development of our stack simulation into two parts. 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 Ð 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 simplifying assumptions: all operands will be integer values, and we will handle only binary operators.

```
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, vector<int> > data;
};
```

Note: A more robust program would check to see if the stack was empty before attempting to perform the pop() operation.

The member function `doOperator()` performs the actual work. It pops values from the stack, 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
    }
}
```

The queue data abstraction

As a data abstraction, a ***queue*** is traditionally defined as any object that implements the following operations:

`empty()`	return true if the collection is empty
`size()`	return number of elements in collection
`front()`	return (but do not remove) the element at the front of the queue
`back()`	return the element at the end of the queue
`push(newElement)`	push a new element on to the end of the queue
`pop()`	remove (but do 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. Programs that utilize the queue data abstraction should include the file `queue`, as well as the include file for the container type (e.g., `list`).

```
# include <queue>
# include <list>
```

A declaration for a queue must specify both the element type as well as the container that will hold the values. For a queue the most common containers are a list or a deque. 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, deque<double> > queueTwo;
queue< Part *, list<Part * > > queueThree;
queue< Customer, list<Customer> > queueFour;
```

The last example creates a queue of a programmer-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 Generic algorithms overview or Ordered collection algorithms overview apply to queues.

Example program: bank teller simulation

Note: The complete version of the bank teller simulation program is found in file `teller.cpp` on the distribution disk.

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 wish 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 will spend at the counter. The latter is a value randomly selected between 2 and 8. (See Random access iterators for a discussion of the `randomInteger()` function.)

```
class Customer {
public:
  Customer (int at = 0) : arrival_Time(at),
       processTime(2 + randomInteger(6)) {}
  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 library containers if they can be compared for equality and ordering, it is necessary to define the < and == operators 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 is then a large loop, cycling once each simulated minute. Each minute a new

customer is, with probability 0.9, 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, deque<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;
}
```

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 amount.

Priority_Queue

11.1 The Priority Queue Data Abstraction
11.2 Priority Queue Operations
11.3 Application Ð Event Driven Simulation

The priority queue data abstraction

A *priority queue* is a data structure useful in problems where it is important to be able 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.

Note: The term priority *queue* is a misnomer, in that the data structure is not a queue, in the sense that we used the term in Stack and queue overview, since it does not return elements in a strict first-in, first-out sequence. Nevertheless, the name is now firmly associated with this particular data type.

A more computer-related example of a priority queue is that used by an operating system to maintain a list of pending processes, 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 will be 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, or will, encounter others.

The priority queue operations

A priority queue is a data structure that can hold elements of type `T` and that implements the following five operations:

`push(T)`	add a new value to the collection being maintained
`top()`	return a reference to the smallest element in collection
`pop()`	delete the smallest element from the collection
`size()`	return the number of elements in the collection
`empty()`	return 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 (the < 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 section. As with all the containers in the Standard Library, there are two 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.

Note: As we noted in earlier sections, support for initializing containers using a pair of iterators requires a feature that is not yet widely supported by compilers. While we document this form of constructor, it may not yet be available on your system.

The priority queue data type 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 library that can be used to construct priority queues: vector***s*** or deque***s***. The following illustrates the declaration of several priority queues:

```
priority_queue< int, vector<int> > queue_one;
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);
```

Queues constructed out of vectors tend to be somewhat smaller, while queues constructed out of deques 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 will generally work in most circumstances.

Programs that utilize the priority queue data abstraction should include the file `queue`, as well as the include file for the container type (e.g., `vector`).

```
# include <queue>
# include <vector>
```

Because the priority queue data structure does not itself know how to construct iterators, very few of the algorithms noted in Generic algorithms overview can be used with priority queues. 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 the next section will illustrate this use.

Note: Details of the algorithms used in manipulating heaps will not be discussed here, however such information is readily available in almost any textbook on data structures.

Priority queues are implemented by internally building a data structure called a *heap*. Abstractly, a heap

is a binary tree in which every node possesses the property that the value associated with the node is smaller than or equal to the value associated with either child node.

Application: event-driven simulation

An extended example will illustrate the use of priority queues. The example illustrates one of the more common uses for 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.

Note: We describe the priority queue as a structure for quickly discovering the *largest* element in a sequence. If, instead, your problem requires the discovery of the *smallest* 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.

Events can be represented as subclasses of a base class, which we will call ***event***. The base class simply records the time at which the event will take place. A pure virtual function named `processEvent` will be 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 will need to maintain a collection of different types of events. Each different form of event will be represented by a different subclass of class ***event***. Not all events will have the same exact type, although they will all be subclasses of class ***event***. (This is sometimes called a *heterogeneous* collection.) For this reason the collection must store *pointers* to events, instead of the events themselves. (In theory one could store references, instead of pointers, however the standard library containers cannot hold references).

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 library this is accomplished by defining a new structure, the sole purpose of which is to define the function invocation operator (the `()` operator) in the appropriate fashion. Since in this particular example we wish to use the priority queue to return the *smallest* 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."

Note: Other example programs in this tutorial have all used containers to store values. In this example the container will maintain pointers to values, not the values them-selves. Note that a consequence of this is that the programmer is then responsible for managing the memory for the objects being manipulated.

```
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 events. In this case we are using a vector as the underlying container. We could just as easily 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:

```
void simulation::run()
{
  while (! eventQueue.empty()) {
    event * nextEvent = eventQueue.top();
    eventQueue.pop();
    time = nextEvent->time;
    nextEvent->processEvent();
    delete nextEvent;  // free memory used by event
    }
}
```

An ice cream store simulation

Note: The complete event simulation is found in the file `icecream.cpp` on the distribution disk.

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 that customers will arrive, the length of time they will stay, and so on.

Our store simulation will be 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. These are 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;
```

```
}
```

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 order event. The function `randomInteger()` (see Random access iterators) is used to compute a random integer between 1 and the argument value.

```
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;
}
```

Generic Algorithms

12.1 Generic algorithms overview
12.2 Initialization Algorithms
12.3 Searching Algorithms
12.4 In-Place Transformations
12.5 Removal Algorithms
12.6 Scalar Producing Algorithms
12.7 Sequence Generating Algorithms
12.8 Miscellaneous Algorithms

Generic algorithms overview

In this section and in section 11 we will examine and illustrate each of the generic algorithms provided by the standard library. The names and a short description of each of the algorithms in this section are given in the following table. We have divided the algorithms into several categories, based on how they are typically used. This division differs from the categories used in the C++ standard definition, which is based upon which algorithms modify their arguments and which do not.

Name	Purpose
algorithms used to initialize a sequence	
`fill`	fill a sequence with an initial value
`fill_n`	fill n positions with an initial value
`copy`	copy sequence into another sequence
`copy_backward`	copy sequence into another sequence
`generate`	initialize a sequence using a generator
`generate_n`	initialize n positions using generator
`swap_ranges`	swap values from two parallel sequences
searching algorithms	
`find`	find an element matching the argument
`find_if`	find an element satisfying a condition
`adjacent_find`	find consecutive duplicate elements
`search`	match a subsequence within a sequence
`max_element`	find the maximum value in a sequence
`min_element`	find the minimum value in a sequence
`mismatch`	find first mismatch in parallel sequences
in-place transformations	
`reverse`	reverse the elements in a sequence
`replace`	replace specific values with new value
`replace_if`	replace elements matching predicate
`rotate`	rotate elements in a sequence around a point
`partition`	partition elements into two groups
`stable_partition`	partition preserving original ordering
`next_permutation`	generate permutations in sequence
`prev_permutation`	generate permutations in reverse sequence
`inplace_merge`	merge two adjacent streams into one
`random_shuffle`	randomly rearrange elements in a sequence
removal algorithms	
`remove`	remove elements that match condition
`unique`	remove all but first of duplicate values in sequences
scalar generating algorithms	
`count`	count number of elements matching value
`count_if`	count elements matching predicate

`accumulate`	reduce sequence to a scalar value
`inner_product`	inner product of two parallel sequences
`equal`	check two sequences for equality
`lexicographical_compare`	compare two sequences
sequence generating algorithms	
`transform`	transform each element
`partial_sum`	generate sequence of partial sums
`adjacent_difference`	generate sequence of adjacent differences
miscellaneous operations	
`for_each`	apply a function to each element of collection

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>
```

In this section we will 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 sections on the various container classes. These cross references have been noted where appropriate.

All of the short example programs described in this section have been collected in a number of files, named `alg1.cpp` through `alg6.cpp`. In the files, the example programs have been augmented with output statements describing the test programs and illustrating the results of executing the algorithms. In order to 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 these test files. The expected output from these programs is also included in the distribution.

Initialization algorithms

Note: The sample programs described in this section can be found in the file `alg1.cpp`.

The first set of algorithms we will cover are those that are chiefly, although not exclusively, used to initialize a newly created sequence with certain values. The standard library provides several initialization algorithms. In our discussion we'll provide examples of how to apply these algorithms, and suggest how to choose one algorithm over another.

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 definitions are as follows:

```
void fill (ForwardIterator first, ForwardIterator last, const T&);
void fill_n (OutputIterator, Size, const T&);
```

Note: 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.

The example program illustrates several uses of the algorithm:

```
void fill_example ()
  // illustrate the use of the fill algorithm
{
    // 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 iterators as arguments, whereas the `fill_n()` algorithm uses a starting iterator and a count.

Example 2 illustrates how, by using an *insert iterator* (see Insert iterators), 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.

The third and fourth examples illustrate how `fill()` can be used to change the values in an existing container. In the third example 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, which we will describe in the next section, a vector is initialized to the values 1 2 3 ... 10. The `find()` algorithm (Find an element satisfying a condition) is then used to locate the position of the element 7, saving the location in an iterator appropriate for the vector data type. 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 library, although insert iterators must be used with ordered containers, such as a set.

Copy one sequence onto 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 library. The definitions for these algorithms are as follows:

```
OutputIterator copy (InputIterator first, InputIterator last,
       OutputIterator result);
```

```
BidirectionalIterator copy_backward
  (BidirectionalIterator first, BidirectionalIterator last,
  BidirectionalIterator result);
```

Note: The result returned by the copy() function is a pointer to the end of the copied sequence. To make a *catenation* of values, the result of one copy() operation can be used as a starting iterator in a subsequent copy().

Uses of the copy algorithm include:

- Duplicating an entire sequence by copying into a new sequence
- Creating subsequences 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

These are illustrated in the following sample program.

```
void copy_example()
  // illustrate the use of the copy algorithm
{
  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>(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;
}
```

The first call on `copy()`, in example 1, 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, i.e., copies of a sequence onto itself, as long as the destination iterator does not fall within the range formed by the source iterators. 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 will be that buffer will be 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, and not the elements themselves that is "backwards"; 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 Output stream iterators). 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, ptrdiff_t> in_stream(cin), eof;
copy(in_stream, eof, inserter(words, words.begin()));
```

This technique is used in the spell checking program described Example program: a spelling checker.

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 will return 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 definition of these algorithms is 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
  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);

    // 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 Functions,in the Standard Library a function is any object that will respond to the function call operator. Using this fact, classes can easily be constructed as functions. The class ***iotaGen***, which we described in Function objects is an example. 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 iterators in the function named `iter_swap()`. The algorithm `swap_ranges()` then 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);
```

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.

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. We use both functions alone and in combination in the example program.

```
void swap_example ()
  // illustrate the use of the algorithm swap_ranges
{
     // 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);
}
```

Searching operations

The next category of algorithms we will describe are those that 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` (Partition a sequence into two groups), a `partition` (Copy one sequence onto another sequence) or an `in-place merge` (Merge two adjacent sequences into one).

Note: The example functions described in this section can be found 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), and the resulting value is used as a starting point for the new search. For example, the following loop from the `adjacent_find()` example program (Find consecutive duplicate elements) will print the value of all repeated characters in a string argument.

Note: The searching algorithms in the standard 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.

```
while ((where = adjacent_find(where, stop)) != stop) {
  cout << "double " << *where << " in position "
     << where - start << endl;
  ++where;
  }
```

Many of the searching algorithms have an optional argument that can specify a function to be used to compare elements, in place of the equality operator for the container element type (operator `==`). 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 definitions 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 Predicates). 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 (the `*` 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 (the `==` operator) for the given datatype.

Note: These algorithms 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 these algorithms:

```
void find_test ()
  // illustrate the use of the find algorithm
{
  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 definition 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 (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 ()
  // illustrate the use of the adjacent_find instruction
{
  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 a subsequence within a sequence

The algorithm `search()` is used to locate the beginning of a particular subsequence 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 ]);
```

Note: 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.

Suppose, for example, that we wish 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
{
  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.

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 (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 ] );
```

Note: The maximum and minimum algorithms can be used with all the datatypes provided by the standard library. However, for the ordered data types, ***set*** and ***map***, the maximum or minimum values are more easily accessed as the first or last elements in the structure.

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()` used to divide a string into words in the string example is described in Example function: split a line into words. The function `randomInteger()` is described in Random access iterators.

```
void max_min_example ()
  // illustrate use of max_element and min_element algorithms
{
  // 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;
}
```

Locate the first mismatched elements in parallel sequences

The name `mismatch()` might lead you to think that this algorithm was the inverse of the `equal()` algorithm, which determines if two sequences are equal (see Test two sequences for pairwise equality). 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 The map data abstraction). 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 iterators denoting the locations of the two differing elements is constructed and returned. If the first sequence becomes 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)
   // illustrate the use of the mismatch algorithm
{
   pair<char *, char *> differPositions(0, 0);
   char * aDiffPosition;
   char * bDiffPosition;

   if (strlen(a) < strlen(b)) {
      // 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 the one illustrated, except it accepts a binary predicate as a fourth argument. This binary function is used to compare elements, in place of the `==` operator.

In-place transformations

Note: The example functions described in this section can be found in the file `alg3.cpp`.

The next category of algorithms in the standard library that we examine are those used to modify and transform sequences without moving them from their original storage locations. 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 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()` (Transform one or two sequences), or `partial_sum` (Partial sums), can also be used to modify a value in place by simply using the same iterator as both input and output specification.

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 the first, an array of characters values is reversed. The algorithm `reverse()` can also be used with list values, as shown in the second example. 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 Function objects). 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 ()
  // illustrate the use of the reverse algorithm
{
  // 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. In both cases 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.

```
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 ()
  // illustrate the use of the replace algorithm
{
    // 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 2nd 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, then swaps the order of the sections, maintaining the relative ordering of the elements within the two sections. 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 illustration presented earlier, that these two segments need not be the same length.

```
void rotate_example()
  // illustrate the use of the rotate algorithm
{
    // 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 (Find an element satisfying a condition) 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 (now containing a 3). The resulting list is 3 4 5 6 7 8 9 10 1 2. The values held in `iList` remain unchanged.

Partition 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 library. The first, provided by the algorithm `partition()`, guarantees only that the elements will be 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.

Note: While there is a unique `stable partition` for any sequence, the `partition()` algorithm can return any number of values. The following, for example, are all legal partitions of the example 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.

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 ()
  // illustrate the use of the partition algorithm
{
     // 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, yet in the result it may follow the element 8. A second version of partition, named `stable_partition()`, guarantees the ordering of the resulting values. For the sample 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()`.

Generate permutations in sequence

A permutation is a rearrangement of values. If values can be compared against each other (such as integers, characters, or words) then it is possible to systematically construct all permutations of a sequence. There are 2 permutations of two values, for example, and six permutations of three values, and 24 permutations of four values.

Note: Permutations can be ordered, with the smallest permutation being the one in which values are listed smallest to largest, and the largest being the sequence that lists values largest to smallest. Consider, for example, the permutations of the integers 1 2 3. The six permutations of these values are, in order:

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

Notice that in the first permutation the values are all ascending, while in the last permutation they are all descending.

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, only 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 ()
  // illustrate the use of the next_permutation algorithm
{
     // example 1, permute the values 1 2 3
  int start [] = { 1, 2, 3};
  do
     copy (start, start + 3,
          ostream_iterator<int> (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 *> (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 finally, `abel`.

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, described elsewhere, 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 (Find an element satisfying a condition) 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
{
    // 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. The results are, of course, unpredictable. Because the arguments must be random access iterators, this algorithm can only be used with vectors, deques, or ordinary pointers. It cannot be used with lists, sets, or maps.

```
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 will respond to the function invocation operator.

```
void random_shuffle_example ()
  // illustrate the use of the random_shuffle algorithm
{
  // 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);
}
```

Removal algorithms

Note: The algorithms in this section set up a sequence so that the desired elements are moved to the front. The remaining values are not actually removed, but the starting location for these values is returned, making it possible to remove these values by means of a subsequent call on erase(). *Remember*, the remove algorithms do not actually remove the unwanted elements.

The following two algorithms can be somewhat confusing the first time they are encountered. Both claim to remove certain values from a sequence. But, in fact, neither one 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 the user 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. The algorithm `remove_if()` 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 obtained 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 definition 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 (the 3 and the 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 ()
  // illustrate the use of the remove algorithm
{
  // 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 have been identified, the remainder of the sequence is left unchanged. For example, a sequence such as 1 3 3 2 2 2 4 will be changed into 1 3 2 4 **|** 2 2 4. I have used 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 ()
  // illustrate use of the unique algorithm
{
     // first make a list of values
  int data[] = {1, 3, 3, 2, 2, 4};
  list<int> aList;
  set<int> aSet;
  copy (data, , 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());
}
```

Algorithms that produce a scalar result

Note: The example functions described in this section can be found in the file `alg5.cpp`.

The next category of algorithms are those that reduce an entire sequence to a single scalar value.

Remember that two of these algorithms, `accumulate()` and `inner_product()`, are defined in the `numeric` header file, not the `algorithm` header file as are the other generic algorithms.

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. Both take as argument a reference to a counting value (typically an integer), and increment this value. Note that the count is passed as a by-reference argument, and is *not* returned as the value of the function. The `count()` function itself yields no value.

```
void count (InputIterator first, InputIterator last,
       const T&, Size &);
void count_if (InputIterator first, InputIterator last,
       Predicate, Size &);
```

Note: Note that the `count()` algorithms do not return the sum as a function result, but instead simply add to the last argument in their 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.

The example code fragment illustrates the use of these algorithms. The call on `count()` will count the number of occurrences of the letter `e` in a sample string, while the invocation of `count_if()` will count the number of vowels.

```
void count_example ()
    // illustrate the use of the count algorithm
{
  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;
}
```

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, `+`, however this can be replaced by any binary function. An initial value (an identity) must be provided. This value is returned for empty sequences, and is otherwise used as the left argument for the first calculation.

```
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 ()
  // illustrate the use of the accumulate algorithm
{
  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 are required to match the container type. This is illustrated in the example program by the invocation of `accumulate()` shown in the second example. 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 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 in place of 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 ()
     // illustrate the use of the inner_product algorithm
{
  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. The arguments are simple input iterators:

```
bool equal (InputIterator first, InputIterator last,
       InputIterator first2 [, BinaryPredicate] );
```

Note: By substituting another function for the binary predicate, the equal and mismatch algorithms can be put to a variety of different uses. Use the `equal()` algorithm if you want a pairwise test that returns a *boolean* result. Use the `mismatch()` algorithm if you want to discover the *location* of elements that fail the test.

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, and in this fashion true will be 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 ()
  // illustrate the use of the equal algorithm
{
  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 of two sequences can be described by noting the features of the most common example, namely the comparison of two words for the purposes of placing them in “dictionary order.” When comparing two words, the elements (that is, the characters) of the two sequences are compared in a pair-wise fashion. As long as they match, the algorithm advances to the next character. If two corresponding characters fail to match, the earlier character determines the smaller word. So, for example, `everybody` is smaller than `everything`, since the `b` in the former word alphabetically precedes the `t` in the latter word. Should one or the other sequence terminate before the other, than the terminated sequence is considered to be smaller than the other. So, for example, `every` precedes both `everybody` and `everything`, but comes after `eve`. Finally, if both sequences terminate at the same time and, in all cases, pair-wise characters match, then the two words are considered to be equal.

The `lexicographical_compare()` algorithm implements this idea, returning `true` if the first sequence is smaller than the second, and `false` otherwise. The algorithm has been generalized to any sequence. Thus the `lexicographical_compare()` algorithm can be used with arrays, strings, vectors, lists, or any of the other data structures used in the standard library.

```
bool lexicographical_compare
  (InputIterator first1, InputIterator last1,
  InputIterator first2, InputIterator last2 [, BinaryFunction ] );
```

Unlike most of the 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.

The example program illustrates the use of this algorithm with character sequences, and with arrays of integer values.

```
void lexicographical_compare_example()
  // illustrate the use of the lexicographical_compare algorithm
{
  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;
}
```

Sequence generating algorithms

Note: The example functions described in this section can be found in the file `alg6.cpp`.

The algorithms described in this section are all 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 Insert iterators), the algorithms can insert the new elements into a variable length structure, such as a set or list. Finally, in some cases which we will note, 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 described in the header file `numeric`, while the other functions are described in the header file `algorithm`.

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 iterators 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 either of the two input sequences.

```
int square(int n) { return n * n; }
void transform_example ()
  // illustrate the use of the transform algorithm
{
  // 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>());
}
```

Partial 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 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, in fact, 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 ()
  // illustrate the use of the partial sum algorithm
{
     // 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 arguments to the adjacent difference algorithm have the following definitions:

```
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 ()
  // illustrate the use of the adjacent difference algorithm
{
     // 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> (cout, " ")), cout << endl;

     // output adjacent sums
  adjacent_difference (aVec.begin(), aVec.end(),
     ostream_iterator<int> (cout, " "),
     plus<int>() );
}
```

Miscellaneous algorithms

In the final section we describe the remaining algorithms found in the standard library.

Apply a function to all elements in a collection

The algorithm `for_each()` takes three arguments. The first two provide the iterators that describe the sequence to be evaluated. The third is a one-argument function. The `for_each()` algorithm applies the function to each value of the sequence, passing the value as argument.

```
Function for_each
  (InputIterator first, InputIterator last, Function);
```

For example, the following code fragment, which uses the `print_if_leap()` function, will print 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;
```

Note: The function passed as the third argument is not permitted to make any modifications to the sequence, so it can only achieve any 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 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, too, is usually 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;
```

Ordered collection algorithms

13.1 Ordered collection algorithms overview
13.2 Sorting Algorithms
13.3 Partial Sort
13.4 Nth Element
13.5 Binary Search
13.6 Merge Ordered Sequences
13.7 Set Operations
13.8 Heap Operations

Ordered collection algorithms overview

In this section we will describe the generic algorithms in the standard library that are specific to ordered collections. These are summarized by the following table:

Name	Purpose
Sorting algorithms	
`sort`	rearrange sequence, place in order
`stable_sort`	sort, retaining original order of equal elements
`partial_sort`	sort only part of sequence
`partial_sort_copy`	partial sort into copy
Find nth largest element	
`nth_element`	locate nth largest element
Binary search	
`binary_search`	search, returning boolean
`lower_bound`	search, returning first position
`upper_bound`	search, returning last position
`equal_range`	search, returning both positions
Merge ordered sequences	
`merge`	combine two ordered sequences
Set operations	
`set_union`	form union of two sets
`set_intersection`	form intersection of two sets
`set_difference`	form difference of two sets
`set_symmetric_difference`	form symmetric difference of two sets
`includes`	see if one set is a subset of another
Heap operations	
`make_heap`	turn a sequence into a heap
`push_heap`	add a new value to the heap
`pop_heap`	remove largest value from the heap
`sort_heap`	turn heap into sorted collection

Ordered collections can be created using the standard 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 later in this section.

Like the generic algorithms described in the previous section, the algorithms described here are not specific to any particular container class. This means they can be used with a wide variety of types. Many of them do, however, require the use of random-access iterators. For this reason they are most easily used with vectors, deques, or ordinary arrays.

Note: The example programs described in this section have been combined and are included in the file `alg7.cpp` in the tutorial distribution. As we did in Generic algorithms overview, we will generally omit output statements from the descriptions of the programs provided here, although they are

included in the executable versions.

Almost all the algorithms described in this section have two versions. The first version uses for comparisons the less than operator (operator `<`) 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 Predicates). Since this argument is optional, we will write it within square brackets in the description of the argument types.

A sequence is considered to be ordered if for every valid (that is, denotable) iterator `i` with a denotable successor `j`, it is the case that 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`.

Algorithm include files

As with the algorithms described in Generic algorithms overview, before you can use any of the algorithms described in this section in a program you must include the `algorithm` header file:

```
# include <algorithm>
```

Sorting algorithms

There are two fundamental sorting algorithms provided by the standard library, 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 will retain their relative orderings in the final result. If order is important, then use the `stable_sort()` version.

Because these algorithms require random access iterators, they can be used only with vectors, deques, 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 iterators.

Note: Yet another sorting algorithm is provided by the heap operations, to be described in Heap operations.

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 ()
  // illustrate the use of the sort algorithm
{
    // 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());
}
```

Partial 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 will be 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 ]);
```

A second version of the algorithm leaves the input unchanged. The output area is described by a pair of random access iterators. If `n` represents the size of this area, then the smallest `n` elements in the input are moved into the output in order. If `n` is larger than the input, then 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 library. In the example program it is used with a list.

```
void partial_sort_example ()
  // illustrate the use of the partial sort algorithm
{
     // 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>());
}
```

Nth element

Imagine we have the sequence 2 5 3 4 7, and we want to discover the median, or middle element. We could 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 will be 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 ()
  // illustrate the use of the nth_element algorithm
{
    // 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;
}
```

Binary search

The standard library provides a number of different variations on binary search algorithms. All will 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 iterators, such as those generated by vectors or deques, when they will also perform approximately `log n` operations in total. However, they will also work with non-random access iterators, such as those generated by lists, in which case they will 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 defined 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 lower_bound (ForwardIterator first,
  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 will 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 iterators.

Our example program shows these functions being used with a vector of random integers.

```
void binary_search_example ()
  // illustrate the use of the binary search algorithm
{
     // 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);
}
```

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 will be described later in this section.

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 iterators, whereas the result is defined by a single output iterator. The arguments are defined as follows:

```
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 ()
  // illustrate the use of the merge algorithm
{
     // 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> (cout, " "));
  cout << endl;
}
```

The algorithm `inplace_merge()` (Merge two adjacent sequences into one) can be used to merge two sections of a single sequence into one sequence.

Set operations

The operations of set union, set intersection, and set difference were all described in Set operations 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 multisets; that is, elements can be repeated. However, if the inputs represent sets, then the result will always be a set. That is, unlike the `merge()` algorithm, none of the set algorithms will produce repeated elements in the output that were not present in the input sets.

The set operations all have the same format. The two input sets are specified by pairs of input iterators. 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, as well as 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 iterators, 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 ()
  // illustrate the use of the generic set algorithms
{
  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;
}
```

Heap operations

A *heap* is a binary tree in which every node is larger than the values associated with either child. A heap (and, for that matter, a binary tree) 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 will always be located in the initial position, and can therefore be very efficiently retrieved. In addition, efficient (logarithmic) algorithms exist that both 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* data type, described in The priority queue data abstraction.The default operator is the less-than operator (operator <) appropriate to the element type. If desired, an alternative operator can be specified. For example, by using the greater-than operator (operator >), one can construct a heap that will locate the smallest element in the first location, instead of the largest.

Note: Note that 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 algorithm `make_heap()` takes a range, specified by random access iterators, 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 ]);
```

A new element is added to a heap by inserting it at the end of a range (using the `push_back()` member function of a vector or deque, for example), followed by an invocation of 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, then 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 (accessible, for example, using the `back()` member function in a vector, and removable using the `pop_back()` member function), while 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 a ordered (sorted) collection. Note that a sorted collection is still a heap, although the reverse is not the case. 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 ()
  // illustrate the use of the heap algorithms
{
    // 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;

    // 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());
}
```

Exception handling

14.1 Eception handling overview
14.2 The Standard Exception Hierarchy
14.3 Using exceptions
14.4 Exception handling example program

Exception handling 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 that are due to 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 that are generated by circumstances outside the control of the program, such as peripheral hardware faults.

The standard exception hierarchy

The library implements the two-category error model described above with a set of classes. These classes are defined in the `stdexcept` header file. They can be used to catch exceptions thrown by the library and to throw exceptions from your own code.

The 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***

Classes ***logic_error*** and ***runtime_error*** inherit from class ***exception***. All other exception classes inherit from either ***logic_error*** or ***runtime_error***.

Using exceptions

All exceptions that are thrown explicitly by any element of the library are guaranteed to be part of the standard exception hierarchy. Review the reference 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, then 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("Your 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 at the end of this section.

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 will return 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 would use the following code:

```
throw exception;
```

This is generally not very useful, since whatever catches this exception will have 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*** as shown 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 demonstrates how the Standard C++ 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.

Example program: exception handling

Note: This program can be found in the file exceptn.cpp in your code distribution.

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 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 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;
}
```

auto_ptr

15.1 Overview
15.2 Creating and using Auto Pointers
15.3 Example Program

auto_ptr 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.

Creating and using auto pointers

Include the utility header file to access the ***auto_ptr*** class.

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 ***auto_ptr***'s copy constructor or assignment operator transfers ownership from one ***auto_ptr*** object to another. For instance, suppose we create ***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 will be 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 will not be affected. However, the pointer *will* be 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. But in this case, the ***auto_ptr***'s sole responsibility 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");
```

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.

Example program: auto_ptr

This program illustrates the use of ***auto_ptr*** to ensure that pointers held in a vector are deleted when they are removed. Often, we might want to hold pointers to strings, since the strings themselves may be quite large and we'll be copying them when we put them into the vector. Particularly in contrast to a string, an ***auto_ptr*** is quite small: hardly bigger than a pointer.

Note: You can find this program in the file autoptr.cpp in the tutorial distribution.

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

Complex

16.1 Complex overview
16.2 Creating and Using Complex Numbers
15.3 Example Program

Complex 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.

Creating and using complex numbers

In the following sections we will describe the operations used to create and manipulate complex numbers.

Header files

Programs that use complex numbers must include the `complex` header file.

```
# include <complex>
```

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 data types 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 + yi`, then the conjugate is the value `y + xi`.

```
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.

Note: Note that, with the exception of the member functions `real()` and `imag()`, most operations on complex numbers are performed using ordinary functions, not member functions.

```
  // the following should be the same
cout << com_one.real() << "+" << com_one.imag() << "i" << endl;
cout << real(com_one) << "+" << imag(com_one) << "i" << endl;
```

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.

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 parenthesis, either as (u) or (u,v), depending upon whether or not the imaginary value is zero. A value is read as a parenthesis 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 (namely, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `sinh()`, `cosh()`, and `tanh()`), have all been extended to complex number arguments. Each takes a single complex number as argument and returns a complex number as result. The function `atan2()` takes two complex number arguments, or a complex number and a real value (in either order), and returns a complex number 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 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.

Example program: roots of a polynomial

Note: This program is found in the file `complx.cpp` in the distribution.

The roots of a polynomial `a x2 + b x + c = 0` are given by the formula:

```
x = (-b ± sqrt(b2 - 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);
}
```

String

15.1 The String Abstraction
15.2 String Operations
15.3 AnExample Function

The string abstraction

A *string* is basically an indexable sequence of characters. In fact, although a string is not declared as a subclass of vector, almost all of the vector operators discussed in Vector operations can be applied to string values. However, a string is also a much more abstract quantity, and, in addition to simple vector operators, the string datatype provides a number of useful and powerful high level operations.

In the standard 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 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 data types string and wstring (for wide string) are simply typedefs of basic_string, defined as follows:

```
typedef basic_string<char,strint_char_traits<char> > string;
typedef basic_string<wchar_t> wstring;
```

Note: In the remainder of this section we will refer to the string data type, however all the operations we will 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 data type, there are two sizes associated with a string. 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 into a string without reallocation of a new internal buffer. As it is in the vector data type, 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.

String include files

Programs that use strings must include the `string` header file:

```
# include <string>
```

String operations

In the following sections, we'll examine the standard library operations used to create and manipulate strings.

Declaring string variables

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 Example program: graphs. 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. Alternative constructors let you explicitly set the initial capacity. Yet another form allows you to set the capacity and initialize the string with repeated copies of a single character value.

```
string s6 ("small value", 100);     // holds 11 values, can hold 100
string s7 (10, '\n');   // holds ten newline characters
```

Note: 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.

Finally, like all the container classes in the standard library, a string can be initialized using a pair of iterators. The sequence being denoted by the iterators must have the appropriate type of elements.

```
string s8 (aList.begin(), aList.end());
```

Resetting size and capacity

As with the vector data type, 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 (if necessary) adjusts the capacity 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. Usually this value is limited only by the amount of available memory.

```
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';
```

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. A single integer argument `n` indicates that only the first `n` characters should be assigned/appended, while two arguments, `pos` and `n`, indicate that the `n` values following position `pos` should be used.

```
s4.assign (s2, 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 library, the contents of two strings 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 a synonym for this operation.

```
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 strings 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 iterators for the string. The values denoted by the iterators will be individual string elements. The functions `rbegin()` and `rend()` return backwards iterators.

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.

Insertion, removal, and replacement

The string member functions `insert()` and `remove()` are similar to the vector functions `insert()` and `erase()`. Like the vector versions, they can take iterators as arguments, and specify the insertion or removal of the ranges specified by the arguments. The function `replace()` is a combination of remove and insert, in effect replacing the specified range with new values.

```
s2.insert(s2.begin()+2, aList.begin(), aList.end());
s2.remove(s2.begin()+3, s2.begin()+5);
s2.replace(s2.begin()+3, s2.begin()+6, s3.begin(), s3.end());
```

In addition, the functions 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 remove function takes two integer arguments, a position and a length, and removes the characters specified. And 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 an initial portion of a string, if the length has been explicitly specified).

```
s3.insert (3, "abc");   //insert abc after position 3
s3.remove (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 of the receiver, then assigns this substring to the 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

Note: Although the function is accessible, users will seldom invoke the member function compare() directly. Instead, comparisons of strings are usually performed using the conventional comparison operators, which in turn make use of the function compare().

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 Lexical comparison for a description of lexical ordering. The function returns a negative value if the receiver is lexicographically 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.

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 then 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
```

Example function: split a line into words

Note: The split function can be found in the concordance program in file `concord.cpp`.

In this section we will 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 Example program: a concordance.

There are three arguments to the function. The first two are strings, describing the line of text and the separators to be used to differentiate words, respectively. The third argument is a list of strings, 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, and 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.

Numeric limits

16.1 Numeric limits overview
16.2 Fundamental Data Types
16.3 Numeric Limit Members

Numeric limits overview

An new feature of the C++ Standard 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`, while the similar constant for a `short` would be known as `SHRT_MIN`, for a float `FLT_MIN`, and so on.

Note: For reasons 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 will, for the present, exist in parallel. 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.

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 data type, 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. The smallest character value is in this fashion yielded 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.

Solving this problem by 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.

Fundamental data types

The standard 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 library includes descriptions of the following fundamental data types.

`bool`	`char`	`int`	`float`
	`signed char`	`short`	`double`
	`unsigned char`	`long`	`long double`
	`wchar_t`	`unsigned short`	
		`unsigned int`	
		`unsigned long`	

Certain implementations my also provide information on other data types. 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 data type is not described by this mechanism.

```
cout << "are strings described " <<
  numeric_limits<string>::is_specialized << endl;
```

For data types that do not have a specialization, the values yielded by the functions and data fields in `numeric_limits` are generally zero or false.

Numeric limit 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

The following table summarizes the information available through the numeric_limits static member data fields and functions.

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 following members 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 data types.

Type	Name	Meaning
T	min()	the minimum positive normalized value
int	digits	the number of digits in the mantissa
int	radix	the base (or radix) of the exponent representation
T	epsilon()	the 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
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 ``Not 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` data type, the value in field `radix`, which represents the base of the exponential representation, is equivalent to the symbolic constant `FLT_RADIX`.

For the types float, double 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.

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.

Glossary

bidirectional iterator An iterator that can be used for reading and writing, and which can move in either a forward or backward direction.

binary function A function that requires two arguments.

binder A function adaptor that is used to convert a two-argument binary function object into a one-argument unary function object, by binding one of the argument values to a specific constant.

constant iterator An iterator that can be used only for reading values, which cannot be used to modify the values in a sequence.

container class A class used to hold a collection of similarly typed values. The container classes provided by the standard library include vector, list, deque, set, map, stack, queue, and priority_queue.

deque An indexable container class. Elements can be accessed by their position in the container. Provides fast random access to elements. Additions to either the front or the back of a deque are efficient. Insertions into the middle are not efficient.

forward iterator An iterator that can be used either for reading or writing, but which moves only forward through a collection.

function object An instance of a class that defines the parenthesis operator as one of its member functions. When a function object is used in place of a function, the parenthesis member function will be executed when the function would normally be invoked.

generic algorithm A templated algorithm that is not specialized to any specific container type. Because of this, generic algorithms can be used with a wide variety of different forms of container.

heap A way of organizing a collection so as to permit rapid insertion of new values, and rapid access to and removal of the largest value of the collection.

heterogeneous collection A collection of values that are not all of the same type. In the standard library a heterogeneous collection can only be maintained by storing pointers to objects, rather than objects themselves.

insert iterator An adaptor used to convert iterator write operations into insertions into a container.

iterator A generalization of the idea of a pointer. An iterator denotes a specific element in a container, and can be used to cycle through the elements being held by a container.

generator A function that can potentially return a different value each time it is invoked. A random number generator is one example.

input iterator An iterator that can be used to read values in sequence, but cannot be used for writing.

list A linear container class. Elements are maintained in sequence. Provides fast access only to the first and last elements. Insertions into the middle of a list are efficient.

map An indexed and ordered container class. Unlike a vector or deque, the index values for a map can be any ordered data type (such as a string or character). Values are maintained in sequence, and can be efficiently inserted, accessed or removed in any order.

multimap A form of map that permits multiple elements to be indexed using the same value.

multiset A form of set that permits multiple instances of the same value to be maintained in the collection.

negator An adaptor that converts a predicate function object, producing a new function object that when invoked yields the opposite value.

ordered collection A collection in which all values are ordered according to some binary comparison operator. The set data type automatically maintains an ordered collection. Other collections (vector, deque, list) can be converted into an ordered collection.

output iterator An iterator that can be used only to write elements into a container, it cannot be used to read values.

past the end iterator An iterator that marks the end of a range of values, such as the end of the set of values maintained by a container.

predicate A function or function object that when invoked returns a boolean (true/false) value or an integer value.

predicate function A predicate.

priority_queue An adaptor container class, usually built on top of a vector or deque. The priority queue is designed for rapidly accessing and removing the largest element in the collection.

queue An adaptor container class, usually built on top of a list or deque. The queue provides rapid access to the topmost element. Elements are removed from a queue in the same order they are inserted into the queue.

random access iterator An iterator that can be subscripted, so as to access the values in a container in any order.

range A subset of the elements held by a container. A range is typically specified by two iterators.

reverse iterator An iterator that moves over a sequence of values in reverse order, such as back to front.

sequence A portion or all of the elements held by a container. A sequence is usually described by a range.

set A ordered container class. The set container is optimized for insertions, removals, and tests for inclusion.

stack An adaptor container class, built usually on top of a vector or deque. The stack provides rapid access to the topmost element. Elements are removed from a stack in the reverse of the order they are inserted into the stack.

stream iterator An adaptor that converts iterator operations into stream operations. Can be use to either read from or write to an iostream.

unary function A function that requires only one argument. Applying a binder to a binary function results in a unary function.

vector An indexable container class. Elements are accessed using a key that represents their position in the container. Provides fast random access to elements. Addition to the end of a vector is efficient. Insertion into the middle is not efficient.

wide string A string with 16-bit characters. Wide strings are necessary for many non-roman alphabets, i.e., Japanese.

Standard C++ Library class reference

This reference guide is an alphabetical listing of all of the classes, algorithms, and function objects provided by this release of Rogue Wave's Standard C++ Library. The gray band on the first page of each entry indicates the category (e.g., algorithms, containers, etc.) that the class belongs to. The tables on the next few pages provide a listing of the classes organized by category.

For each class, the reference begins with a brief summary of the class, and a synopsis, which indicates the header file(s), a declaration and definition of a class object, and any type definitions for the class. The reference continues with a description and, in most cases, an example. All methods associated with a class, including constructors, operators, member functions, etc., are grouped in categories according to their general use and described. The categories are not a part of the C++ language, but do provide a way of organizing the methods.

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

Standards conformance

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

Algorithms

`#include <algorithm>`	adjacent_find
	binary_search
	copy
	copy_backward
	count
	count_if
	equal
	equal_range
	fill
	fill_n
	find
	find_first_of
	find_if
	for_each
	generate
	generate_n
	includes
	inplace_merge
	iter_swap
	lexicographical_compare
	lower_bound
	make_heap
	max
	max_element
	merge
	min
	min_element
	mismatch
	next_permutation
	nth_element
	partial_sort
	partial_sort_copy
	partition
	pop_heap
	prev_permutation
	push_heap

random_shuffle
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
search
set_difference
set_intersection
set_symmetric_difference
set_union
sort
sort_heap
stable_partition
stable_sort
swap
swap_ranges
transform
unique
unique_copy
upper_bound

Complex number library

`#include <complex>`	complex

Containers

`#include <deque>`	deque
`#include <list>`	list
`#include <map>` for map and multimap	map
	multimap
	multiset
`#include <queue>` for queue and priority_queue	priority_queue
	queue
`#include <set>` for set and multiset	set

Header	Name
#include <stack>	stack
#include <vector>	vector

Function adaptors

Header	Name
#include <functional>	bind1st
	bind2nd
	not1
	not2
	ptr_fun

Function objects

Header	Name
#include <functional>	binary_function
	binary_negate
	binder1st
	binder2nd
	divides
	equal_to
	greater
	greater_equal
	less
	less_equal
	logical_and
	logical_not
	logical_or
	minus
	modulus
	negate
	not_equal_to
	plus
	pointer_to_binary-function
	pointer_to_unary_function
	times
	unary_function

Generalized numeric operations

Header	Name
#include <numeric>	accumulate
	adjacent_difference
	accumulate
	inner_product
	partial_sum

Insert iterators

Header	Name
#include <iterator>	back_insert_iterator

back_inserter
front_insert_iterator
front_inserter
insert_iterato
inserter

Iterators

#include <iterator>

bidirectional iterator
forward iterator
input iterator
output iterator
random access iterator
reverse_bidirectional_iterator
reverse_iterator

Iterator operations

#include <iterator>

advance
distance

Memory handling primitives

#include <memory>

allocate
construct
deallocate
destroy
get_temporary_buffer
return_temporary_buffer

Memory management

#include <memory>

raw_storage_iterator
uninitialized_copy
uninitialized_fill
uninitialized_fill_n

Numeric limits library

#include <limits>

numeric limits

String library

#include <string>

basic_string
string
wstring

Utility classes

#include <utility>

pair

Utility operators

#include <utility>

operator!=

operator>
operator<=
operator>=

accumulate

Generalized numeric operation

Accumulate all elements within a range into a single value.

Syntax

```
#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

This algorithm accumulates, or "sums" all elements in the range `[first, last)` into a single value. For instance, applying ***accumulate*** to the sequence {1, 2, 3} will produce {6}.

The first version of the algorithm uses plus (+) as the default operator. The second version lets you specify any binary operation.

To avoid the problem of accumulating an empty range, ***accumulate*** requires an initial value, `init`, that is used as the de facto first element of the accumulation. Usually `init` is the identity value for the binary operation of ***accumulate***. (For example, in the default version of the algorithm, `init` is normally equal to 0. If you specify that `binary_op` is multiplication, then `init` would normally be equal to 1.)

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`.

accumulate performs exactly `last-first` applications of the binary operation.

Example

```
#include <numeric>   //for accumulate
#include <vector>    //for vector
#include <functional> //for times
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;
}
```

adjacent_difference

Generalized numeric operation

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

Syntax

```
#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} will produce 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))
```

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

The iterator that `adjacent_difference` returns is equal to `result + (last - first)`.

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

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

Example

```
#include<numeric>        //For adjacent_difference
#include<vector>         //For vector
#include<functional>     //For times
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>(cout," "));
  cout << endl << endl;

  cout << "The differences between adjacent elements are: "
       << endl << "     ";
  copy(diffs.begin(),diffs.end(),
       ostream_iterator<int>(cout," "));
  cout << endl << endl;

  cout << "The products of adjacent elements are: "
       << endl << "     ";
  copy(prods.begin(),prods.end(),
       ostream_iterator<int>(cout," "));

  cout << endl;
  return 0;
}
```

adjacent_find

Algorithm

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

Syntax

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

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

Example

```
#include <vector>
 #include <algorithm>
 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);
   // 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;
}
```

advance

Iterator operation

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

Syntax

```
#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 bi-directional and random access iterators, this distance may be negative. This function uses operator + and operator - for random access iterators, which provides a constant time implementation. For input, forward, and bi-directional iterators, ***advance*** uses operator ++ to provide linear time implementations. ***advance*** also uses operator -- with bi-directional iterators operator to provide 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 bi-directional iterators only.

Example

```
#include<iterator>
#include<list>
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);
  //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>(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;
}
```

Algorithms

Generic algorithms for performing various operations on containers and sequences.

Syntax

The synopsis of each algorithm appears in its entry in the *Reference Guide.*

Description

The Standard C++ Library provides a very flexible framework for applying generic algorithms to containers. The library also provides a rich set of these algorithms for searching, sorting, merging, transforming, scanning, and much 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 templated 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.

There are several ways to group algorithms. The broadest categorization groups the algorithms based on whether or not they change the elements in the sequence. Those algorithms that alter (or mutate) the contents of a container fall into the mutating group. All others are considered non-mutating. Algorithms can also be grouped according to the type of operations they perform. Finally, because most algorithms rely on iterators to gain access to data, they can be grouped according to the type of iterator they require. The following three sections provide lists of algorithms grouped according to these criteria.

Algorithms by mutating/non-mutating function

The broadest categorization groups algorithms into two main types: mutating and non-mutating. Those 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

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

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 swap
make_heap                       rotate_copy
merge                           set_difference
nth_element                     set_symmetric_difference
next_permutation                set_intersection
partial_sort                    set_union
```

partial_sort_copy
partition
prev_permutation
push_heap
pop_heap
random_shuffle
remove
remove_copy
remove_copy_if
sort
sort_heap
stable_partition
stable_sort
swap
swap_ranges
transform
unique
unique_copy

Note that the library provides both in place and copy versions of many algorithms, such as ***replace*** and ***replace_copy***. The library also provides 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 (e.g., ***replace***, which will use 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
fill_n
generate
generate_n

Search operations

adjacent_find
count
count_if
find
find_if
find_first_of
search

Binary search operations (Elements must be sorted)

binary_search
equal_range
lower_bound
upper_bound

Compare operations

equal
lexicographical_compare
mismatch

Copy operations

copy
copy_backward

Transforming operations

partition
random_shuffle
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
stable_partition
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 iterator 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

Require only input_iterators

accumulate	find	mismatch
count	find_if	
count_if	includes	
equal	inner_product	
for_each	lexicographical_compare	

Require only output_iterators

fill_n	generate_n

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_intersedtion	
remove_copy	set_symmetric_difference	
remove_copy_if	set_union	

Require forward iterators

adjacent_find	lower_bound	rotate
binary_search	max_element	search
equal_range	min_element	swap_ranges

fill	remove	unique
find_first_of	remove_if	upper_bound
generate	replace	
iter_swap	replace_if	

Read from forward_iterators and write to output_iterators

rotate_copy

Require bidirectional_iterators

copy_backward	partition
inplace_merge	prev_permutation
next_permutation	reverse
	stable_permutation

Read from bidirectional_iterators and write to output_iterators

reverse_copy

Require random access iterators

make_heap	pop_heap	sort
nth_element	push_heap	sort_heap
partial_sort	random_shuffle	stable_sort

Read from input_iterators and write to random_access_iterators

partial_sort_copy

allocate

See also   Memory handling primitive

Pointer based primitive for handling memory.

Syntax

```
#include <memory>
template <class T>
 T* allocate (ptrdiff_t n, T*);
```

Description

allocate reserves an uninitialized memory buffer of size `n* sizeof(T)`, in system memory and returns a typed pointer to that buffer.

associative containers

Associative containers are ordered containers. These containers provide member functions that allow the efficient insertion, retrieval and manipulation of keys. The standard library provides the ***map***, ***multimap***, ***set*** and ***multiset*** associative containers. ***map*** and `multimnap` 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.

auto_ptr

Memory management

A simple, smart pointer class.

Syntax

```
#include <memory>
template <class X> class auto_ptr {
 public:
  // constructor/copy/destroy
  explicit auto_ptr (X* p = 0);
  auto_ptr (auto_ptr<X>&);
  void operator= (auto_ptr<X>&);
  ~auto_ptr ();
  // members
  X& operator* () const;
  X* operator-> () const;
  X* get () const;
  X* release ();
  void reset (X* p = 0);
};
```

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 provides 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.

Constructor

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

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

```
auto_ptr (auto_ptr<X>& a);
```

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

Destructor

```
~auto_ptr ();
```

Deletes the underlying pointer.

Operators

```
void
operator= (auto_ptr<X>& a);
```

Assignment operator. Copies the argument `a` to `*this`. `*this` becomes the new owner of the underlying 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.

Member functions

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

Returns the underlying pointer.

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

Releases ownership of the underlying pointer. Returns that pointer.

```
void
reset (X* p = 0);
```

Requires that `p` points to an object of class `X` or a class derived from `X` for which `delete p` is defined and accessible, or `p` is a null pointer. Deletes the current underlying pointer, then resets it to `p`.

Example

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

back_insert_iterator, back_inserter

See also Insert iterator

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

Syntax

```
#include <iterator>
template <class Container>
 class back_insert_iterator : public output_iterator {
protected:
   Container& container;
public:
   back_insert_iterator (Container& x);
   back_insert_iterator<Container>&
   operator= (const Container::value_type& 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& x)
```

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.

Example

```
/***************************************************************
 *
 * ins_itr.cpp - Example program of insert iterator.
 *
 * $Id: ins_itr.cpp,v 1.7 1995/10/06 18:18:03 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <iterator>
 #include <deque>
 using namespace std;
 int main ()
 {
  //
  // 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;
 }
```

Constructor

```
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 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 function

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

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

basic_string

String library

A templated 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.

Specializations

```
basic_string <char>
basic_string <wchar_t>
```

Syntax

```
#include <string>
template <class charT,
     class traits = string_char_traits<charT>,
     class Allocator = allocator>
class basic_string {
public:
// Types
typedef traits                               traits_type;
typedef typename traits::char_type           value_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 reverse_iterator<const iterator,
                    value_type,
                    const reference,
                    difference_type>
const_reverse_iterator;
typedef reverse_iterator<iterator,
                    value_type,
                    reference,
                    difference_type>    reverse_iterator;
static const size_type npos = -1;
// Constructors/Destructors
explicit basic_string(Allocator& = Allocator());
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, Allocator& = Allocator());
basic_string(const charT* s, size_type n,
             Allocator& = Allocator());
basic_string(const charT* s, Allocator& = Allocator());
basic_string(size_type n, charT c,
             Allocator& = Allocator());
template <class InputIterator>
basic_string(InputIterator begin, InputIterator end,
             Allocator& = Allocator());
~basic_string();
// Assignment operators
 basic_string& operator=(const basic_string& str);
```

```
 basic_string& operator=(const charT* s);
 basic_string& operator=(charT c);

// 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;
   size_type       max_size() const;
   void            resize(size_type n, charT c);
   void            resize(size_type n);
   size_type       capacity() const;
   void            reserve(size_type res_arg);
   bool            empty() const;

// Element access
   charT           operator[](size_type pos) const;
   reference       operator[](size_type pos);
   const_reference at(size_type n) const;
   reference       at(size_type n);

// Modifiers
   basic_string& operator+=(const basic_string& rhs);
   basic_string& operator+=(const charT* s);
   basic_string& operator+=(charT c);
   basic_string& append(const basic_string&,
                     size_type pos = 0,
                     size_type = npos);
   basic_string& append(const charT* s, size_type n);
   basic_string& append(const charT* s);
   basic_string& append(size_type n, charT c = charT());
   template<class InputIterator>
   basic_string& append(InputIterator, InputIterator);

   basic_string& assign(const basic_string& str,
                     size_type pos = 0,
                     size_type n = npos);
   basic_string& assign(const charT* s, size_type n);
   basic_string& assign(const charT* s);
   basic_string& assign(size_type n, charT c = charT());
   template<class InputIterator>
   basic_string& assign(InputIterator, InputIterator);

   basic_string& insert(size_type pos1,
                     const basic_string& str,
                     size_type pos = 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 = charT());
```

```
    iterator insert(iterator p, charT c = charT());
    iterator insert(iterator p, size_type n,
                    charT c = charT() );
    template<class InputIterator>
    void insert(iterator p, InputIterator first,
                InputIterator last);

    basic_string& remove(size_type pos = 0,
                         size_type n = npos);
    basic_string& remove(iterator position);
    basic_string& remove(iterator first, iterator last);

    basic_string& replace(size_type pos, size_type n1,
                          const charT* s, size_type n2);
    basic_string& replace(size_type pos1, size_type n1,
                          const basic_string& str,
                          size_type pos2 = 0,
                          size_type n2 = npos);
    basic_string& replace(size_type pos, size_type n,
                          const charT* s);
    basic_string& replace(size_type pos, size_type n,
                          charT c = charT());
    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 = charT() );
    template<class InputIterator>
    basic_string& replace(iterator i1, iterator i2,
                          InputIterator j1,
                          InputIterator j2);

    size_type copy(charT* s, size_type n, size_type pos = 0);
    void swap(basic_string<charT, traits, Allocator>&);

 // String operations
    const charT* c_str() const;
    const charT* data() const;

    size_type find(const basic_string& str,
                   size_type pos = 0) const;
    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;

    size_type rfind(const basic_string& str,
                    size_type pos = npos) const;
    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;

    size_type find_first_of(const basic_string& str,
                            size_type pos = 0) const;
    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;
   size_type find_last_of(const basic_string& str,
                             size_type pos = npos) const;
   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;
   size_type find_first_not_of(const basic_string& str,
                                  size_type pos = 0) const;
   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;
   size_type find_last_not_of(const basic_string& str,
                                 size_type pos = npos) const;
   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;
   basic_string substr(size_type pos = 0,
                          size_type n = npos) const;
   int compare(const basic_string& str, size_type pos = 0,
                size_type n = npos) const;
   int compare(charT* s, size_type pos, size_type n) const;
   int compare(charT* s, size_type pos = 0) const;
};
```

Description

basic_string<charT, traits, allocator> is a homogeneous collection of character-like entities. It provides general string functionality such as compare, append, assign, insert, remove, replace and various searches. ***basic_string*** also functions as an STL sequence container, providing 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 `string_char_traits` class is provided or the default traits class is applicable.

Example

```
#include<string>
using namespace std;
int main()
{
  string test, result;
  //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 << endl << "You typed in: " << test << endl << endl;
   cout << "Changing the third character from " << test[2] <<
             " to * " << endl;
   test[3] = '*';
   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;
 }
```

Constructors and destructors

In all cases, the `Allocator` parameter specifies the storage for the constructed string.

```
explicit
basic_string (const Allocator& a = Allocator());
```

The default constructor. Creates a ***basic_string*** of length zero.

```
explicit
```

basic_string (size_type size, const Allocator& a = Allocator());

Creates a string of `size` entities.

```
basic_string (const basic_string<T, traits,
                Allocator>& s);
```

Creates a string that is a copy of `s`.

```
basic_string (const basic_string<T, traits, Allocator>&s,
              size_type pos,
              const Allocator& a = Allocator());
```

Creates a string that is a copy of `s` starting at character `pos`.

```
basic_string (const charT * s, size_type n,
              const Allocator& a = Allocator());
```

Creates a string that contains the first `n` characters of `s`.

```
basic_string (const charT * s,
              const Allocator& a = Allocator());
```

Creates a string containing all characters in `s` up to, but not including, a `traits::eos()` character. `s` cannot be a null pointer.

```
basic_string (size_type n, charT c,
              const Allocator& a  = Allocator());
```

Creates a string containing `n` repetitions of `c`.

```
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)`.

```
~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.

```
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`.

```
charT
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 `traits::eos()`, the behavior of the non-`const` version is undefined. The reference returned is invalidated by any call to `c_str()`, `data()`, or any non-`const` member function.

```
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 ();
iterator begin () const;
```

Return an iterator initialized to the first element of the string.

```
iterator end ();
iterator end () const;
```

Return an iterator initialized to the position after the last element of the string.

```
iterator rbegin ();
iterator rbegin () const;
```

Returns an iterator equivalent to `reverse_iterator(end())`.

```
iterator rend ();
iterator rend () const;
```

Returns an iterator equivalent to `reverse_iterator(begin())`.

Member functions

```
basic_string&
append (const basic_string& s, size_type
        pos = 0, size_type n = npos);

basic_string&
append (const charT* s, size_type n);

basic_string&
append (const charT* s);

basic_string&
append (charT c);

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 function appends the lesser of `n` and `s.size() - pos` characters of `s`, beginning at position `pos` to this string. This member will throw an `out_of_range` exception if `pos > str.size()`. The second member appends `n` characters of the array pointed to by `s`. The third variation appends elements from the array pointed to by `s` up to, but not including, a `traits::eos()` character. The fourth and fifth variations append one or `n` repetitions of `c`, respectively. The final `append` function appends the elements specified in the range `[first, last)`.

All functions will throw a `length_error` exception if the resulting length will exceed `max_size()`. All return a reference to this string after completion.

```
basic_string&
assign (const  basic_string& s,
       size_type pos = 0, size_type n = npos);

basic_string&
assign (const charT* s, size_type n);

basic_string&
assign (const charT* s);

basic_string&
assign (charT c);

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 variation assigns the lesser of `n` and `s.size() - pos` characters of `s`, beginning at position `pos`. It throws an `out_of_range` exception if `pos > str.size()`. The second version of the function assigns `n` characters of the array pointed to by `s`. The third version assigns elements from the array pointed to by `s` up to, but not including, a `traits::eos()` character. The fourth and fifth assign one or `n` repetitions of `c`, respectively. The last variation assigns the members specified by the range `[first, last)`.

All functions will throw a `length_error` exception if the resulting length will exceed `max_size()`. All return a reference to this string after completion.

```
const_reference
at (size_type n) const;
```

```
reference
at (size_type n);
```

If `n < size()`, returns the element at position `n` 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()`.

```
int
compare (const basic_string& str,
         size_type pos =  0, size_type n = npos);
```

Returns the result of a lexigraphical comparison between elements of this string and elements of `str`. Throws an `out_of_range` exception if `pos > size()`. The return value is:

```
traits::compare (data()+pos, str.data(),
                  min{n, size()-pos, str.size()}).
int
```

```
compare (const charT* s, size_type pos,
         size_type n) const;
int
compare (const charT* s, size_type pos = 0) const;
```

Return the result of a lexigraphical comparison between elements of this string and a given comparison string. The members return, respectively:

```
compare(basic_string(s, n), pos)
compare(basic_string(s), pos)
```

```
size_type
copy (charT* s, size_type n,  size_type pos = 0);
```

Replaces elements in memory with copies of elements from this string. An `out_of_range` exception will be 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;
```

Return a pointer to the initial element of an array whose first `size()` elements are copies of the elements in this string. A `traits::eos()` 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 `c_str()` returns a pointer to a `traits::eos()` character. See also the `data()` function.

```
const charT*
data () const;
```

Return a pointer to the initial element of an array whose first `size()` elements are copies of the elements in this string. A `traits::eos()` 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 `NULL` pointer. See also the `c_str()` function.

```
bool empty () const;
```

Returns `size() == 0`.

```
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;
```

Search 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;
```

Search 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;
```

Search for the first occurrence in this string of any element in a specified string. The `find_first_of` variations return, respectively:

```
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 = 0) const;
size_type
find_last_not_of (charT c, size_type pos = 0) const;
```

Search 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 = 0) const;
size_type
find_last_of (charT c, size_type pos = 0) const;
```

Search 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 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_strint&
insert (size_type pos, charT c);
basic_string&
```

```
insert (size_type pos, size_type n, charT c);
```

Insert additional elements at position `pos` in this string. All of the variants of this function will throw an `out_of_range` exception if `pos > size()`. All variants will also throw a `length_error` if the resulting string will exceed `max_size()`. Elements of this string will be moved apart as necessary to accommodate the inserted elements. All return a reference to this string after completion.

The first 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 will throw an `out_of_range` exception if `pos2 > s.size()`. The second version inserts `n` characters of the array pointed to by `s`. The third inserts elements from the array pointed to by `s` up to, but not including, a `traits::eos()` character. Finally, the fourth and fifth variations insert one or `n` repetitions of `c`.

```
iterator
insert (iterator p, charT c);
```

```
iterator
insert (iterator p, size_type n, charT c);
```

```
template<class InputIterator>
void
insert (iterator p, InputIterator first, InputIterator last);
```

Insert 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 two versions return `p`.

```
size_type
length () const;
```

Return 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 this string, starting at position `pos`. Note that only the first character of the substring must be `<= pos`; the remaining characters may extend beyond `pos`. If found, the index of the first character of 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 = 0) const;
```

```
size_type
rfind (charT c, size_type pos = 0) 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&
remove (size_type pos = 0,  size_type  n  = npos);
```

```
basic_string&
```

```
remove (iterator p);

basic_string&
remove (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 will be thrown if `pos > size()`. The second version requires that `p` is a valid iterator on this string, and removes the character referred to by `p`. The last version of `remove` requires that both `first` and `last` are valid iterators on this string, and removes the characters defined by the range `[first, last)`. The destructors for all removed characters are called. All versions of `remove` return a reference to this string after completion.

```
basic_string&
replace (size_type pos, size_type n1, const basic_string& s,
         size_type pos2 = 0, size_type n2 = npos);

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, 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 first version replaces elements of the original string with `n2` characters from string `s` starting at position `pos2`. It will throw the `out_of_range` exception if `pos2 > s.size()`. The second variation of the function replaces elements in the original string with `n2` elements from the array pointed to by `s`. The third version replaces elements in the string with elements from the array pointed to by `s`, up to, but not including, a `traits::eos()` character. The fourth replaces elements with `n` repetitions of character `c`.

```
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 = charT());

template<class InputIterator>
basic_string&
replace (iterator  i1,  iterator  i2,
         InputIterator j1, InputIterator j2);
```

Replace 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 with 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 traits::eos() 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);
```

Assures that the storage capacity is at least res_arg.

```
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 it 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 traits::eos().

```
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.

```
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 members 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 lexigraphical 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>
bool
operator < (const basic_string&  lhs, const charT* rhs)
```

Returns a boolean value representing the lexigraphical 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 lexigraphical 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 lexigraphical greater-than relationship of `lhs` and `rhs`. Greater-than is defined by the `compare()` member. The members 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 lexigraphical 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 lexigraphical 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 lexigraphical 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 lexigraphical greater-than-or-equal relationship of `lhs` and `rhs`. Greater-than-or-equal is defined by the `compare()` member. The members return, respectively:

```
basic_string(lhs) >= rhs
lhs >= basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
basic_istream<charT>&
```

```
operator >> (basic_istream<charT>& is,
             basic_string<charT,
             traits, Allocator>& str);
```

Reads `basic_string<charT, traits, Allocator>` 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>&
operator << (basic_ostream<charT>& os,
             const basic_string<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 function

```
template <class charT, class IS_traits, class STR_traits,
          class STR_Alloc>
basic_istream<charT, IS_traits>&
getline (basic_istream<charT, IS_traits>& is,
         basic_string<charT, STR_traits, STR_Alloc>& str,
         charT delim = IS_traits::newline() );
```

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 `STR_traits::char_in()`.

If the new `iostreams` is not available on your system, we provide the equivalent `operator >>`, `operator <<`, and `getline` functions that work with the old `iostreams`.

bidirectional iterator

See also Iterator

An iterator that can both read and write and can traverse a container in both directions.

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.

The following key pertains to the iterator descriptions 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 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`
`a == b, a != b`	return value convertable to `bool`
`*a`	return value convertable to `T&`
`++r`	returns `X&`
`r++`	return value convertable to const `X&`
`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return `value` convertable 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.

binary_function

See also Function object

Abstract base function for binary function objects.

Syntax

```
#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 library provides 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 provide the typedefs `first_argument_type`, `second_argument_type`, and `result_type`. The ***binary_function*** class makes the task of creating templated binary function objects easier by providing the necessary typedefs for a binary function object. You can create your own binary function objects by inheriting from ***binary_function***.

binary_negate

See also Function object

Function object that returns the complement of the result of its binary predicate.

Syntax

```
#include <functional>
template<class Predicate>
class binary_negate
  : public binary_function<Predicate::first_argument_type,
                           Predicate::second_argument_type,
                           bool                            >
{
public:
   explicit binary_negate (const Predicate& pred);
   bool operator() (const first_argument_type& x,
                    const second_argument_type& y) const;
};
template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Description

binary_negate is a function object class that provides 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***.

Constructor

```
explicit binary_negate (const Predicate& pred);
```

Construct a binary_negate object from predicate `pred`.

Operator

```
bool operator() (const first_argument_type& x,
                 const second_argument_type& y) const;
```

Return the result of `pred(x,y)`.

binary_search

See also Algorithm

Performs a binary search for a value on a container.

Syntax

```
#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
```

binary_search performs at most `log(last - first) + 2` comparisons.

Example

```
#include <vector>
#include <algorithm>
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
  bool b1 = binary_search(v1.begin(),v1.end(),3);
  // b1 = true
  bool b2 =
     binary_search(v1.begin(),v1.end(),11,less<int>());
  // b = false
  // Output results
  cout << "In the vector: ";
```

```
  copy(v1.begin(),v1.end(),
          ostream_iterator<int>(cout," "));
  cout << endl << "The number 3 was "
       << (b1 ? "FOUND" : "NOT FOUND");
  cout << endl << "The number 11 was "
       << (b2 ? "FOUND" : "NOT FOUND") << endl;
  return 0;
}
```

bind1st, bind2nd, binder1st, binder2nd

Function object

Templatized utilities to bind values to function objects.

Syntax

```
#include <functional>
// Class binder1st template <class Operation>
 class binder1st
     : public unary_function<Operation::second_argument_type,
                             Operation::result_type> {
protected:
   Operation op; argument_type value;
public:
   binder1st(const Operation& x, const
               Operation::first_argument_type& y) : op(x). value(y) {}
   result_type operator() (const argument_type& x) const
};
// Creator bind1st template<class Operation, class T>
 binder1st <Operation> bind1st(const Operation& op, const T& x)
// Class binder2nd template <class Operation>
 class binder2nd
     : public unary_function<Operation::first_argument_type,
                               Operation::result_type> {
protected:
   Operation op; argument_type value;
public:
   binder2nd(const Operation& x, const
             Operation::second_argument_type& y) : op(x). value(y) {}
   result_type operator() (const argument_type& x) const
};
// Creator bind2nd template<class Operation, class T>
 binder2nd <Operation> bind2nd(const Operation& op, const T& x)
```

Description

Because so many functions provided by 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***. 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.

The `bind1st()` and `bind2nd()` member functions are used inline to create the unary predicates. 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)
```

This function adds one to `littleNums` each time the predicate is `true`, i.e., each time 7 is greater than the element.

Example

```
#include <functional>
#include <algorithm>
#include <vector>
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);

  // Now use this new predicate in a call to find_if
  iterator it1 = find_if(v1.begin(),v1.end(),equal_to_3);
  // it1 = v1.begin() + 2

  // Even better, construct the new predicate on the fly
  iterator it2 =
     find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
  // it2 = v1.begin() + 2

  // 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;
}
```

bitset

Container

A template class and related functions for storing and manipulating fixed-size sequences of bits.

Syntax

```
#include <bitset>
template <size_t N>
 class bitset {
public:
// bit reference:
 class reference {
 public:
  ~reference()
  reference& operator=(bool x);
  reference& operator=(const reference&);
  bool operator~() const;
  operator bool() const;
  reference& flip();
 };
// Constructors
 bitset ();
 bitset (unsigned long val);
 explicit bitset (const string& str,
          size_t pos = 0,
          size_t n = size_t(-1));
// Bitwise Operators and Bitwise Operator Assignment
  bitset<N>& operator&= (const bitset<N>& rhs);
  bitset<N>& operator|= (const bitset<N>& rhs);
  bitset<N>& operator^= (const bitset<N>& rhs);
  bitset<N>& operator<<= (size_t pos);
  bitset<N>& operator>>= (size_t pos);
// Set, Reset, Flip
  bitset<N>& set ();
  bitset<N>& set (size_t pos, int val = 1);
  bitset<N>& reset ();
  bitset<N>& reset (size_t pos);
  bitset<N> operator~() const;
  bitset<N>& flip ();
  bitset<N>& flip (size_t pos);
// element access
  reference operator[] (size_t pos);
  unsigned long to_ulong() const;
  string to_string() const;
  size_t count() const;
  size_t size() const;
  bool operator== (const bitset<N>& rhs) const;
  bool operator!= (const bitset<N>& rhs) const;
  bool test (size_t pos) const;
  bool any() const;
  bool none() const;
  bitset<N> operator<< (size_t pos) const;
  bitset<N> operator>> (size_t pos) const;
```

```
private:
//implementation
};
// bitset 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>&);
template <size_t N>
bitset<N> operator^ (const bitset<N>&, const bitset<N>&);
template <size_t N>
istream& operator>> (istream& is, bitset<N>& x);
template <size_t N>
ostream& operator<< (ostream& os, const bitset<N>& x);
```

Description

bitset<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, then you will get an assertion failure instead of an exception.

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>.

```
explicit
bitset (const string& str, size_t pos = 0,
    size_t n = size_t(-1));
```

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 requires that pos <= str.length(), otherwise it throws an out_of_range exception.

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.

```
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.

```
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.

```
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`.

```
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`.

```
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 ~ ();
```

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 occurs on the input sequence
- The next character is neither '0' nor '1'. In this case, the character is not extracted.

```
template <size_t N>
ostream&
operator << (ostream& os, const bitset<N>& x);
```

Returns os << x.to_string()

Member functions

```
bool
any () const;
```

Returns true is 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) const;
```

Flips the bit at position pos in *this. Throws an out_of_range exception if pos does not correspond to a valid bit position.

```
bool
none () const;
```

Returns true is 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` is the bit at position `pos` is set. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
string
to_string() const;
```

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`.

compare

A binary function or a function object that returns true or false. "`compare`" is used for ordering elements.

complex

Complex number library

C++ complex number library.

Specializations

```
complex <float>
complex <double>
complex <long double>
```

Syntax

```
#include <complex>
template <class T>
class complex {
public:
   complex ();
   complex (T re);
   complex (T re , T im);
   template <class X> complex
    (const complex<X>&);
   T real () const;
   T imag () const;
   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>&);
};
// 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>
 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 X>
 istream& operator>>
  (istream&, complex<X>&);
template <class X>
 ostream& operator<<
  (ostream&, const complex<X>&);
// 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
 (T, T);

// Transcendentals
template<class T> complex<T> acos
 (const complex<T>&);
template<class T> complex<T> asin
 (const complex<T>&);
template<class T> complex<T> atan
 (const complex<T>&);
template<class T> complex<T> atan2
 (const complex<T>&, const complex<T>&);
template<class T> complex<T> atan2
 (const complex<T>&, T);
template<class T> complex<T> atan2
 (T, const complex<T>&);
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
 (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>&);
```

Description

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 exponentiation, logarithmic, power, and square root are also available.

Warning: On compilers that don't support member function templates, the arithmetic operators will not work on any arbitrary type. (They will work only on float, double and long doubles.) You also will only be able to perform binary arithmetic on types that are the same. Compilers that don't support non-converting constructors will permit unsafe downcasts (i.e., long double to double, double to float, long double to float).

Example

```
#include <complex>
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;
}
```

Constructors

```
complex
(const T& re_arg = 0, const T& im_arg = 0);
```

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>&);
```

Copy constructor. Constructs a complex number from another complex number.

Assignment operators

```
template <class X>
complex<T>
operator = (const complex<X>& c);
```

Assignment operator. 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.

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, T rhs);
```

```
template<class T> complex<T>
operator + (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, T rhs);
```

```
template<class T> complex<T>
operator - (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, T rhs);
```

```
template<class T> complex<T>
operator * (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, T rhs);
template<class T> complex<T>
operator / (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, 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 == (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, 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 != (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 X> istream&
operator >> (istream& is, complex<X>& 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, the `ios::badbit flag` is set.

```
template <class X> ostream&
operator << (ostream& os, const complex<X>& x);
```

Returns `os << "(" << x.real() << "," << x.imag() << ")"`.

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> complex<T>
acos (const complex<T>& c);
```

Returns the arccosine of `c`.

```
template<class T> T
arg (const complex<T>& c);
```

Returns the phase angle of `c`.

```
template<class T> complex<T>
asin (const complex<T>& c);
```

Returns the arcsine of `c`.

```
template<class T> complex<T>
atan (const complex<T>& c);
```

Returns the arctangent of `c`.

```
template<class T> complex<T>
atan2 (T a, const complex<T>& b);
```

Returns the arctangent of `a/b`.

```
template<class T> complex<T>
atan2 (const complex<T>& a, T b);
```

Returns the arctangent of `a/b`.

```
template<class T> complex<T>
atan2 (const complex<T>& a, const complex<T>& b);
```

Returns the arctangent of `a/b`.

```
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.

```
T
imag() const;
```

Returns the imaginary part of the complex number.

```
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 natural logarithm of x.

```
template<class T> complex<T>
log10 (const complex<T>& x);
```

Returns the logarithm base 10 of x.

```
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);
```

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>
pow (const complex<T>& x, T y);
```

```
template<class T> complex<T>
pow (const complex<T>& x, const complex<T>& y);
```

```
template<class T> complex<T>
pow (T x, const complex<T>& y);
```

Returns x raised to the y power.

```
T
real() const;
```

Returns the real part of the complex number.

```
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 square root of `x`.

```
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`.

construct

See also Memory handling primitive

Pointer based primitive for initializing memory.

Syntax

```
#include <memory>
template <class T1, class T2>
 void construct (T1 *p, const T2& value)
```

Description

The ***construct*** templated function initializes memory location `p` to `value`.

Containers

A standard template library (STL) collection.

Description

Within the standard template library, collection classes are often described as containers. A container stores a collection of other objects and provides certain basic functionality that supports the use of generic algorithms. Containers come in two basic flavors: 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***, and ***string*** fall into this category. *Associative containers* map values onto keys, which provides efficient retrieval of the values based on the keys. The STL provides 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 a value, base 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:

- A container allocates all storage for the objects it holds.
- A container `X` of objects of type `T` provides 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`. May be of any iterator type except output.
`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`

- A container provides a default constructor, a copy constructor, an assignment operator, and a full complement of comparison operators.
- A container provides the following member functions:

`begin()`	Returns an iterator pointing to the first element in the collection
`end()`	Returns an iterator pointing just beyond the last element in the collection
`swap(container)`	Swaps elements between this container and the swap's argument.
`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.

Reversible containers

A container may be reversible. Essentially, a reversible container provides 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 provides the following types:

`X::reverse_iterator`	An iterator type pointing to `T`
`X::const_reverse_iterator`	An iterator type pointing to `T`

- A reversible container provides the following member functions:

`rbegin()`	Returns a `reverse_iterator` pointing past the end of the collection
`rend()`	Returns a `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 provides the following constructors:

`X(n, t)`	Constructs a container with n elements t.
`X(i, j)`	Constructs a container with elements from the range [i,j).

- A sequence must provides 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 elements 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 provide 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()`
`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`

Associative containers

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 provides 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 provides the following additional constructors:

X(c)	Construct 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 provides 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.
a_uniq.insert(t)	Inserts t if and only if there is no element in the container with key equal to the key of 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(t)	Insert the element t and returns an iterator pointing to the newly inserted element.
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).
erase(k)	Erases all elements with key equal to k.
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 and 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 not less than 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)`.

copy, copy_backward

Algorithm

Copies a range of elements.

Syntax

```
#include <algorithm>
template <class InputIterator, class OutputIterator>
  OutputIterator
  copy(InputIterator first, InputIterator last,
       OutputIterator result);
template <class InputIterator, class OutputIterator>
  OutputIterator
  copy_backward(InputIterator first, InputIterator last,
                OutputIterator result);
```

Description

The ***copy*** algorithm copies values from the range specified by `[first ,last)` to the range that 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)`.

Both ***copy*** and ***copy_backward*** perform exactly `last - first` assignments.

Example

```
#include <algorithm>
#include <vector>
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> 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;
}
```

count, count_if

Algorithm

Count the number of elements in a container that satisfy a given condition.

Syntax

```
#include <algorithm>
template <class InputIterator, class T, class Size>
  void
  count(InputIterator first, InputIterator last,
        const T& value, Size& n);
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`, and increments a counting value `n` each time it finds a match. i.e., ***count*** adds to `n` the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
*i == value
```

The ***count_if*** algorithm lets you specify a predicate, and increments `n` each time an element in the sequence satisfies the predicate. That is, ***count_if*** adds to `n` the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
pred(*i) == true.
```

Both ***count*** and ***count_if*** perform exactly `last-first` applications of the corresponding predicate.

Example

```
#include <vector>
#include <algorithm>
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
  // i = 2, j = 0

  // Count all less than 8
  count_if(v.begin(),v.end(),bind2nd(less<int>(),8),k);
  // k = 7
  cout << i << " " << j << " " << k << endl;
  return 0;
}
```

deallocate

See also Memory handling primitive

A pointer based primitive for handling memory.

Syntax

```
#include <memory>
template <class T>
 void deallocate (T* buffer);
```

Description

The ***deallocate*** templated function frees the memory used by `buffer` for system-wide use.

deque

Container

A sequence that supports random access iterators and efficient insertion/deletion at both beginning and end.

Syntax

```
#include <deque>
template <class T>
 class deque {

public:
 // Types
   typedef typename reference;
   typedef typename const_reference;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef typename difference_type;
   typedef T value_type;
   typedef reverse_iterator<iterator,
           value_type, reference,
           difference_type> reverse_iterator;
   typedef reverse_iterator<const_iterator,
           value_type, const_reference,
           difference_type> const_reverse_iterator;

 // Construct/Copy/Destroy
   explicit deque ();
   explicit deque (size_type, const T& = T());
   deque (const deque<T>&);
   template <class InputIterator>
    deque (InputIterator, InputIterator);
   ~deque ();
   deque<T> operator= (const deque<T>);
   template <class InputIterator>
    assign (InputIterator, InputIterator);
   template <class Size, class T>
    void assign (Size n, const T& t = T());

 // 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,T c = T());
   bool empty () const;

// Element access
   reference operator[] (size_type);
   const_reference operator[] (size_type) const;
```

```
   reference at (size_type n);
   const_reference at (size_type n) 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& = T());
   void insert (iterator, size_type, const T& = T());
   template <class InputIterator>
    void insert (iterator, InputIterator, InputIterator);
   void pop_front ();
   void pop_back ();
   void erase (iterator);
   void erase (iterator, iterator);
   void swap (deque<T>&);
};
 // Comparison
template <class T>
 bool operator== (const deque<T>&, const deque <T>&);
template <class T>
 bool operator< (const deque<T>, const deque <T>&);
```

Description

deque<T> 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 automatically.

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`):

Default constructor	`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`

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this is the constructor for ***deque<T>*** that takes two templated 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 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 ***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.

Example

```
#include <deque>
#include <string>
using namespace std;
deque<string> deck_of_cards;
deque<string> current_hand;
void initialize_cards(deque<string>& cards) {
  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;
}
```

Constructors and destructors

```
explicit
deque ();
```

The default constructor. Creates a deque of zero elements.

```
explicit
```

```
deque (size_type n, const T& value = T());
```

Creates a deque of length `n`, containing `n` copies of `value`.

```
deque (const deque<T>& x);
```

Copy constructor. Creates a copy of x.

```
template <class InputIterator>
deque (InputIterator first, InputIterator last);
```

Creates a deque of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`.

```
~deque ();
```

The destructor. Releases any allocated memory for self.

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 iterator that points to the past-the-end value.

```
const_reverse_iterator rbegin () const;
```

Returns a constant random access iterator that points to the past-the-end value.

```
reverse_iterator rend ();
```

Returns a random access iterator that points to the first element.

```
const_reverse_iterator rend () const;
```

Returns a constant random access iterator that points to the first element.

Assignment operator

```
deque<T>&
operator= (const deque<T>& x);
```

Assignment operator. 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 less one.

```
const_reference operator[] (size_type) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the size less one.

Comparison operators

```
template <class T>
bool
operator== (const deque<T>& x, const deque T>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T>
bool
operator< (const deque<T>& x, const deque T>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

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)`.

```
template <class Size, class T>
void
assign (Size n, const T& 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.

```
bool
empty () const;
```

Returns true if the size of self is zero.

```
void
erase (iterator position);
```

Removes the element pointed to by `position`.

```
void
erase (iterator first, iterator last);
```

Removes the elements in the range `[first, last)`.

```
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 = T());
```

Inserts x before position. The return value points to the inserted x.

```
void
insert (iterator position, size_type n, const T& x =T());
```

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

```
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, T c = T());
```

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. If sz is equal to capacity, no action is taken.

```
size_type
size () const;
```

Returns the number of elements.

```
void
swap (deque<T>& x);
```

Exchanges self with x.

destroy

See also		Memory handling primitive

Invoke the destructor for values pointed to by iterators or pointers.

Syntax

```
#include <memory>
template <class ForwardIterator>
void destroy (ForwardIterator first, ForwardIterator last)

template <class T>
void destroy (T* pointer)
```

Description

```
template <class T>
void
destroy (T* pointer)
```

Invokes the destructor for the value pointed to by the argument pointer.

```
template <class ForwardIterator>
void
destroy (ForwardIterator first, ForwardIterator last)
```

Destroys all of the values in the range `[first, last)`.

distance

See also   Iterator operation

Computes the distance between two iterators.

Syntax

```
#include <iterator>
template <class InputIterator, class Distance>
 void distance (InputIterator first,
                InputIterator last,
                Distance& n);
```

Description

The ***distance*** template function computes the distance between two iterators and stores that value in `n`. The last iterator must be reachable from the first iterator.

distance increments `n` by the number of times it takes to get from `first` to `last`. ***distance*** must be a three argument function that stores the result into a reference instead of returning the result, because the distance type cannot be deduced from built-in iterator types such as `int*`.

Example

```
#include<iterator>
#include<vector>
using namespace std;
int main()
{
  //Initialize a vector using an array
  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>(cout," "));
  cout << endl << endl;
  cout << "When the iterator is initialized to point to "
       << *itr << endl << 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;
}
```

divides

See also		Function object

Returns the result of dividing its first argument by its second.

Syntax

```
#include <functional>
template <class T>
   struct divides : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const
              { return x / y; }
   };
```

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(), vec2.end(),
            vecResult.begin(), divides<int>());
```

After this call to ***transform***, `vecResult[n]` will contain `vec1[n]` divided by `vec2[n]`.

equal

Algorithm

Compares two ranges for equivalence.

Syntax

```
#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`.

This algorithm assumes that there are at least as many elements available after `first2` as there are in the range `[first1, last1)`.

equal performs at most `last1-first1` comparisons or applications of the predicate.

Example

```
#include <algorithm>
#include <vector>
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,d1 + 4), v2(d2,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;
}
```

equal_range

Algorithm

Determines the valid range for insertion of a value in a container.

Syntax

```
#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 provides 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`, i.e., 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` will point to the first element not "less" than value, and `j` will point 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
```

equal_range performs at most `2 * log(last - first) + 1` comparisons.

Example

```
#include <vector>
#include <algorithm>
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 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;
}
```

equal_to

See also Function object

Binary function object that returns `true` if its first argument equals its second.

Syntax

```
#include <functional>
    template <class T>
    struct equal_to : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x == y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was equal to `vec2(n)` or a "0" if `vec1(n)` was not equal to `vec2(n)`.

exception

standard exception

Classes supporting logic and runtime errors.

Syntax

```
#include <stdexcept>
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:
  logic_error (const string& what_arg);
};
class domain_error : public logic_error {
 public:
  domain_error (const string& what_arg);
};
class invalid_argument : public logic_error {
 public:
  invalid_argument (const string& what_arg);
};
class length_error : public logic_error {
 public:
  length_error (const string& what_arg);
};
class out_of_range : public logic_error {
 public:
  out_of_range (const string& what_arg);
};
class runtime_error : public exception {
 public:
  runtime_error (const string& what_arg);
};
class range_error : public runtime_error {
 public:
  range_error (const string& what_arg);
};
class overflow_error : public runtime_error {
 public:
  overflow_error (const string& what_arg);
};
```

Description

The class exception defines the base class for the types of objects thrown as exceptions by Standard C++ Library components, and certain expressions, to report errors detected during program execution. User's can also use these exceptions to report errors in their own programs.

Constructors

```
exception () throw();
```

Constructs an object of class exception.

```
exception (const exception&) throw();
```

The copy constructor. Copies an exception object.

Destructor

```
virtual
~exception() throw();
```

Destroys an object of class exception.

Operators

```
exception&
operator= (const exception&) throw();
```

The assignment operator. Copies an exception object.

Member function

```
virtual const char*
what()const 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`.

Example

```
#include <iostream.h>
 #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)
  {
    cout << "Got an exception: " << e.what() << endl;
  }
  return 0;
}
```

fill, fill_n

Algorithm

Initializes a range with a given value.

Syntax

```
#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.

fill makes exactly `last - first` assignments, and ***fill_n*** makes exactly `n` assignments.

Example

```
#include <algorithm>
#include <vector>
using namespace std;
int main()
{
  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> 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>(cout," "),3,5);
  cout << endl;
  return 0;
}
```

find

See also Algorithm

Find an occurrence of value in a sequence.

Syntax

```
#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`.

find performs at most `last-first` comparisons.

Example

```
#include <vector>
#include <algorithm>
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);
  // 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;
}
```

find_first_of

See also Algorithm

Finds a match for a given subsequence within a sequence.

Syntax

```
#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 a subsequence, specified by `first2,last2`, in a sequence specified by `first1, last1`. 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. The algorithm returns an iterator in the range `[first1,last1)` that points to the first element of the matching subsequence . If the subsequence is not located in the sequence, ***find_first_of*** returns `last1`.

In other words, ***find_first_of*** returns an iterator `i` in the range `[first1,last1)` such that for some `j` in the range `[first2,last2)`:

```
*i ==*(first2 + n)
```

or

```
pred(i, first2+n)==true
```

find_first_of performs at most `(last1 - first1)` applications of the corresponding predicate.

Example

```
#include <vector>
#include <iterator>
#include <algorithm>
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>(cout," " ));
  cout << " and ";
  copy(v2.begin(),v2.end(),
       ostream_iterator<int>(cout," " ));
  cout << endl << endl
      << "both versions of find_first_of point to: "
      << *it1;

  return 0;
}
```

find_if

See also Algorithm

Finds an occurrence of value in a sequence.

Syntax

```
#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 ***find_if***'s 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`.

find_if performs at most `last-first` applications of the corresponding predicate.

Example

```
#include <vector>
#include <algorithm>
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);
  // 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;
}
```

for_each

Algorithm

Applies a function to each element in a range.

Syntax

```
#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.

The function `f` is applied exactly `last - first` times.

Example

```
#include <vector>
#include <algorithm>
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;
  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;
}
```

forward iterator

See also Iterator

A forward-moving iterator that can both read and write.

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.

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 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 convertable to `bool`
`*a`	return value convertable to `T&`
`++r`	returns `X&`
`r++`	return value convertable to const `X&`
`*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.

adjacent_find, find, find_first_of

front_insert_iterator, front_inserter

See also Insert iterator

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

Syntax

```
#include <iterator>
template <class Container>
 class front_insert_iterator : public output_iterator {
protected:
   Container& container;
public:
   front_insert_iterator (Container& x);
   front_insert_iterator<Container>&
   operator= (const Container::value_type& value);
   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& x)
```

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 ***vector***s, ***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 will find, after the three insertions, that the first three elements of `exmpl` are "3 2 1".

Example

```
/***************************************************************
 *
 * ins_itr.cpp - Example program of insert iterator.
 *
 * $Id: ins_itr.cpp,v 1.7 1995/10/06 18:18:03 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <iterator>
 #include <deque>
 using namespace std;
 int main ()
 {
  //
  // 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;
 }
```

Constructor

```
front_insert_iterator (Container& x);
```

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

Operators

```
front_insert_iterator<Container>&
operator = (const 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`.

Helper function

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

Returns a ***front_insert_iterator*** that will insert elements at the beginning of container `x`. This function allows you to create front insert iterators inline.

function object

Objects with an operator() defined. Function objects are used in place of pointers to functions as arguments to templated algorithms.

Syntax

```
#include<functional>
// typedefs
   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& x, const T& y) const
             { return x + y; }
  };
  template <class T>
  struct minus : binary_function<T, T, T> {
       T operator() (const T& x, const T& y) const
             { return x - y; }
  };
  template <class T>
  struct times : binary_function<T, T, T> {
       T operator() (const T& x, const T& y) const
             { return x * y; }
  };
  template <class T>
  struct divides : binary_function<T, T, T> {
       T operator() (const T& x, const T& y) const
             { return x / y; }
  };
  template <class T>
  struct modulus : binary_function<T, T, T> {
       T operator() (const T& x, const T& y) const
             { return x % y; }
  };
  template <class T>
  struct negate : unary_function<T, T, T> {
       T operator() (const T& x) const
             { return -x; }
  };
// Comparisons
   template <class T>
   struct equal_to : binary_function<T, T, bool> {
```

```
        bool operator() (const T& x, const T& y) const
              { return x == y; }
   };
   template <class T>
   struct not_equal_to : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x != y; }
   };
   template <class T>
   PD 0 struct greater : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x > y; }
   };
   template <class T>
   struct less : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x < y; }
   };
   template <class T>
   struct greater_equal : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x >= y; }
   };
   template <class T>
   struct less_equal : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x <= y; }
   };
// Logical Comparisons
   template <class T>
   struct logical_and : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x && y; }
   };
   template <class T>
   struct logical_or : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return x || y; }
   };
   template <class T>
   struct logical_not : unary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const
              { return !x; }
   };
```

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 provides 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 provide the typedefs `argument_type` and `result_type`. Similarly, function objects that

take two arguments are called *binary function objects* and, as such, are required to provide the typedefs `first_argument_type`, `second_argument_type`, and `result_type`.

The classes `unary_function` and `binary_function` make the task of creating templated function objects easier. The necessary typedefs for a unary or binary function object are provided by inheriting from the appropriate function object class.

The function objects provided by 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`
`times`	multiplication `x * y`
`divides`	division `x / y`
`modulus`	remainder `x % y`
`negate`	negation `- x`
Comparison functions	
`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`

Example

```
#include<functional>
#include<deque>
#include<vector>
#include<algorithm>

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>());

  //Print the results
  cout << "The following numbers: " << endl << "     ";
  copy(d.begin(),d.end(),ostream_iterator<int>(cout," "));

  cout << endl << endl;
  cout << "Have the factorials: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));

  return 0;
}
```

generate, generate_n

Algorithm

Initialize a container with values produced by a value-generator class.

Syntax

```
#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.

The ***generate*** and ***generate_n*** algorithms invoke `gen` and assign its return value exactly `last – first` (or `n`) times.

Example

```
#include <algorithm>
#include <vector>
using namespace std;
// 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> 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;
}
```

get_temporary_buffer

See also   Memory handling primitive

Pointer based primitive for handling memory.

Syntax

```
#include <memory>
template <class T>
 pair<T*, ptrdiff_t> get_temporary_buffer (ptrdiff_t n, T*);
```

Description

The ***get_temporary_buffer*** templated 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)`.

greater

See also Function object

Binary function object that returns `true` if its first argument is greater than its second.

Syntax

```
#include <functional>
    template <class T>
         struct greater : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
                { return x > y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), greater<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was greater than `vec2(n)` or a "0" if `vec1(n)` was less than or equal to `vec2(n)`.

greater_equal

See also   Function object

Binary function object that returns `true` if its first argument is greater than or equal to its second.

Syntax

```
#include <functional>
    template <class T>
    struct greater_equal : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x >= y; }
    };
```

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` will be sorted in descending order.

Heap operations

See also Algorithm

See the entries for ***make_heap***, ***pop_heap***, ***push_heap*** and ***sort_heap.***

includes

See also Algorithm

Basic set operation for sorted sequences.

Syntax

```
#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 default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
#include<algorithm>
#include<set>
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};
  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>(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>(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>(cout," "));
 cout << endl << endl;
 return 0;
}
```

inner_product

Generalized numeric operation

Computes the inner product `A X B` of two ranges `A` and `B`.

Syntax

```
#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 BinaryOperation last1>
T inner_product (InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2,
                T init,
                BinaryOperation1 binary_op1,
                BinaryOperation 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))` in order. The algorithm returns `acc`.

The second version of the function initializes `acc` with `init`, then computes the result:

```
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))` in order.

The ***inner_product*** algorithm computes exactly `(last1 - first1)` applications of either

```
acc + (*i1) * (*i2)
```

or

```
binary_op1(acc, binary_op2(*i1, *i2)).
```

Example

```
#include <numeric>       //For inner_product
#include <list>          //For list
#include <vector>        //For vectors
#include <functional>    //For plus and minus
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>(cout," "));
  cout << endl << " and  ";
  copy(l.begin(),l.end(),ostream_iterator<int>(cout," "));
  cout << "," << endl << endl;
  cout << "The inner product is: " << inner_prod << endl;
  cout << "The wacky result is: " << wacky << endl;
  return 0;
}
```

inplace_merge

See also Algorithm

Merge two sorted sequences into one.

Syntax

```
#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, that is, 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 as the default for comparison, and the second version accepts a third argument that specifies a comparison operator.

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 may be used.

Example

```
#include <algorithm>
 #include <vector>
 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> 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>(cout," "));
  cout << endl;
  return 0;
}
```

input iterator

See also Iterator

A read-only, forward moving iterator.

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.

The following key pertains to the iterator requirement descriptions 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 input iterators

The following expressions must be valid for input iterators:

`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 convertable to `bool`
`*a`	`a == b` implies `*a == *b`
`++r`	returns `X&`
`r++`	return `value` convertable 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. That is they should not pass through the same iterator twice.

The value of type `T` does not have to be an `lvalue`.

insert_iterator, inserter

Insert iterator

An insert iterator used to insert items into a collection rather than overwrite the collection.

Syntax

```
#include <iterator>
template <class Container>
 class insert_iterator : public output_iterator {
protected:
    Container& container;
    Container::iterator iter;
public:
   insert_iterator (Container& x, Container::iterator i);
   insert_iterator<Container>&
   operator= (const Container::value_type& value);
   insert_iterator<Container>& operator* ();
   insert_iterator<Container>& operator++ ();
   insert_iterator<Container> operator++ (int);
};
template <class Container>
insert_iterator<Container> inserter (Container& x, Iterator i)
```

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.

Example

```
 #include<iterator>
 #include<vector>
 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(),ostream_iterator<int>(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>(cout," "));
   return 0;
 }
```

Constructor

```
insert_iterator (Container& x, Container::iterator i);
```

Constructor. Creates an instance of an ***insert_iterator*** associated with container `x` and iterator `i`.

Operators

```
insert_iterator<Container>&
operator= (const 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`.

Helper function

```
template <class Container, class Iterator>
insert_iterator<Container>
inserter (Container& x, Iterator i)
```

Returns an ***insert_iterator*** that will insert elements into container `x` at location `i`. This function allows you to create insert iterators inline.

Insert iterator

See also Insert iterator

Iterator adaptor that allows an iterator to insert into a container rather than overwrite elements in the container.

Syntax

```
#include <iterator>
template <class Container>
 class back_insert_iterator : public output_iterator {
protected:
   Container& container;
public:
   back_insert_iterator (Container& x);
   back_insert_iterator<Container>&
   operator= (const Container::value_type& value);
   back_insert_iterator<Container>& operator* ();
   back_insert_iterator<Container>& operator++ ();
   back_insert_iterator<Container> operator++ (int);
};
template <class Container>
 class front_insert_iterator : public output_iterator {
protected:
   Container& container;
public:
   front_insert_iterator (Container& x);
   front_insert_iterator<Container>&
   operator= (const Container::value_type& value);
   front_insert_iterator<Container>& operator* ();
   front_insert_iterator<Container>& operator++ ();
   front_insert_iterator<Container> operator++ (int);
};
template <class Container>
 class insert_iterator : public output_iterator {
protected:
    Container& container;
    Container::iterator iter;
public:
   insert_iterator (Container& x, Container::iterator i);
   insert_iterator<Container>&
   operator= (const Container::value_type& value);
   insert_iterator<Container>& operator* ();
insert_iterator<Container>& operator++ ();
   insert_iterator<Container> operator++ (int);
};
 template <class Container>
 back_insert_iterator<Container> back_inserter (Container& x)
 template <class Container>
 front_insert_iterator<Container> front_inserter (Container& x)
 template <class Container>
 insert_iterator<Container> inserter (Container& x, Iterator i)
```

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***.

istream_iterator

See also Iterators

Stream iterator that provides iterator capabilities for istreams. This iterator allows generic algorithms to be used directly on streams.

Syntax

```
#include <iterator>
Istream iterator
template class T, class Distance = ptrdiff_t>
 class istream_iterator : public input_iterator<T, Distance>
 {
 public:
  istream_iterator();
  istream_iterator (istream& s);
  istream_iterator (const istream_iterator <T, Distance>& x);
  ~istream_itertor ();
  const T& operator*() const;
  istream_iterator <T, Distance>& operator++();
  istream_iterator <T, Distance> operator++ (int)
 };
```

Description

Stream iterators provide the standard iterator interface for input and output streams.

The class ***istream_iterator*** reads elements from an input stream. A value of `T` is retrieved and stored when the iterator is constructed and each time `operator++` is called. The iterator will be equal to the end-of-stream iterator value if the end-of-file is reached. 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 dereference the end-of-stream iterator; it plays the same role as the past-the-end iterator provided by 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).

Constructors

`istream_iterator ();`

Construct an end-of-stream iterator. This iterator can be used to compare against and end-of-file condition. Use it to provide end iterators to algorithms.

`istream_iterator (istream& s);`

Construct an ***istream_iterator*** on the given stream.

`istream_iterator (const istream_iterator<T, Distance>& x);`

Copy constructor.

Destructors

`~istream_iterator ();`

Destructor.

Operators

`const T& operator* () const;`

Return the current value stored by the iterator.

```
istream_iterator<T, Distance>&
operator++ ()

istream_iterator<T, Distance>
operator++ (int)
```

Retrieve the next element from the input stream.

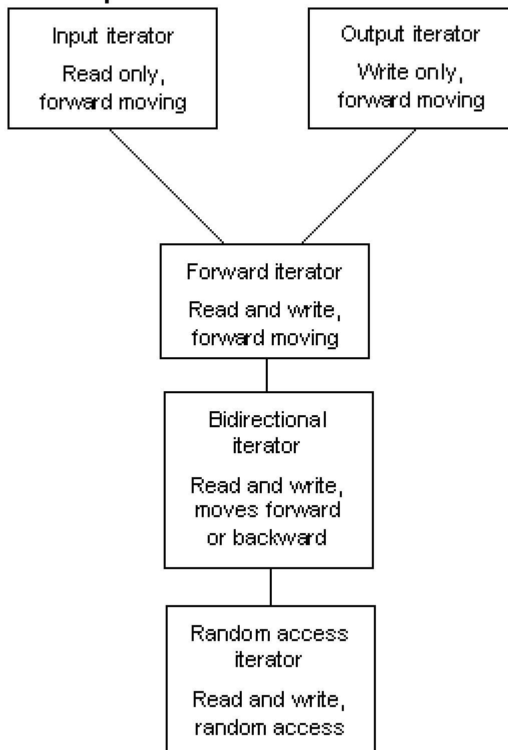
Example

```
#include <iterator>
#include <vector>
#include <numeric>
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,vector<int>::difference_type>(cin),
   istream_iterator<int,vector<int>::difference_type>(),
   inserter(d,d.begin()));
 //
 // stream the whole vector and the sum to cout
 //
 copy(d.begin(),d.end()-1,ostream_iterator<int>(cout," + "));
 if (d.size())
  cout << *(d.end()-1) << " = " <<
    accumulate(d.begin(),d.end(),total) << endl;
 return 0;
}
```

Iterators

Pointer generalizations for traversal and modification of collections.

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 what type of iterator it generates. And, 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 provide their own iterators. An instance of a container or sequence's iterator may be declared using either of the following:

```
class name ::iterator
class name ::const_iterator
```

Containers and sequences must also provide `const` iterators to the beginning and end of their collections. These may be accessed using the class members, `begin()` and `end()`.

The semantics of iterators are a generalization of the semantics of C++ pointers. Every template function that takes iterators will work using C++ pointers for processing typed contiguous memory sequences.

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 dereferencable.

Every function provided by an iterator is required to be realized in amortized constant time.

The following key pertains to all of the iterator requirement descriptions in this section:

`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 input iterators

The following expressions must be valid for input iterators:

`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 convertable to `bool`
`*a`	`a == b` implies `*a == *b`
`++r`	returns `X&`
`r++`	return `value` convertable 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. That is 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 convertable to `const X&`
`*r++ = t`	

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.

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 convertable to `bool`
`*a`	return value convertable to `T&`
`++r`	returns `X&`
`r++`	return value convertable to const `X&`
`*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` convertable 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 convertable 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 convertable to `bool`.

iter_swap

See also Algorithm

Exchange values pointed at in two locations.

Syntax

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
 void iter_swap (ForwardIterator1 a, ForwardIterator2 b);
```

Description

The ***iter_swap*** algorithm exchanges the values pointed at by the two iterators `a` and `b`.

Example

```
/****************************************************************
 *
 * swap.cpp - Example program of swap algorithm.
 *
 * $Id: swap.cpp,v 1.7 1995/10/06 20:05:43 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ****************************************************************/
 #include <vector>
 #include <algorithm>
 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>(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
    << "Swaping the first 5 elements with the last 5 gives: "
    << endl << "    ";
  copy(v.begin(), v.end(), ostream_iterator<int>(cout," "));
  //
  // Now an example of iter_swap -- swap first and last elements.
  //
  iter_swap(v.begin(), v.end()-1);
  //
  // Output result.
  //
```

```
 cout << endl << endl
   << "Swaping the first and last elements gives: "
   << endl << "    ";
 copy(v.begin(), v.end(), ostream_iterator<int>(cout," "));
 cout << endl;

 return 0;
}
```

less

See also Function object

Binary function object that returns `true` if its first argument is less than its second.

Syntax

```
#include<functional>
    template <class T>
         struct less : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x < y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), less<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was less than `vec2(n)` or a "0" if `vec1(n)` was greater than or equal to `vec2(n)`.

less_equal

See also Function object

Binary function object that returns `true` if its first argument is less than or equal to its second.

Syntax

```
#include<functional>
    template <class T>
    struct less_equal : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x <= y; }
    };
```

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(),greater_equal<int>());
```

After this call to ***sort***, `vec1` will be sorted in ascending order.

lexicographical_compare

Algorithm

Compares two ranges lexicographically.

Syntax

```
#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 "<" 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 used in place of the default "<" operator.

The ***lexicographic_compare*** functions can be used with all the datatypes provided by the standard library.

lexicographical_compare performs at most `min((last1 - first1), (last2 - first2))` applications of the comparison function.

Example

```
#include <algorithm>
#include <vector>
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;
}
```

limits

Numeric limits library

Refer to the ***numeric_limits*** section of this reference guide.

list

See also Container

A sequence that supports bidirectional iterators.

Syntax

```
#include <list>
template <class T>
 class list {
public:
// typedefs
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename reference;
   typedef typename const_reference;
   typedef typename size_type;
   typedef typename difference_type;
   typedef T value_type;
   typedef  reverse_iterator<iterator, value_type,
            reference, difference_type> reverse_iterator;
   typedef  const_reverse_iterator<const_iterator,
            value_type, reference,
            difference_type> const_reverse_iterator;
// Construct/Copy/Destroy
   explicit list ();
   explicit list (size_type n, const T& value = T());
   template <class InputIterator>
    list (InputIterator first, InputIterator last);
   list(const list<T>& x);
   ~list();
   list<T>& operator= (const list<T>&);
   template <class InputIterator>
    void assign (InputIterator first, InputIterator last);
   template <class Size, class T>
    void assign (Size n, const T& t = T());
// 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 sz, T c = T());
// Element Access
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;
```

```
// Modifiers
   void push_front (const T& x);
   void pop_front ();
   void push_back (const T& x);
   void pop_back ();

   iterator insert (iterator position, const T& x = T());
   void insert (iterator position, size_type n, const T& x =
   T());
   template <class InputIterator>
    void insert  (iterator  position,  InputIterator  first,
   InputIterator last);

   void erase (iterator position);
   void erase (iterator position, iterator last);
   void swap (list<T>& x);

// Special mutative operations on list
   void splice (iterator position, list<T>& x);
   void splice (iterator position, list<T>& x, iterator i);
   void splice (iterator position, list<T>& x,
               iterator first,iterator last);

   void remove (const T& value);
   template <class Predicate>
    void remove_if (Predicate pred);

   void unique ();

   template <class BinaryPredicate>
    void unique (BinaryPredicate binary_pred);

   void merge (list<T>& x);
   template <class Compare>
    void merge (list<T>& x, Compare comp);

   void sort ();
   template <class Compare>
    void sort (Compare comp);

   void reverse();
};
// Non-member List Operators
template <class T>
 bool operator== (const list<T>&, const list<T>&);

template <class T>
 bool operator< (const list<T>&, const list<T>&);
```

Description

list<T> is a type of sequence that supports bidirectional iterators. A ***list<T>*** 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 provide the following (where `T` is the type, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
Default constructor   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
```

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for ***list<T>*** that takes two templated iterators:

```
template <class InputIterator>
 list (InputIterator, InputIterator);
```

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, 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 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.

Example

```
#include <list>
#include <string>
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>(cout," "));
  return out;
}
int main(void)
{
  // create a list of critters
  list<string> critters;
  int i;
  // 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;
}
```

Constructors and destructors

`explicit list();`

Creates a list of zero elements.

`explicit list (size_type n, const T& value = T());`

Creates a list of length `n`, containing `n` copies of `value`.

```
template <class InputIterator>
list (InputIterator first, InputIterator last);
```

Creates a list of length `last - first`, filled with all values obtained by dereferencing the `InputIterator` s on the range `[first, last)`.

list (const list<T>& x);

Copy constructor. Creates a copy of `x`.

`~list ();`

The destructor. Releases any allocated memory for this list.

Assignment operator

`list<T>& operator= (const list<T>& x)`

Assignment operator. Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to `*this`.

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.

`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)`.

```
template <class Size, class T>
void
assign (Size n, const T& 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.

```
bool
empty () const;
```

Returns `true` if the `size` is zero.

```
void
erase (iterator position);
```

Removes the element pointed to by `position`.

```
void
erase (iterator first, iterator last);
```

Removes the elements in the range `[first, last)`.

```
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 = T());
```

Inserts `x` before `position`. Returns an iterator that points to the inserted `x`.

```
void
insert (iterator position, size_type n, const  T& x = T());
```

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`.

```
max_size () const;
```

Returns `size()` of the largest possible list.

```
void merge (list<T>& x);
```

Merges a sorted `x` with a sorted self using operator<. For equal elements in the two lists, elements from self will always precede the elements from `x`. The `merge` function leaves `x` empty.

```
template <class Compare>
void
merge (list<T>& x, Compare comp);
```

Merges a sorted `x` with sorted self using a compare function object, `comp`. For same elements in the two lists, elements from self will 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
void push_back (const T& x);
```

Appends a copy of `x` to the end.

```
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);
```

Removes all elements in the list referred 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, T c = T());
```

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. If `sz` is equal to capacity 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>& x);
```

Inserts **x** before position leaving **x** empty.

```
void
splice (iterator position, list<T>&  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>&  x,  iterator first, iterator last);
```

Moves the elements in the range `[first, last)` in `x` to self, inserting before `position`. The elements in the range `[first, last)` are removed from `x`.

```
void
swap (list<T>& 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);
```

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>& x, const list <T>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool
operator< (const list<T>& x, const list <T>& 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`.

logical_and

See also Function object

Binary function object that returns `true` if both of its arguments are `true`.

Syntax

```
#include <functional>
    template <class T>
    struct logical_and : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x && y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), logical_and<bool>());
```

After this call to ***transform***, `vecResult(n)` will contain 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`.

logical_not

See also Function object

Unary function object that returns `true` if its argument is `false`.

Syntax

```
#include<functional>
    template <class T>
    struct logical_not : unary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return !x; }
    };
```

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".

logical_or

See also   Function object

Binary function object that returns `true` if either of its arguments are `true`.

Syntax

```
#include<functional>
    template <class T>
    struct logical_or : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
               { return x || y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), logical_or<bool>());
```

After this call to ***transform***, `vecResult(n)` will contain 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`.

lower_bound

See also   Algorithm

Determine the first valid position for an element in a sorted container.

Syntax

```
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
```

lower_bound performs at most `log(last - first) + 1` comparisons.

Example

```
#include <vector>
#include <algorithm>
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;
}
```

make_heap

See also Algorithm

Creates a heap.

Syntax

```
#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 the 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 the 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)`.

This algorithm makes at most `3 * (last - first)` comparisons.

Example

```
#include <algorithm>
#include <vector>
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> 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;
}
```

map

Container

An associative container providing access to non-key values using unique keys.

Syntax

```
#include <map>
template <class Key, class T, class Compare = less<Key> >
 class map {
public:
// types
   typedef Key key_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef typename reference;
   typedef typename const_reference;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef typename difference_type;
   typedef reverse_iterator<iterator,
                  value_type,
                  reference,
                  difference_type> reverse_iterator;
   typedef reverse_iterator<const_iterator,
                  value_type,
                  const_reference,
                  difference_type> const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type, bool>
   {
     friend class map;
     protected :
       Compare comp;
       value_compare(Compare c) : comp(c) {}
     public :
       bool operator() (const value_type& x,
                        const value_type& y) const
         { return comp(x.first, y.first); }
   };
// Construct/Copy/Destroy
   explicit map (const Compare& comp = Compare());
   template <class InputIterator>
    map (InputIterator first,

        InputIterator last,
        const Compare& comp = Compare());
   map (const map<Key, T, Compare>& x);
   ~map();
   map<Key, Compare>& operator= (const map<Key, T, Compare>&
   x);
// 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
    T& operator[] (const key_type& x);
    const T& operator[] (const key_type& x) const;
// Modifiers
    pair<iterator, bool> insert (const value_type& x);
    iterator insert (iterator position, const value_type& x);
    template <class InputIterator>
     void insert (InputIterator first, InputIterator last);
    void erase (iterator position);
    size_type erase (const key_type& x);
    void erase (iterator first, iterator last);
    void swap (map<Key, Compare>& x);
// Observers
    key_compare key_comp() const;
    value_compare value_comp() const;
// Map operations
    iterator find (const key_value& x);
    const_iterator find (const key_value& x) const;
    size_type count (const key_type& x) const;
    iterator lower_bound (const key_type& x);
    const_iterator lower_bound (const key_type& x) const;
    iterator upper_bound (const key_type& x);
    const_iterator upper_bound (const key_type& x) const;
    pair<iterator, iterator> equal_range (const key_type& x);
    pair<const_iterator, const_iterator>
      equal_range (const key_type& x) const;
};
template <class Key,
          class T,
          class Compare>
 bool operator== (const map<Key, T, Compare>& x,
                  const map<Key, T, Compare>& y);
template <class Key,
          class T,
          class Compare>
 bool operator< (const map<Key, T, Compare>& x,
                 const map<Key, T, Compare>& y);
```

Description

map provides fast access to stored values of type `T` which are indexed by unique keys of a separate type. The default operation for key comparison is the `<` operator.

map provides 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*** provides a

typedef to this pair called value_type.

The types used for both the template parameters Key and 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
```

The type used for the Compare template parameter must satisfy the requirements for binary functions.

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for map<Key,Compare> that takes two templated iterators:

```
template <class InputIterator>
 map (InputIterator, InputIterator, Compare);
```

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 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 ***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, 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 need to always supply the Compare template argument.

Example

```
#include <string>
#include <map>
using namespace std;
typedef map<string, int, less<string> > months_type;
// 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>(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 Febuary 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;
}
```

Constructors and destructors

```
explicit map (const Compare& comp = Compare());
```

Default constructor. Constructs an empty map that will use the relation `Compare` to order keys, if it is supplied.

```
template <class InputIterator>
map (InputIterator first, InputIterator last,
     const Compare& comp = Compare());
```

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.

```
map (const map<Key, T, Compare>& x);
```

Copy constructor. Creates a new map by copying all pairs of `key` and `value` from `x`.

```
~map ();
```

The destructor. Releases any allocated memory for this map.

Iterators

`iterator` **`begin`**`() ;`

Returns a 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 a iterator pointing to the last element stored in the map, i.e., the off-the-end value.

`const_iterator` **`end`**`() const;`

Returns a `const_iterator` pointing to the last element stored in the map.

`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, i.e., 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>&
operator= (const map<Key, T, Compare>& x);
```

Assignment. Replaces the contents of `*this` with a copy of the map `x`.

`T& operator[] (const key_type& x);`

If an element with the key `x` exists in the map, then a reference to its associated value will be returned. Otherwise the pair `x,T()` will be inserted into the map and a reference to the default object `T()` will be returned.

Member functions

```
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` is 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);
```

Erases the map element pointed to by the iterator `position`.

size_type
erase (const key_type& x);

Erases the element with the key value x from the map, if one exists.

void
erase (iterator first, iterator last);

Providing the iterators first and last point to the same map and last is reachable from first, all elements in the range [first, last) will be erased from the map.

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. A position may be supplied as a hint regarding where to do the insertion. If the insertion may be done right after position then it takes amortized constant time. Otherwise it will take O(log N) time.

template <class InputIterator>
void
insert (InputIterator first, InputIterator last);

Copies of each element in the range [first, last) which possess a unique key, one not already in the map, will be 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 an iterator to the smallest map element whose key is greater 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 map. This size is only constrained by the number of unique keys which 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>& x);

Swaps the contents of the map x with the current map, *this.

```
iterator
upper_bound (const key_type& x)
```

Returns an iterator to the largest map element whose key is smaller 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 key values using the comparison operation, `Compare`, of the current map. This function is identical to `key_comp` for sets.

Non-member operators

```
bool operator== (const map<Key, T, Compare>& x,
const map<Key, T, Compare>& 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 map<Key, T, Compare>& x,
                const map<Key, T, Compare>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

max

See also Algorithm

Find and return the maximum of a pair of values.

Syntax

```
#include <algorithm>
template <class T>
 const T& max(const T& a, const T& b);
template <class T, class Compare>
 const T& max(const T& a, const T& b, Compare comp);
```

Description

The ***max*** algorithm determines and returns the maximum of a pair of values. The optional argument `comp` defines a comparison function that can be used in place of the default "<" operator. This function can be used with all the datatypes provided by the standard library.

max returns the first argument when the arguments are equal.

Example

```
#include <algorithm>
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;

  // 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;
}
```

max_element

See also Algorithm

Finds maximum value in a range.

Syntax

```
#include <algorithm>
template <class ForwardIterator>
 InputIterator
 max_element(ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Compare>
 InputIterator
 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 less than (<) operator. This function can be used with all the datatypes provided by the standard library.

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.`

Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done for ***max_element***.

Example

```
#include <algorithm>
#include <vector>
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;
}
```

merge

Algorithm

Merge two sorted sequences into a third sequence.

Syntax

```
#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 provided by the function call. If a comparison function is provided, ***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 will 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, i.e., `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.

For ***merge*** at most `(last - first1) + (last2 - first2) - 1` comparisons are performed.

Example

```
#include <algorithm>
#include <vector>
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> 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>(cout," "));
  cout << endl;
  return 0;
}
```

min

See also Algorithm

Find and return the minimum of a pair of values.

Syntax

```
#include <algorithm>
template <class T>
 const T& min(const T& a, const T& b);
template <class T, class Compare>
 const T& min(const T& a, const T& b, Compare comp);
```

Description

The ***min*** algorithm determines and returns the minimum of a pair of values. In the second version of the algorithm, the optional argument `comp` defines a comparison function that can be used in place of the default "<" operator. This function can be used with all the datatypes provided by the standard library.

min returns the first argument when the two arguments are equal.

Example

```
#include <algorithm>
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;

  // 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;
}
```

min_element

See also Algorithm

Finds the minimum value in a range.

Syntax

```
#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 less than (<) operator. This function can be used with all the datatypes provided by the standard library.

Algorithm ***min_element*** returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range same range, the following corresponding conditions hold:

```
!(*j < *i)
```

or

```
comp(*j, *i) == false.
```

min_element performs exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

Example

```
#include <algorithm>
#include <vector>
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;
}
```

minus

See also Function object

Returns the result of subtracting its second argument from its first.

Syntax

```
#include<functional>
   template <class T>
   struct minus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const
              { return x - y; }
   };
```

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(), vec2.end(),
            vecResult.begin(), minus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` minus `vec2(n)`.

mismatch

Algorithm

Compares elements from two sequences and returns the first two elements that don't match each other.

Syntax

```
#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

```
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)`.

At most `last1 - first1` applications of the corresponding predicate are done.

Example

```
#include <algorithm>
#include <vector>
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;
}
```

modulus

See also Function object

Returns the remainder obtained by dividing the first argument by the second argument.

Syntax

```
#include<functional>
   template <class T>
   struct modulus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const
              { return x % y; }
   };
```

Description

modulus is a binary function object. Its `operator()` returns the remainder resulting from of `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(), vec2.end(),
            vecResult.begin(), modulus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain the remainder of `vec1(n)` divided by `vec2(n)`.

multimap

See also Container

An associative container providing access to non-key values using keys. ***multimap*** keys are not required to be unique.

Syntax

```
#include <map>
template <class Key, class T, class Compare = less<Key> >
 class multimap {

public:
// types
   typedef Key key_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef typename reference;
   typedef typename const_reference;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef typename difference_type;
   typedef reverse_iterator<iterator,
                 value_type,
                 reference,
                 difference_type> reverse_iterator;
   typedef reverse_iterator<const_iterator,
                 value_type,
                 const_reference,
                 difference_type> const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type, bool>

// Construct/Copy/Destroy
   explicit multimap (const Compare& comp = Compare());
   template <class InputIterator>
    multimap (InputIterator first,
        InputIterator last,
        const Compare& comp = Compare());
   multimap (const multimap<Key, T, Compare>& x);
   ~multimap ();
multimap<Key, T, Compare>& operator=
       const multimap<Key, T, Compare>& x);

// 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& x);
   iterator insert (iterator position, const value_type& x);
   template <class InputIterator>
    void insert (InputIterator first, InputIterator last);
   void erase (iterator position);
   size_type erase (const key_type& x);
   void erase (iterator first, iterator last);
   void swap (multimap<Key, T, Compare>& x);
// Observers
   key_compare key_comp () const;
   value_compare value_comp () const;
// Multimap operations
   iterator find (const key_value& x);
   const_iterator find (const key_value& x) const;
   size_type count (const key_type& x) const;
   iterator lower_bound (const key_type& x);
   const_iterator lower_bound (const key_type& x) const;
   iterator upper_bound (const key_type& x);
   const_iterator upper_bound (const key_type& x) const;
   pair<iterator, iterator> equal_range (const key_type& x);
   pair<const_iterator, const_iterator>
     equal_range (const key_type& x) const;
};
template <class Key,class T,class Compare>
 bool operator==
    (const multimap<Key, T, Compare>& x,
     const multimap<Key, T, Compare>& y);
template <class Key,
          class T,
          class Compare>
 bool operator<
    (const multimap<Key, T, Compare>& x,
     const multimap<Key, T, Compare>& y);
```

Description

multimap provides fast access to stored values of type `T` which are indexed by keys of a separate type. The default operation for key comparison is the `<` operator. Unlike ***map***, ***multimap*** allows insertion of duplicate keys.

multimap provides bidirectional iterators which 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*** provides a `typedef` to this pair called `value_type`.

The types used for both the template parameters `Key` and `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
```

The type used for the `Compare` template parameter must satisfy the requirements for binary functions.

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for ***multimap<Key,Compare>*** that takes two templated iterators:

```
template <class InputIterator>
 multimap (InputIterator, InputIterator, Compare);
```

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 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 multimap in the following two ways:

```
multimap<int,int,less<int> > ::value_type intarray[10];
multimap<int,int,less<int> >
   first_multimap(intarray,intarray + 10);
multimap<int, less<int> >
   second_multimap(first_multimap.begin(),first_multimap.end());
```

but not this way:

```
multimap<long, long, less<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 need to always supply the `Compare` template argument.

Example

```
#include <string>
#include <map>
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>(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("Febuary")));
   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>(cout,"\n"));

   return 0;
 }
```

Constructors and destructors

```
explicit multimap (const Compare& comp = Compare());
```

Default constructor. Constructs an empty multimap that will use the optional relation `comp` to order keys, if it is supplied.

```
template <class InputIterator>
multimap (InputIterator first,
          InputIterator last,
          const Compare& comp = Compare());
```

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 Value>`.

```
multimap (const multimap<Key, T, Compare>& x);
```

Copy constructor. Creates a new multimap by copying all pairs of `key` and `value` from `x`.

```
~multimap ();
```

The destructor. Releases any allocated memory for this multimap.

Assignment operator

```
multimap<Key, T, Compare>&
operator= (const multimap<Key, T, Compare>& x);
```

Assignment operator. Replaces the contents of `*this` with a copy of the multimap `x`.

Iterators

```
iterator begin() ;
```

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 `iterator` pointing to the last element stored in the multimap, i.e. 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, i.e., 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

`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)`

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);`

Erases the multimap element pointed to by the iterator `position`.

`size_type`
`erase` `(const key_type& x);`

Erases all elements with the key value `x` from the multimap, if any exist. Returns the number of erased elements.

`void`
`erase` `(iterator first, iterator last);`

Providing the iterators `first` and `last` point to the same multimap and `last` is reachable from `first`, all elements in the range `[first, last)` will be erased from the multimap.

`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 may be done right after `position` then it takes amortized constant time. Otherwise it will take `O(log N)` time.

```
template <class InputIterator>
void
insert (InputIterator first, InputIterator last);
```

Copies of each element in the range `[first, last)` will be 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 smallest multimap element whose `key` is greater or equal to `x`. If no such element exists then `end()` is returned.

```
const_iterator
lower_bound (const key_type& x)
```

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>& x);
```

Swaps the contents of the multimap `x` with the current multimap, `*this`.

```
upper_bound (const key_type& x)
```

Returns an iterator to the largest multimap element whose key is smaller or equal to `x`. If no such element exists then `end()` is returned.

```
const_iterator
upper_bound (const key_type& x)
```

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>& x,
                const multimap<Key, T, Compare>& 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>& x,
                const multimap<Key, T, Compare>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

multiset

Container

An associative container providing fast access to stored key values. Storage of duplicate keys is allowed.

Syntax

```
#include <set>
template <class Key, class Compare = less<Key> >
 class multiset {
public:
// typedefs
   typedef Key key_type;
   typedef Key value_type;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef typename reference;
   typedef typename const_reference;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef difference_type;
   typedef reverse_iterator<iterator,
                      value_type,
                      reference,
                      difference_type> reverse_iterator;
   typedef reverse_iterator<const_iterator,
                      value_type,
                      const_reference,
                      difference_type>
const_reverse_iterator;
// Construct/Copy/Destroy
   explicit multiset (const Compare& comp = Compare());
   template <class InputIterator>
    multiset (InputIterator first, InputIterator last,
             const Compare& comp = Compare());
   multiset (const multiset<Key, Compare>& x);
   ~multiset ();
   multiset<Key, Compare>&  operator=  (const  multiset<Key,
   Compare>& x);
// 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& x);
   iterator insert (iterator position, const value_type& x);
   template <class InputIterator>
    void insert (InputIterator first, InputIterator last);
   void erase (iterator position);
   size_type erase (const key_type& x);
   void erase (iterator first, iterator last);
   void swap (multiset<Key, Compare>& x);
// Observers
   pkey_compare key_comp () const;
   value_compare value_comp () const;
// Multiset operations
   iterator find (const key_value& x);
   size_type count (const key_type& x) const;
   iterator lower_bound (const key_type& x);
   iterator upper_bound (const key_type& x);
   pair<iterator, iterator> equal_range (const key_type& x);
   };
template <class Key,
          class Compare>
 bool operator==
    (const multiset<Key, Compare>& x,
     const multiset<Key, Compare>& y);
template <class Key,
          class Compare>
 bool operator<
    (const multiset<Key, Compare>& x,
     const multiset<Key, Compare>& y);
```

Description

multiset provides 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 provides bidirectional iterators which point to a stored key.

Any type used for the template parameter `Key` 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
```

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for ***multiset<Key,Compare>***, which takes two templated iterators:

```
template <class InputIterator>
 multiset (InputIterator, InputIterator);
```

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 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). 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,less<int> > first_multiset(intarray,intarray  +10);
multiset<int,                  less<int>                    >
second_multiset(first_multiset.begin(),first_multiset.end());
```

but not this way:

```
multiset<long,                 less<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 need to always supply the `Compare` template argument.

Example

```
#include <set>
 using namespace std;

 typedef multiset<int,less<int> > set_type;
 ostream& operator<<(ostream& out, const set_type& s)
 {
   copy(s.begin(),s.end(),
        ostream_iterator<set_type::value_type>(cout," "));
   return out;
 }

 int main(void)
 {
   // create a multiset  of int's
   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;
 }
```

Constructor and destructor

`explicit multiset (const Compare& comp = Compare());`

Default constructor. Constructs an empty multiset which will use the optional relation `Compare` to order keys, if it is supplied.

```
template <class InputIterator>
multiset (InputIterator first,
             InputIterator last,
             const Compare& comp = Compare());
```

Constructs a multiset containing values in the range `[first, last)`.

`multiset (const multiset<Key, Compare>& x);`

Copy constructor. Creates a new multiset by copying all key values from `x`.

`~multiset ();`

The destructor. Releases any allocated memory for this multiset.

Assignment operator

```
multiset<Key, Compare>&
operator=  (const  multiset<Key, Compare>& x);
```

Assignment operator. Replaces the contents of `*this` with a copy of the contents of `x`.

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, i.e., the off-the-end value.

`const_iterator` **`end`**`()`

Returns a `const_iterator` pointing to the last element stored in the multiset, i.e., 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.

`reverse_iterator` **`rend`**`()`

Returns a `reverse_iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

`const_reverse_iterator` **`rend`**`()`

Returns a `const_reverse_iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

Member functions

`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)`

Returns the pair `(lower_bound(x), upper_bound(x))`.

`void`
`erase` `(iterator position);`

Erases the multiset element pointed to by the iterator `position`.

`size_type`
`erase` `(const key_type& x);`

Erases all elements with the key value `x` from the multiset, if any exist. Returns the number of erased elements.

`void`
`erase` `(iterator first, iterator last);`

Providing the iterators `first` and `last` point to the same multiset and `last` is reachable from `first`, all elements in the range `[first, last)` will be erased from the multiset.

`iterator`
`find` `(const key_type& x);`

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 may be done right after position then it takes amortized constant time. Otherwise it will take `O(log N)` time.

`template <class InputIterator>`
`void`
`insert` `(InputIterator first, InputIterator last);`

Copies of each element in the range `[first, last)` will be 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)`

Returns an iterator to the smallest multiset element whose key is greater 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 () const;
```

Returns the number of elements in the multiset.

```
void
swap (multiset<Key, Compare >& x);
```

Swaps the contents of the multiset `x` with the current multiset, `*this`.

```
iterator
upper_bound (const key_type& x)
```

Returns an iterator to the largest multiset element whose key is smaller 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. This function is identical to `key_comp` for sets.

Non-member operators

```
operator==(const multiset<Key, Compare>& x,
const multiset<Key, Compare>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==)`. Otherwise it returns `false`.

```
operator< (const multiset<Key, Compare>& x,
const multiset<Key, Compare>& y);
```

Returns `true` if `x` is lexicographically less than y. Otherwise, it returns `false`.

negate

See also   Function object

Unary function object that returns the negation of its argument.

Syntax

```
#include <functional>
   template <class T>
   struct negate : unary_function<T, T, T> {
        T operator() (const T& x) const
              { return -x; }
   };
```

Description

negate is a unary function object. Its `operator()` returns the negation of its argument, i.e., `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)` will contain the negation of the element in `vec1(n)`.

negators

See also Function object

Function adaptors and function objects used to reverse the sense of predicate function objects.

Syntax

```
#include <functional>
template <class Predicate>
class unary_negate
  : public unary_function<Predicate::argument_type, bool> {
public:
   explicit unary_negate (const Predicate& pred);
   bool operator() (const argument_type& x) const;
};
template<class Predicate>
unary_negate <Predicate> not1 (const Predicate& pred);
template<class Predicate>
class binary_negate
  : public binary_function<Predicate::first_argument_type,
                           Predicate::second_argument_type,
                           bool                             >
{
public:
   explicit binary_negate (const Predicate& pred);
   bool operator() (const first_argument_type& x,
                    const second_argument_type& y) const;
};
template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Description

not1 and ***not2*** are functions that take predicate function objects as arguments and return predicate function objects with the opposite sense. ***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 provide return types for the negators, ***not1*** and ***not2***.

Example

```
#include<functional>
#include<algorithm>
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;
}
```

next_permutation

See also Algorithm

Generate successive permutations of a sequence based on an ordering function.

Syntax

```
#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`. So, for example, if a sequence includes the integers 1 2 3, that sequence has six permutations, which, in order from first to last 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 (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`.

At most `(last - first)/2` swaps are performed.

Example

```
#include <numeric>    //for accumulate
#include <vector>        //for vector
#include <functional> //for less
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>(cout," "));
  cout << endl << "     ";
  cout << "Previous permutation: ";
  copy(prev_m1.begin(),prev_m1.end(),
       ostream_iterator<int>(cout," "));

  cout << endl<< "     ";
  cout << "Next Permutation:     ";
  copy(next_m1.begin(),next_m1.end(),
       ostream_iterator<int>(cout," "));
  cout << endl << endl;

  cout << "Example 2: " << endl << "     ";
  cout << "Original values: ";
  copy(m2.begin(),m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << "     ";
  cout << "Previous Permutation: ";
  copy(prev_m2.begin(),prev_m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << "     ";

  cout << "Next Permutation:     ";
  copy(next_m2.begin(),next_m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << endl;

  return 0;
}
```

not1

See also Function adaptor

Function adaptor used to reverse the sense of a unary predicate function object.

Syntax

```
#include <functional>
template<class Predicate>
unary_negate <Predicate> not1 (const Predicate& pred);
```

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 provides 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***.

not2

See also   Function adaptor

Function adaptor used to reverse the sense of a binary predicate function object.

Syntax

```
#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 provides 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***.

not_equal_to

See also   Function object

Binary function object that returns `true` if its first argument is not equal to its second.

Syntax

```
#include <functional>
    template <class T>
    struct not_equal_to : binary_function<T, T, bool> {
         bool operator() (const T& x, const T& y) const
              { return x != y; }
    };
```

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(), vec2.end(),
            vecResult.begin(), not_equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was not equal to `vec2(n)` or a "1" if `vec1(n)` was equal to `vec2(n)`.

nth_element

Algorithm

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.

Syntax

```
#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 the provided comparison operator. After the algorithm applies, 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 precede that position in an ordered collection
- All elements following the nth position would 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.

The algorithm is linear, on average, where `N` is the size of the range `[first,last)`.

Example

```
#include<algorithm>
#include<vector>
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;
}
```

numeric_limits

Numeric limits library

A class for representing information about scalar types.

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>
```

Syntax

```
#include <limits>
template <class T>
 class numeric_limits {
 public:
 // General -- meaningful for all specializations.
    static const bool is_specialized ;
    static T min ();
    static T max ();
    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 ();
    static T round_error ();
    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 bool has_denorm ;
    static const bool tinyness_before ;
    static const float_round_style round_style ;
    static T denorm_min ();
    static T infinity ();
```

```
    static T quiet_NaN ();
    static T signaling_NaN ();
};

enum float_round_style {
  round_indeterminate       = -1,
  round_toward_zero         =  0,
  round_to_nearest          =  1,
  round_toward_infinity     =  2,
  round_toward_neg_infinity =  3
};
```

Description

numeric_limits is a class for representing information about scalar types. Specializations are provided for each fundamental type, both floating point and integer, including `bool`.

This class encapsulates information that is contained in the `<climits>` and `<cfloat>` headers, as well as providing additional information that is not contained in any existing C or C++ header.

Not all of the information provided by members is meaningful for all specializations of ***numeric_limits***. Any value which is not meaningful for a particular type is set to `0` or `false`.

Warning: The specializations for `wide chars` and `bool` will only be available if your compiler has implemented them as real types and not simulated them with typedefs.

Example

```
#include <limits>
using namespace std;

int main()
{
    numeric_limits<float> float_info;
    if (float_info.is_specialized && float_info.has_infinity)
    {
      // get value of infinity
      float finfinity=float_info.infinity();
    }
    return 0;
}
```

Member fields and functions

`static T`
`denorm_min ();`

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 ;`

Number of radix digits which can be represented without change. For built-in integer types, `digits` will usually be 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 ;`

Number of base 10 digits that can be represented without change. Meaningful for all specializations that declare `is_bounded` to be `true`.

`static T`

epsilon ();

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 bool
has_denorm ;

This field is `true` if the type allows denormalized values (variable number of exponent bits). 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` is 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 ();

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 ;

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. Must be `true` for any type claiming conformance to IEC 559.

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`. In the default ***numeric_limits<T>*** template, this flag must be `false`.

static T
max ();

Returns the maximum finite value. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

static const int
max_exponent ;

Maximum positive integer such that the radix raised to that power is in range. This field is meaningful for floating point types only.

static const int
max_exponent10 ;

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 ();

Returns the minimum finite value. For floating point types with denormalization, `min()` must return the minimum normalized value. The minimum denormalized value is provided by `denorm_min()`. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

static const int
min_exponent ;

Minimum negative integer such that the radix raised to that power is in range. This field is meaningful for floating point types only.

static const int
min_exponent10 ;

Minimum negative integer such that 10 raised to that power is in range. This field is meaningful for floating point types only.

static T
quiet_NaN ();

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 ();

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();

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.

operator!=, operator>, operator<=, operator>=

Utility operators

Operators for the C++ Standard Template Library.

Syntax

```
#include <utility>
template <class T>
 bool operator!= (const T& x, const T& y)
template <class T>
 bool operator> (const T& x, const T& y)
template <class T>
 bool operator<= (const T& x, const T& y)
template <class T>
 bool operator>= (const T& x, const T& y)
```

Description

To avoid redundant definitions of `operator !=` out of `operator ==` and of `operators >`, `<=`, and `>=` out of `operator<`, the library provides these definitions:

```
    operator != returns !(x==y),
    operator >  returns y<x,
    operator <= returns !(y<x), and
    operator >= returns !(x<y).
```

ostream_iterator

See also Iterator

Stream iterators provide iterator capabilities for ostreams and istreams. They allow generic algorithms to be used directly on streams.

Syntax

```
#include <iterator>
template <class T>
 class ostream_iterator : public output_iterator<T, Distance>
{
 public:
  ostream_iterator(ostream& s);
  ostream_iterator (ostream& s, const char* delimiter);
  ostream_iterator (const ostream_iterator<T>& x);
  ~ostream_itertor ();
  ostream_iterator<T>& operator=(const  T&  value);
  ostream_iterator<T>& operator* () const;
  ostream_iterator<T>& operator++ ();
  ostream_iterator<T> operator++ (int);
 };
template <class T, class Distance> inline bool
operator== (const istream_iterator<T, Distance>& x,
        const istream_iterator<T, Distance>& y);
```

Description

Stream iterators provide 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 will be 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.

Constructors

```
ostream_iterator (ostream& s);
```

Construct an ***ostream_iterator*** on the given stream.

```
ostream_iterator (ostream& s, const char* delimiter);
```

Construct an ***ostream_iterator*** on the given stream. The null terminated string delimitor is written to the stream after every element.

```
ostream_iterator (const ostream_iterator<T>& x);
```

Copy constructor.

Destructor

```
~ostream_iterator ();
```

Destructor

Operators

```
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 all do nothing. They simply allow the iterator to be used in common constructs.

Example

```
#include <iterator>
#include <numeric>
#include <deque>
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>(cout," + "));
 cout << *(d.end()-1) << " = " <<
    accumulate(d.begin(),d.end(),total) << endl;
 return 0;
}
```

output iterator

See also Iterator

A write-only, forward moving iterator.

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 read-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.

The following key pertains to the iterator descriptions 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 convertable 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.

pair

Utility class

A template for heterogeneous pairs of values.

Syntax

```
#include <utility>
template <class T1, class T2>
 struct pair {
        T1 first;
        T2 second;
        pair (const T1& x, const T2& y);
};
template <class T1, class T2>
 bool operator== (const pair<T1, T2>& x,
                  const pair T1, T2>& y);
template <class T1, class T2>
 bool operator< (const pair<T1, T2>& x,
                 const pair T1, T2>& y);
template <class T1, class T2>
 pair<T1,T2> make_pair (const T1&, const T2&);
```

Description

The ***pair*** class provides a template for encapsulating pairs of values that may be of different types.

Constructor

```
pair (const T1& x, const T2& y);
```

The constructor creates a pair of types `T1` and `T2`, making the necessary conversions in `x` and `y`.

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`.

```
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`.

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`.

partial_sort

See also Algorithm

Templated algorithm for sorting collections of entities.

Syntax

```
#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 (<) as the comparison operator for the sort. The second version uses the comparison function `comp`.

partial_sort does approximately `(last - first) * log(middle-first)` comparisons.

partial_sort_copy

See also Algorithm

Templated algorithm for sorting collections of entities.

Syntax

```
#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`. (i.e., the range: `[result_first, result_first+min(last-first, result_last- result_first))`. Basically, the effect is as if the range `[first,last)` were placed in a temporary buffer, sorted and then as many elements as possible were copied into the range `[result_first,result_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`.

partial_sort_copy does approximately `(last-first) * log(min(last-first, result_last-result_first))` comparisons.

Example

```
/***************************************************************
 *
 * partsort.cpp - Example program of partial sort.
 *
 * $Id: partsort.cpp,v 1.6 1995/10/06 19:03:56 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <vector>
 #include <algorithm>
 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>(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>(cout," "));
 cout << endl;

 return 0;
}
```

partial_sum

Generalized numeric operation

Calculates successive partial sums of a range of values.

Syntax

```
#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, 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,) will yield (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.

Exactly `(last - first) - 1` applications of the default `+` operator or `binary_op` are performed.

Example

```
#include <numeric>   //for accumulate
#include <vector>    //for vector
#include <functional> //for times
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 fto 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>(cout," "));
```

```
  cout << endl << endl;

  cout << "The partial sums: " << endl << "     " ;
  copy(sums.begin(),sums.end(),
       ostream_iterator<int>(cout," "));
  cout <<" should each equal (N*N + N)/2" << endl << endl;

  cout << "The partial products: " << endl << "     ";
  copy(prods.begin(),prods.end(),
       ostream_iterator<int>(cout," "));
  cout << " should each equal N!" << endl;

  return 0;
}
```

partition

See also Algorithm

Places all of the entities that satisfy the given predicate before all of the entities that do not.

Syntax

```
#include <algorithm>
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator
partition (BidirectionalIterator first,
           BidirectionalIterator last,
           Predicate pred);
```

Description

The ***partition*** algorithm places all the elements in the range `[first, last)` that satisfy `pred` before all the 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.

The ***partition*** algorithm does at most `(last - first)/2` swaps, and applies the predicate exactly `last - first` times.

Example

```
/***************************************************************
 *
 * prtition.cpp - Example program for partition.
 *
 * $Id: prtition.cpp,v 1.7 1995/10/06 19:18:57 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include<functional>
 #include<deque>
 #include<algorithm>
 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 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>(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>(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>(cout," "));
 cout << endl;

 return 0;
}
```

permutation

See also   Algorithm

Generate successive permutations of a sequence based on an ordering function.

plus

See also Function object

A binary function object that returns the result of adding its first and second arguments.

Syntax

```
#include <functional>
   template<class T>
   struct plus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const
              { return x + y; }
   };
```

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(), vec2.end(),
            vecResult.begin(), plus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` plus `vec2(n)`.

pointer_to_binary-function

A function object which adapts a pointer to a binary function to work where a ***binary_function*** is called for.

Syntax

```
#include <functional>
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& x, const Arg2& y) const;
};
template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
  ptr_fun (Result (*x)(Arg1, Arg2));
```

Description

The ***pointer_to_binary_function*** class encapsulates a pointer to a two-argument function. The class provides an `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 provided with the appropriate pointer to a function.

pointer_to_unary_function

See also Function object

A function object class that adapts a *pointer to a function* to work where a ***unary_function*** is called for.

Syntax

```
#include <functional>
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& x) const;
};
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
  ptr_fun (Result (*f)(Arg));
```

Description

The ***pointer_to_unary_function*** class encapsulates a pointer to a single-argument function. The class provides an `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 provided with the appropriate pointer to a function.

pop_heap

See also Algorithm

Moves the largest element off the heap.

Syntax

```
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 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 (i.e., 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 `[first, last -1)` back into a heap. You can then access the element in `last` using the vector or deque `back()` member function, or 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.

pop_heap performs at most `2 * log(last - first)` comparisons.

Example

```
#include <algorithm>
#include <vector>

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> 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;
}
```

predicate

A function or a function object that returns a boolean (true/false) value or an integer value.

prev_permutation

See also Algorithm

Generate successive permutations of a sequence based on an ordering function.

Syntax

```
#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`. So, for example, if a sequence includes the integers 1 2 3, that sequence has six permutations, which, in order from first to last, 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 (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`.

At most `(last - first)/2` swaps are performed.

Example

```
#include <numeric>    //for accumulate
#include <vector>        //for vector
#include <functional> //for less
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>(cout," "));
  cout << endl << "     ";
  cout << "Previous permutation: ";
  copy(prev_m1.begin(),prev_m1.end(),
       ostream_iterator<int>(cout," "));

  cout << endl<< "     ";
  cout << "Next Permutation:     ";
  copy(next_m1.begin(),next_m1.end(),
       ostream_iterator<int>(cout," "));
  cout << endl << endl;

  cout << "Example 2: " << endl << "     ";
  cout << "Original values: ";
  copy(m2.begin(),m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << "     ";
  cout << "Previous Permutation: ";
  copy(prev_m2.begin(),prev_m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << "     ";

  cout << "Next Permutation:     ";
  copy(next_m2.begin(),next_m2.end(),
       ostream_iterator<char>(cout," "));
  cout << endl << endl;

  return 0;
}
```

priority_queue

Container

A container adapter which behaves like a priority queue. Items popped from the queue are in order with respect to a "priority."

Syntax

```
#include <queue>
template <class T,
          class Container = vector<T>,
          class Compare = less<Container::value_type>>
 class priority_queue {
public:
// typedefs
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
protected:
   Container c;
   Compare comp;
public:
   explicit priority_queue (const Compare& x = Compare());
   template <class InputIterator>
    priority_queue (InputIterator first,
                    InputIterator last,
                    const Compare& x = Compare());
   bool empty () const;
   size_type size () const;
   const value_type& top () const;
   void push (const value_type& x)
   void pop()
};
```

Description

priority_queue is a container adaptor which 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 `comp`, is brought to the front of the queue whenever anything is pushed onto or popped off the queue.

priority_queue adapts any container that provides `front()`, `push_back()` and `pop_back()`.

Caveats

If your compiler does not support default template parameters, you must always provide a container 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>> var;
```

Example

```
#include <queue>
#include <deque>
#include <vector>
#include <string>
using namespace std;
```

```
int main(void)
{
  // Make a priority queue of int  using a deque 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
  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 greater
  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;
}
```

Constructor

```
explicit priority_queue (const Compare& x = Compare());
```

Default constructor. Constructs a priority queue that uses `Container` for its underlying implementation and `Compare` as its standard for determining priority.

```
template <class InputIterator>
    priority_queue (InputIterator first, InputIterator last,
                    const Compare& x = Compare());
```

Constructs a new priority queue and places into it every entity in the range `[first, last)`.

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 reference to the element in the queue with the highest priority.

ptr_fun

See also   Function adaptor

A function that is overloaded to adapt a *pointer to a function* to work where a function is called for.

Syntax

```
#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 a pointers to functions. and provide an `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 provided with the appropriate pointer to a function.

Example

```
#include<functional>
#include<deque>
#include<vector>
#include<algorithm>
using namespace std;
//Create a function
int factorial(int x)
{
  int result = 1;
  for(int i = 2; i <= x; i++)
  result *= i;
  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>(cout," "));
  cout << endl << endl;
  cout << "Have the factorials: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
  return 0;
}
```

push_heap

See also Algorithm

Places a new element into a heap.

Syntax

```
#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 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. It then 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)`.

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***.

For ***push_heap*** at most `log(last - first)` comparisons are performed.

Example

```
#include <algorithm>
#include <vector>
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> 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;
}
```

queue

See also Container

A container adaptor that behaves like a queue (first in, first out).

Syntax

```
#include <queue>
template <class T, class Container = deque<T>>
 class queue {
public:
// typedefs
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
protected:
   Container c;
public:
// Accessors
   bool empty () const { return c.empty; }
   size_type size () const { return c.size; }
   value_type& front () { return c.front; }
   const value_type& front () const { return c.front(); }
   value_type& back () { return c.back; }
   const value_type& back () const { return c.back(); }
   void push (const value_type& x) { c.push_back(x); }
   void pop () { c.pop_back(); }
};
template <class T, class Container>
 bool operator== (const queue<T, Container>& x,
                  const queue<T Container>& y)
   { return x.c == y.c; }
template <class T, class Container>
 bool operator< (const queue<T, Container>& x,
                 const queue<T, Container>& y)
   { return x.c < y.c; }
```

Description

The ***queue*** container adaptor lets a container function as a queue. In a queue, items are pushed onto the back of the container and removed from the front. The first items pushed onto 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 ***list*** and ***deque*** can be used.

Caveats

If your compiler does not support default template parameters, you must always provide 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;
```

Example

```
#include <queue>
#include <string>
#include <deque>
```

```
#include <list>
using namespace std;
int main(void)
{
  // Make a queue using a deque 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;
}
```

Member functions

`value_type&`
`back` `();`

Returns the item at the back of the queue (the last item pushed into the queue).

`const value_type&`
`back``() const;`

Returns 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` `();`

Returns the item at the front of the queue. This will be the first item pushed onto the queue unless `pop()` has been called since then.

`const value_type&`
`front` `() const;`

Returns 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.

random access iterator

See also Iterator

An iterator that reads and writes, and provides random access to a container.

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 and write, and provide random access to the containers they serve. These iterators satisfy the requirements listed below.

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 convertable to `bool`
`*a`	return value convertable to `T&`
`++r`	returns `X&`
`r++`	return value convertable to const `X&`
`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return `value` convertable 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 convertable 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 convertable to `bool`.

random_shuffle

Algorithm

Randomly shuffles elements of a collection.

Syntax

```
#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`.

The ***random_shuffle*** algorithm `(last - first) -1` swaps are done.

Example

```
#include<algorithm>
#include<vector>
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 random_shuffle: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(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>(cout," "));
  cout << endl;
  return 0;
}
```

raw_storage_iterator

Memory management

Enables iterator-based algorithms to store results into uninitialized memory.

Syntax

```
#include <memory>
template <class OutputIterator, class T>
 class raw_storage_iterator : public output_iterator {
public:
   explicit raw_storage_iterator (OutputIterator x);
   raw_storage_iterator<OutputIterator, t>& operator*();
   raw_storage_iterator<OutputIterator, T>&
     operator= (const T& element);
   raw_storage_iterator<OutputIterator>& operator++();
   raw_storage_iterator<OutputIterator> 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`.

Constructor

```
raw_storage_iterator (OutputIterator x);
```

Initializes the iterator to point to the same value that `x` points to.

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.

```
raw_storage_iterator <OutputIterator, T>&
operator++()
```

Pre-increment: advances the iterator and returns a reference to the updated iterator.

```
raw_storage_iterator<OutputIterator>
 operator++ (int);
```

Post-increment: advances the iterator and returns the old value of the iterator.

remove

See also Algorithm

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Syntax

```
#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 designates the end of the resulting range. ***remove*** is stable, that is, 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 operates by: 1) copying the values that are to be *retained* to the front of the sequence, and 2) returning an iterator that describes where the sequence of retained values ends. Elements that are after 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 "2's" removed.

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
#include<algorithm>
#include<vector>
#include<iterator>
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>(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>(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>(cout," "));
  cout << endl << endl;

  return 0;
}
```

remove_copy

See also Algorithm

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Syntax

```
#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 the end of the resulting range. ***remove_copy*** is stable, that is, 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***.

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
/***************************************************************
 *
 * remove.cpp - Example program of remove algorithm.
*
 * $Id: remove.cpp,v 1.9 1995/10/06 20:59:40 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <algorithm>
 #include <vector>
 #include <iterator>
 using namespace std;
 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>(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>(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>(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>(cout," "),
        all_true<int>());

 return 0;
}
```

remove_copy_if

See also Algorithm

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Syntax

```
#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 the end of the resulting range. ***remove_copy_if*** is stable, that is, the relative order of the elements in the resulting range is the same as their relative order in the original range.

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
/***************************************************************
 *
 * remove.cpp - Example program of remove algorithm.
*
 * $Id: remove.cpp,v 1.9 1995/10/06 20:59:40 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <algorithm>
 #include <vector>
 #include <iterator>
 using namespace std;
 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>(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>(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>(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>(cout," "),
        all_true<int>());

 return 0;
}
```

remove_if

See also   Algorithm

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Syntax

```
#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 the end of the resulting range. ***remove_if*** is stable, that is, 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 operates by: 1) copying the values that are to be *retained* to the front of the sequence, and 2) returning an iterator that describes where the sequence of retained values ends. Elements that are after 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.

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
 #include<algorithm>
 #include<vector>
 #include<iterator>
 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>(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>(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>(cout," "));
  cout << endl << endl;

  return 0;
}
```

replace

See also   Algorithm

Substitutes elements stored in a collection with new values.

Syntax

```
#include <algorithm>
template <class ForwardIterator, class T>
void replace (ForwardIterator first,
              ForwardIterator last,
              const T& old_value,
              const T& new_value);
```

Description

The ***replace*** algorithm replaces elements referred to by iterator `i` in the range `[first, last)` with `new_value` when the following condition holds: `*i == old_value`

Exactly `last - first` comparisons or applications of the corresponding predicate are done.

Example

```
#include<algorithm>
#include<vector>
#include<iterator>
using namespace std;

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, arr+10);

  //Print out original vector
  cout << "The original list: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(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>(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>(cout," "));
  cout << endl << endl;

  return 0;
}
```

replace_copy

See also Algorithm

Substitutes elements stored in a collection with new values.

Syntax

```
#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`. The algorithm compares elements referred to by iterator `i` in the range `[first, last)` with `old_value`. If `*i` does not equal `old_value`, then the ***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)`.

Exactly `last - first` comparisons between values are done.

Example

```
/***************************************************************
 *
 * replace.cpp - Example program of replace algorithm
 *
 * $Id: replace.cpp,v 1.9 1995/10/06 21:01:02 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <algorithm>
 #include <vector>
 #include <iterator>
 using namespace std;
 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>(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>(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>(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>(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>(cout, " "),
            all_true<int>(), 19);
    cout << endl;

    return 0;
}
```

replace_copy_if

See also   Algorithm

Substitutes elements stored in a collection with new values.

Syntax

```
#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`. The algorithm compares each element `*i` in the range `[first,last)` 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)`.

Exactly `last - first` applications of the predicate are performed.

Example

```
/***************************************************************
 *
 * replace.cpp - Example program of replace algorithm
 *
 * $Id: replace.cpp,v 1.9 1995/10/06 21:01:02 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ***************************************************************/
 #include <algorithm>
 #include <vector>
 #include <iterator>
 using namespace std;
 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>(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>(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>(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>(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>(cout, " "),
         all_true<int>(), 19);
 cout << endl;

 return 0;
}
```

replace_if

See also   Algorithm

Substitutes elements stored in a collection with new values.

Syntax

```
#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`.

Exactly `last - first` applications of the predicate are done.

Example

```
#include<algorithm>
#include<vector>
#include<iterator>
using namespace std;
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, arr+10);
  //Print out original vector
  cout << "The original list: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(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>(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>(cout," "));
  cout << endl << endl;
```

```
  return 0;
}
```

return_temporary_buffer

See also Memory handling primitive

Pointer based primitive for handling memory.

Syntax

```
#include <memory>
template <class T>
 void return_temporary_buffer (T* p, T*);
```

Description

The ***return_temporary_buffer*** templated function returns a buffer, previously allocated through ***get_temporary_buffer***, to available memory. Parameter `p` points to the buffer.

reverse

See also   Algorithm

Reverse the order of elements in a collection.

Syntax

```
#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.

reverse performs exactly `(last - first)/2` swaps.

Example

```
#include<algorithm>
#include<vector>
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 reverse: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int>(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>(cout," "));
  cout << endl;
  return 0;
}
```

reverse_bidirectional_iterator, reverse_iterator

See also Iterator

An iterator that traverses a collection backwards.

Syntax

```
#include <iterator>
template <class BidirectionalIterator,
          class T,
          class Reference = T&,
          class Distance = ptrdiff_t>
class reverse_bidirectional_iterator
   : public bidirectional_iterator<T, Distance>  {
  protected:
    BidirectionalIterator current;
  public:
    reverse_bidirectional_iterator ();
    explicit reverse_bidirectional_iterator
      (BidirectionalIterator x);
    BidirectionalIterator base ();
    Reference operator* ();
    reverse_bidirectional_iterator <BidirectionalIterator,
                                    T,
                                    Reference,
                                    Distance> &
      operator++ ();
    reverse_bidirectional_iterator <BidirectionalIterator,
                                    T,
                                    Reference,
                                    Distance>
      operator++ (int);
    reverse_bidirectional_iterator <BidirectionalIterator,
                                    T,
                                    Reference,
                                    Distance> &
      operator-- ();
    reverse_bidirectional_iterator <BidirectionalIterator,
                                    T,
                                    Reference,
                                    Distance>
      operator-- (int);
  };

  template <class BidirectionalIterator,
            class T,
            class Reference,
            class Distance>
  bool operator== (
     const reverse_bidirectional_iterator
        <BidirectionalIterator,
         T,
         Reference,
         Distance>& x,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,
         T,
         Reference,
```

```
         Distance>& y);
template <class RandomAccessIterator,
          class T,
          class Reference = T&,
          class Distance = ptrdiff_t>
class reverse_iterator
   : public random_access_iterator<T, Distance>  {
  protected:
    RandomAccessIterator current;
  public:
    reverse_iterator ();
    explicit reverse_iterator (RandomAccessIterator x);
    RandomAccessIterator base ();
    Reference operator* ();
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance> &
      operator++ ();
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance>
      operator++ (int);
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance> &
      operator-- ();
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance>
      operator-- (int);
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance>
      operator+ (Distance n) const;
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance> &
      operator+= (Distance n);
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance>
      operator- (Distance n) const;
    reverse_iterator <RandomAccessIterator,
                      T,
                      Reference,
                      Distance> &
      operator-= (Distance n);
    Reference operator[] (Distance n);
```

```
template <class RandomAccessIterator,
          class T,
          class Reference,
          class Distance> bool operator== (
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& x,
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& y);

template <class RandomAccessIterator,
          class T,
          class Reference,
          class Distance> bool operator< (
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& x,
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& y);

template <class RandomAccessIterator,
          class T,
          class Reference,
          class Distance> Distance operator- (
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& x,
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
       Distance>& y);

template <class RandomAccessIterator,
          class T,
          class Reference,
          class Distance>
reverse_iterator<RandomAccessIterator,
                 T,
                 Reference,
                 Distance> operator+ (
    Distance n,
    const reverse_iterator
      <RandomAccessIterator,
       T,
       Reference,
```

```
            Distance>& x);
};
```

Description

The iterators ***reverse_iterator*** and ***reverse_bidirectional_iterator*** correspond to ***random_access_iterator*** and ***bidirectional_iterator***, except they traverse the collection they point to in the opposite direction. The fundamental relation 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_bidirectional_iterators*** :

{bmc. bullet.bmp}   These iterators may be instantiated with the default constructor or by a single argument constructor that initializes the new `reverse_bidirectional_iterator` with a `bidirectional_iterator`.

- `operator*` returns a reference to the current value pointed to.
- `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==` This non-member operator 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 which initializes the new `reverse_iterator` with a `random_access_iterator`.
- `operator*` returns a reference to the current value pointed to.
- `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 returns `true` if the iterators `x` and `y` point to the same item.
- The remaining operators (`<, +, - , +=, -=`) are redefined to behave exactly as they would in a `random_access_iterator`, except with the sense of direction reversed.

All iterator operations are required to take at most amortized constant time.

Example

```
#include<iterator>
#include<vector>
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_itertor: " << endl
         << "     ";
  for(; rev != rev_end; rev++)
    cout << *rev << " ";

  return 0;
}
```

reverse_copy

See also Algorithm

Reverse the order of elements in a collection while copying them to a new collection.

Syntax

```
#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.

reverse_copy performs exactly `(last - first)` assignments.

Example

```
/*****************************************************************
 *
 * reverse.cpp - Example program reverse algorithm.
 *        See Class Reference Section
 *
 * $Id: reverse.cpp,v 1.7 1995/10/06 19:35:37 hart Exp $
 *
 * $$RW_INSERT_HEADER "slyrs.str"
 *
 ****************************************************************/
 #include <algorithm>
 #include <vector>
 using namespace std;
 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 elements in original (sorted) order.
  //
  cout << "Elements before reverse: " << endl << "    ";
  copy(v.begin(), v.end(), ostream_iterator<int>(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>(cout," "));
 cout << endl << endl;

 cout << "A reverse_copy to cout: " << endl << "   ";
 reverse_copy(v.begin(), v.end(),
         ostream_iterator<int>(cout, " "));
 cout << endl;

 return 0;
}
```

reverse_iterator

See the ***reverse_bidirectional_iterator*** section of this reference.

rotate, rotate_copy

Algorithm

Left rotates the order of items in a collection, placing the first item at the end, second item first, etc., until the item pointed to by a specified iterator is the first item in the collection.

Syntax

```
#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 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 will illustrate 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 will be:

1 3 5 2 4 6 8

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.

For ***rotate*** at most `last - first` swaps are performed.

For ***rotate_copy*** `last - first` assignments are performed.

Example

```
#include<algorithm>
#include<vector>
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>(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>(cout," "));
  cout << endl;

  return 0;
}
```

search

Algorithm

Finds a subsequence within a sequence of values that is element-wise equal to the values in an indicated range.

Syntax

```
#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);
```

Description

The ***search*** algorithm searches for a subsequence (`first2, last2`) within a sequence (`first1, last1`), and returns the beginning location of the subsequence. If it does not find the subsequence, ***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.

search performs at most `(last1 - first1)*(last2-first2)` applications of the corresponding predicate.

Example

```
#include<algorithm>
#include<list>
using namespace std;
int main()
{
  // Initialize a list sequence and
  // subsequence 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 subsequence, " << subseq
       << ", was found at the ";
  cout << endl << "location identified by a '*'"
       << endl << "     ";
  // Create an iterator to identify the location of
  // subsequence 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;
}
```

Sequence

A *sequence* is a container that organizes a set of objects, all 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 will 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.

set

See also Container

An associative container that supports unique keys.

Syntax

```
#include <set>
template <class Key, class Compare = less<Key>>
 class set {
public:
 // types
   typedef Key key_type;
   typedef Key value_type;
   typedef typename reference;
   typedef typename const_reference;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef difference_type;
   typedef reverse_iterator<iterator, value_type,
           reference, difference_type> reverse_iterator;
   typedef const_reverse_iterator<const_iterator,
           value_type, reference  difference_type>
           const_reverse_iterator;
 // Construct/Copy/Destroy
   explicit set (const Compare& = Compare());
   template <class InputIterator>
    set (InputIterator, InputIterator, const Compare& = Compare());
   set (const set<Key, Compare>&);
   ~set ();
   set<Key, Compare>& operator= (const set Key, Compare>&);
 // Iterators
   iterator begin () const;
   iterator end () const;
   reverse_iterator rbegin ();
   reverse_iterator rend ();
 // 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 (iterator, InputIterator, InputIterator);
   void erase (iterator);
   size_type erase (const key_type&);
   void erase (iterator, iterator);
   void swap (set<Key, Compare>&);
 // 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_value&) const;
   iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&) const
};
 // Comparison
template <class Key, class Compare>
 bool operator== (const set<Key, Compare>& x,
                  const set <Key,Compare>& y);
template <class Key, class Compare>
 bool operator< (const set <Key, Compare>& x,
                 const set <Key, Compare>& y);
```

Description

set<T,Compare> is a kind of associative container that supports unique keys and provides 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`.

Any type used for the template parameter `Key` 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
```

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Caveats

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for set `<Key,Compare>` that takes two templated iterators:

```
template <class InputIterator>
 set (InputIterator, InputIterator);
```

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 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 set in the following two ways:

```
int intarray[10];
set<int,less<int> > first_set(intarray,intarray + 10);
set<int, less<int> >
second_set(first_set.begin(),first_set.end());
```

but not this way:

```
set<long, less<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 need to always supply the `Compare` template argument.

Example

```
#include <set>
using namespace std;

typedef set<double,less<double> > set_type;
ostream& operator<<(ostream& out, const set_type& s)
{
  copy(s.begin(), s.end(),
       ostream_iterator<set_type::value_type>(cout," "));
  return out;
}

int main(void)
{
  // create a set of double's, and one of int's
  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;
}
```

Constructors and destructors

```
explicit
```

```
set (const Compare& comp = Compare());
```

The default constructor. 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 (<)`.

```
template <class InputIterator>
set (InputIterator first, InputIterator last, const Compare& comp = Compare
  ());
```

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 (<)`.

```
set (const set<Key, Compare>& x);
```

Copy constructor. Creates a copy of `x`.

```
~set ();
```

The destructor. Releases any allocated memory for self.

Assignment operator

```
set<Key, Compare>&
operator= (const set Key, Compare>& x);
```

Assignment operator. Self will share an implementation with `x`. Returns a reference to self.

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.

Public member functions

```
size_type
count (const key_type& x) const;
```

Returns the number of elements equal to `x`. Since a set supports unique keys, `count` will always return 1.

```
bool
empty () const;
```

Returns `true` if the size is zero.

```
pair<iterator, iterator>
equal_range (const key_type&  x) const;
```

Returns pair<lower_bound(),upper_bound()>. The equal_range function indicates the valid range for insertion of x into the set.

```
void
erase (iterator position);
```

Removes the element pointed to by position.

```
size_type
erase (const key_type& x);
```

Removes all the elements matching x. Returns the number of elements erased. Since a set supports unique keys, erase will always return 1.

```
void
erase (iterator first, iterator last);
```

Removes the elements in the range [first, last).

```
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 in self according to the comparison function object. The template's default comparison function object is less (<).

```
iterator
insert (iterator position, const value_type& x);
```

Inserts x before position. The return value points to the inserted x.

```
template <class InputIterator>
void
insert(iterator position, InputIterator first,
         InputIterator last);
```

Inserts copies of the elements in the range [first, last] before position.

```
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 size () of the largest possible set.

```
size_type
size () const;
```

Returns the number of elements.

```
void
swap (set<Key, Compare>& 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 `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.

Global operators

```
template <class Key, class Compare>
 bool operator== (const set<Key, Compare>& x,
                  const set<Key, Compare>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class Key, class Compare>
bool operator< (const set <Key, Compare>& x,
                const set <Key, Compare>& y);
```

Returns true if the elements contained in `x` are lexicographically less than the elements contained in `y`.

set_difference

See also Algorithm

Basic set operation for sorted sequences.

Syntax

```
#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

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.

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
#include<algorithm>
#include<set>
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, a1+10), even(a2, 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>(cout," "));
  cout << "} - {";
  copy(even.begin(),even.end(),
       ostream_iterator<int>(cout," "));
  cout << "} =" << endl << "{";
  set_difference(all.begin(), all.end(),
                 even.begin(), even.end(), odd_ins);
  copy(odd.begin(),odd.end(),
       ostream_iterator<int>(cout," "));
  cout << "}" << endl << endl;

  return 0;
}
```

set_intersection

See also Algorithm

Basic set operation for sorted sequences.

Syntax

```
#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.

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
#include<algorithm>
#include<set>
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, a1+6),
                       result, small(a3,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>(cout," "));
  cout << "} intersection {";
  copy(odd.begin(),odd.end(),
       ostream_iterator<int>(cout," "));
  cout << "} =" << endl << "{";
  set_intersection(small.begin(), small.end(),
```

```
                    odd.begin(), odd.end(), res_ins);
  copy(result.begin(),result.end(),
       ostream_iterator<int>(cout," "));
  cout << "}" << endl << endl;

  return 0;
}
```

set_symmetric_difference

See also   Algorithm

Basic set operation for sorted sequences.

Syntax

```
#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

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.

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
#include<algorithm>
#include<set>
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,a1+6), result,
                     small(a3,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>(cout," "));
  cout << "} with {";
  copy(odd.begin(),odd.end(),
       ostream_iterator<int>(cout," "));
  cout << "} =" << endl << "{";
  set_symmetric_difference(small.begin(), small.end(),
                    odd.begin(), odd.end(), res_ins);
  copy(result.begin(),result.end(),
       ostream_iterator<int>(cout," "));
  cout << "}" << endl << endl;

  return 0;
}
```

set_union

See also Algorithm

Basic set operation for sorted sequences.

Syntax

```
#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, that is, 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.

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
#include<algorithm>
#include<set>
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, a2+6),
                        result, small(a3,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>(cout," "));
  cout << "} union {";
  copy(even.begin(),even.end(),
       ostream_iterator<int>(cout," "));
  cout << "} =" << endl << "{";
```

```
  set_union(small.begin(), small.end(),
           even.begin(), even.end(), res_ins);
  copy(result.begin(),result.end(),
       ostream_iterator<int>(cout," "));
  cout << "}" << endl << endl;

  return 0;
}
```

sort

See also Algorithm

Templated algorithm for sorting collections of entities.

Syntax

```
#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.

sort performs approximately `NlogN`, where `N` equals `last - first`, comparisons on the average.

Example

```
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
struct associate
{
  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;
}
```

sort_heap

Algorithm

Converts a heap into a sorted collection.

Syntax

```
#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 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, i.e., the elements may not be in the same relative order after ***sort_heap*** is applied.

sort_heap performs at most `NlogN` comparisons where `N` is equal to `last - first`.

Example

```
#include <algorithm>
#include <vector>

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> 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;
}
```

stable_partition

See also   Algorithm

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.

Syntax

```
#include <algorithm>
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator
stable_partition (BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred);
```

Description

The ***stable_partition*** algorithm places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it. It returns an iterator `i` that is one past the end of the group of elements that satisfy `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.

The ***stable_partition*** algorithm does at most `(last - first) * log(last - first)` swaps. and applies the predicate exactly `last - first` times.

Example

```
#include<functional>
 #include<deque>
 #include<algorithm>
 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>(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>(cout," "));
  return 0;
}
```

stable_sort

See also Algorithm

Templated algorithm for sorting collections of entities.

Syntax

```
#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.

stable_sort does at most `N(logN) **2`, where `N` equals `last -first`, comparisons; if enough extra memory is available, it is `NlogN`.

Example

```
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
struct associate
{
  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;
}
```

stack

See also Container

A container adaptor which behaves like a stack (last in, first out).

Syntax

```
#include <stack>
template <class T, class Container = deque<T>>
 class stack {
public:
// typedefs
   typedef Container::value_type value_type;
   typedef Container::size_type size_type;
protected:
   Container c;
public:
// Accessors
   bool empty () const;
   size_type size () const;
   value_type& top ();
   const value_type& top () const;
   void push (const value_type& x);
   void pop ();
};
template <class T, class Container = deque<T>>
 bool operator== (const stack<Container>& x, const
 stack<Container>& y);
template <class T, class Container = deque<T>>
 bool operator< (const stack<Container>& x, const
 stack<Container>& y);
```

Description

The ***stack*** container adaptor 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 provides the operations, `back()`, `push_back()`, and `pop_back()`. In particular, ***vector*** , ***list*** , and ***deque*** can be used.

Caveats

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;
```

Example

```
#include <stack>
#include <vector>
#include <deque>
#include <string>
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++)
  {
    cout << ss.top() << endl;
    ss.pop();
  }

  return 0;
}
```

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 the item at the top of the stack. This will be the last item pushed onto the stack unless `pop()` has been called since then.

`const value_type&`
`top` `() const;`

Returns the item at the top of the stack as a `const value_type`.

Stream iterators

See also Iterators

Stream iterators provide iterator capabilities for ostreams and istreams. They allow generic algorithms to be used directly on streams.

string

See also String library

A specialization of the ***basic_string*** class.

swap

See also Algorithm

Exchange values stored in two locations.

Syntax

```
#include <algorithm>
template <class T>
 void swap (T& a, T& b);
```

Description

The ***swap*** algorithm exchanges the values in `a` and `b`.

swap_ranges

See also   Algorithm

Exchange a range of values in one location with those in another.

Syntax

```
#include <algorithm>
template <class ForwardIterator1, class ForwardIterator2>
 ForwardIterator2 swap_ranges (ForwardIterator1 first1,
                              ForwardIterator 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, i.e., `first2 + (last1 -first1)`. The result of ***swap_ranges*** is undefined if the two ranges `[first, last)` and `[first2, first2 + (last1 - first1))` overlap.

Example

```
#include <vector>
#include <algorithm>
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,d1 + 10);
  // Output original vector
  cout << "For the vector: ";
  copy(v.begin(),v.end(),ostream_iterator<int>(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>(cout," "));
  return 0;
}
```

times

See also Function object

A binary function object that returns the result of multiplying its first and second arguments.

Syntax

```
#include<functional>
   template <class T>
   struct times : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const
              { return x * y; }
   };
```

Description

times is a binary function object. Its `operator()` returns the result of multiplying `x` and `y`. You can pass a ***times*** 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. ***times*** 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(), times<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` times `vec2(n)`.

transform

Algorithm

Applies an operation to a range of values in a collection and stores the result.

Syntax

```
#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 inplace.

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)))
```

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.

Exactly `last1 - first1` applications of `op` or `binary_op` are performed.

Example

```
#include<functional>
#include<deque>
#include<algorithm>
#include<iomanip.h>
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, arr1+5), d2(arr2, 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(), times<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;
}
```

unary_function

See also Function object

Abstract base function for unary function objects.

Syntax

```
#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 provides 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 provide the typedefs `argument_type` and `result_type`. The ***unary_function*** class makes the task of creating templated 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***.

unary_negate

See also Function adaptor (negator)

Function object that returns the complement of the result of its unary predicate.

Syntax

```
#include<functional>
template <class Predicate>
class unary_negate
  : public unary_function<Predicate::argument_type, bool> {
public:
   explicit unary_negate (const Predicate& pred);
   bool operator() (const argument_type& x) const;
};
template<class Predicate>
unary_negate <Predicate> not1 (const Predicate& pred);
```

Description

unary_negate is a function object class that provides a return type for the function adaptor ***not1***. ***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.

Note that ***not1*** works only with function objects that are defined as subclasses of the class ***unary_function***.

Constructor

```
explicit unary_negate (const Predicate& pred);
```

Construct a unary_negate object from predicate `pred`.

Operator

```
bool operator() (const argument_type& x) const;
```

Return the result of `pred(x)`.

uninitialized_copy

See also Memory management

An algorithms that uses ***construct*** to copy values from one range to another location.

Syntax

```
#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.

uninitialized_fill

See also   Memory management

Algorithm that uses the ***construct*** algorithm for setting values in a collection.

Syntax

```
#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.

uninitialized_fill_n

See also Memory management

Algorithm that uses the ***construct*** algorithm for setting values in a collection.

Syntax

```
#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.

unique, unique_copy

Algorithm

Removes consecutive duplicates from a range of values and places the resulting unique values into the result.

Syntax

```
#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 tests for equality, and the other tests whether a binary predicate applied to adjacent elements is true. 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.

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.

Exactly `(last - first) - 1` applications of the corresponding predicate are performed.

Example

```
#include<algorithm>
#include<vector>
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, 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>(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>(cout," "));

   return 0;
}
```

upper_bound

See also Algorithm

Determines the last valid position for a value in a sorted container.

Syntax

```
#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 part 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
```

upper_bound performs at most `log(last - first) + 1` comparisons.

Example

```
#include <vector>
#include <algorithm>
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;
}
```

vector

See also Container

Sequence that supports random access iterators.

Syntax

```
#include <vector>
template <class T>
public:
 // Types
   typedef typename reference;
   typedef typename const_reference;
   typedef typename iterator;
   typedef typename const_iterator;
   typedef typename size_type;
   typedef typename difference_type;
   typedef T value_type;
   typedef  reverse_iterator<iterator, value_type,
            reference, difference_type> reverse_iterator;
   typedef const_reverse_iterator<const_iterator,
           value_type,      reference,      difference_type>
const_reverse_iterator;
 // Construct/Copy/Destroy
   explicit vector ();
   explicit vector (size_type, const T& = T());
   vector (const vector<T>&);
   template <class InputIterator>
    vector (InputIterator, InputIterator);
   ~vector ();
   vector<T>& operator= (const vector<T>&);
   template <class InputIterator>
    void assign (InputIterator first, InputIterator last);
   template <class Size, class T>
    void assign (Size n, const T& t = T());
 // 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,T c = T());
   size_type capacity () const;
   bool empty () const;
   void reserve (size_type);
 // Element Access
   reference operator[] (size_type);
   const_reference operator[] (size_type) const;
   reference at (size_type n);
```

```
   const_reference at (size_type n) 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& = T());
   void insert (iterator, size_type, const T& = T());
   template <class InputIterator>
    void insert (iterator, InputIterator, InputIterator);
   void erase (iterator);
   void erase (iterator, iterator);
   void swap (vector<T>&);
};
 // Comparison
template <class T>
 bool operator== (const vector<T>&, const vector <T>&);
template <class T>
 bool operator< (const vector<T>&, const vector<T>&);
```

Description

vector<T> 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 random access iterator referring to `const T`. 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`):

```
Default constructor   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
```

Caveats

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>*** that takes two templated iterators:

```
template <class InputIterator>
 vector (InputIterator, InputIterator);
```

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.

Example

```
#include <vector>
using namespace std;

ostream& operator<<(ostream& out, const vector<int>& v)
{
  copy(v.begin(), v.end(), ostream_iterator<int>(out," "));
  return out;
}

int main(void)
{
  // create a vector of double's, and one of int's
  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;
}
```

Constructors and destructors

`explicit vector ();`

The default constructor. Creates a vector of length zero.

`explicit vector (size_type n, const T& value = T());`

Creates a vector of length `n`, containing `n` copies of value.

`vector (const vector<T>& x);`

Creates a copy of `x`.

```
template <class InputIterator>
vector (InputIterator first, InputIterator last);
```

Creates a vector of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last);`

`~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 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 iterator that points to the past-the-end value.

`const_reverse_iterator **rbegin** () const;`

Returns a constant random access iterator that points to the past-the-end value.

`reverse_iterator **rend** ();`

Returns a random access iterator that points to the first element.

`const_reverse_iterator **rend** () const;`

Returns a constant random access iterator that points to the first element.

Assignment operator

`vector<T>& operator**=** (const vector<T>& x);`

Assignment operator. Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to self.

Reference operatorsreference 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)`.

```
template <class Size, class T>
void
assign (Size n, const T& 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.

```
bool
empty () const;
```

Returns `true` if the `size` is zero.

```
void
erase (iterator position);
```

Removes the element pointed to by `position`.

```
void
erase (iterator first, iterator last);
```

Removes the elements in the range `[first, last)`.

```
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 = T());
```

Inserts `x` before `position`. The return value points to the inserted `x`.

```
void
insert (iterator position, size_type n, const  T& x = T());
```

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.

```
void
resize (size_type sz, T c = T());
```

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>& x);
```

Exchanges self with `x`.

Non-member operatorstemplate <class T>

```
bool operator== (const vector<T>& x, const  vector  <T>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T>
bool operator< (const vector<T>& x,  const  vector  <T>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

wstring

String library

A specialization of the ***basic_string*** class. For more information about strings, see the entry ***basic_string***.