Systems/C++
C++ Library 2.25

Systems/C++
C++ Library
Version 2.25

Contents

Chapter 1

Introduction

The Systems/C++ C++ library provides the classes and functionality associated with the C++ language. It consists of two library files. One, `LIBCXX` on OS/390 or `libcxx_mvs.a` on Unix or Windows, provides basic language-level functionality, such as helper functions for exceptions and the default `operator new` function. This library is a central part of the C++ language and should always be linked into any program built with Systems/C++. The other, `LIBSTDCX` on OS/390 or `libstdcxx_mvs.a` on Unix or Windows, provides the complete ANSI C++11 (ISO/IEC 14882:2011) standard library, including iostreams and the Standard Template Library. This document focuses on the Standard Template Library.

Chapter 2

Linking with the Systems/C++ library for OS/390 and z/OS

To produce Systems/C++ programs for OS/390 and z/OS, the `PLINK` utility must be used to prepare the objects for linking with the IBM BINDER or older IEWL linker. `PLINK` prepares the program, gathering the objects together, and processing C++ language features, such as static C++ constructors/destructors.

The *Systems/C Utility* manual has more detailed information regarding the `PLINK` utility. Also, the *Systems/C++ Compiler* manual has more information regarding linking and running C++ programs.

For example, to pre-link the Systems/C++ object named `PROG` found in the *mypds* PDS on OS/390 or z/OS:

```
//PLINK EXEC PGM=PLINK
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB  DD DSN=DIGNUS.LIBCXX,DISP=SHR
//        DD DSN=DIGNUS.LIBSTDCX,DISP=SHR
//        DD DSN=DIGNUS.LIBCR,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

Note that the `LIBCXX` and `LIBSTDCX` PDSs were specified in the `SYSLIB DD` definition, as appropriate to your installation.

On UNIX and Windows workstations, to link the Systems/C++ program `prog.obj`, the `PLINK` command would be:

```
plink prog.obj C:\sysc\lib\libstdcxx_mvs.a C:\sysc\lib\libcxx_mvs.a
"-SC:\sysc\lib\objs_rent\&M"
```

This PLINK command specifies the two C++ DAR archives for MVS, followed by the Systems/C re-entrant library.

Chapter 3

Linking with Systems/C++ z/Architecture library for z/OS

Systems/C++ also provides the Systems/C++ z/Architecture library, for z/OS z/Architecture programs.

Programs compiled with the *–march=zarch* option should be linked with the z/Architecture Systems/C and Systems/C++ libraries.

The Systems/C++ library provides all of the z/Architecture features of the Systems/C library, including full access to the entire 64-bit addressing range for data, z/Architecture Direct-CALL (DCALL) support, and using z/Architecture code in an AMODE other than 64-bit.

For more details about these features, consult the *Systems/C C Library* manual.

Linking with the Systems/C++ z/Architecture library is analogous to linking with the OS/390 and z/OS library, simply replacing the non-z/Architecture libraries with their z/Architecture versions.

The *Systems/C C Library* manual provides the details of how to locate the z/Architecture C library on the various supported hosts.

The z/Architecture variants Systems/C++ library can be found in the `libcxx_mvsz.a` and `libstdcxx_mvsz.a` **DAR** archive libraries on cross-platform hosts. On OS/390 and z/OS the z/Architecture library is located in the `LIBCXXZ` and `LIBSTCXZ` PDSs.

The following JCL sample executes the Systems/C pre-linker (**PLINK**) on OS/390 or z/OS, linking with the z/Architecture C++ library, and then the z/Architecture C library:

```
//PLINK EXEC PGM=PLINK
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB  DD DSN=DIGNUS.LIBCXXZ,DISP=SHR
//        DD DSN=DIGNUS.LIBSTCXZ,DISP=SHR
//        DD DSN=DIGNUS.LIBCRZ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

On a cross-platform host, Windows in this example, the analagous **PLINK** command would be:

```
plink -omyoutput.obj prog.obj C:\sysc\lib\libstdcxx_mvsz.a
C:\sysc\lib\libcxx_mvsz.a "-SC:\sysc\lib\objs_rent_z\&M"
```

This `PLINK` command specifies the two C++ `DAR` archives for the Systems/C++ z/Architecture library, followed by the directory for the Systems/C z/Architecture library.

Chapter 4

Linking with the Systems/C++ library for Linux and z/Linux

Since version 1.95 of Systems/C++, **DCXX** is compatible with **g++** when building for Linux and z/Linux, and will work with the distribution-provided GNU `libstdc++`. As a result, the Systems/C++ library no longer supports Linux and z/Linux.

For more information about linking on Linux/390 and z/Linux, see the `cc` and `ld` command manual pages on these systems.

Chapter 5

Introduction to the STL

The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

Containers and algorithms

Like many class libraries, the STL includes *container* classes: classes whose purpose is to contain other objects. The STL includes the classes `vector`, `list`, `deque`, `set`, `multiset`, and `map`. . Each of these classes is a template, and can be instantiated to contain any type of object. You can, for example, use a `vector<int>` in much the same way as you would use an ordinary C array, except that `vector` eliminates the chore of managing dynamic memory allocation by hand.

```
vector<int> v(3);            // Declare a vector of 3 elements.
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];          // v[0] == 7, v[1] == 10, v[2] == 17
```

The STL also includes a large collection of *algorithms* that manipulate the data stored in containers. You can reverse the order of elements in a `vector`, for example, by using the `reverse` algorithm.

```
reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7
```

There are two important points to notice about this call to `reverse`. First, it is a global function, not a member function. Second, it takes two arguments rather than one: it operates on a *range* of elements, rather than on a container. In this particular case the range happens to be the entire container `v`.

The reason for both of these facts is the same: `reverse`, like other STL algorithms, is decoupled from the STL container classes. This means that `reverse` can be used not only to reverse elements in vectors, but also to reverse elements in lists, and even elements in C arrays. The following program is also valid.

```
double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
reverse(A, A + 6);
for (int i = 0; i < 6; ++i)
   cout << "A[" << i << "] = " << A[i];
```

This example uses a *range*, just like the example of reversing a `vector`: the first argument to reverse is a pointer to the beginning of the range, and the second argument points one element past the end of the range. This range is denoted `[A, A + 6)`; the asymmetrical notation is a reminder that the two endpoints are different, that the first is the beginning of the range and the second is *one past* the end of the range.

Iterators

In the example of reversing a C array, the arguments to `reverse` are clearly of type `double*`. What are the arguments to reverse if you are reversing a `vector`, though, or a `list`? That is, what exactly does `reverse` declare its arguments to be, and what exactly do `v.begin()` and `v.end()` return?

The answer is that the arguments to `reverse` are *iterators*, which are a generalization of pointers. Pointers themselves are iterators, which is why it is possible to reverse the elements of a C array. Similarly, `vector` declares the nested types `iterator` and `const_iterator`. In the example above, the type returned by `v.begin()` and `v.end()` is `vector<int>::iterator`. There are also some iterators, such as `istream_iterator` and `ostream_iterator`, that aren't associated with containers at all.

Iterators are the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container. Consider, for example, how to write an algorithm that performs linear search through a range. This is the STL's `find` algorithm.

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

`Find` takes three arguments: two iterators that define a range, and a value to search for in that range. It examines each iterator in the range `[first, last)`, proceeding from the beginning to the end, and stops either when it finds an iterator that points to `value` or when it reaches the end of the range.

`First` and `last` are declared to be of type `InputIterator`, and `InputIterator` is a template parameter. That is, there isn't actually any type called `InputIterator`: when you call `find`, the compiler substitutes the actual type of the arguments for the formal type parameters `InputIterator` and `T`. If the first two arguments to `find` are of type `int*` and the third is of type `int`, then it is as if you had called the following function.

```
int* find(int* first, int* last, const int& value) {
   while (first != last && *first != value) ++first;
   return first;
}
```

Concepts and Modeling

One very important question to ask about any template function, not just about STL algorithms, is what the set of types is that may correctly be substituted for the formal template parameters. Clearly, for example, `int*` or `double*` may be substituted for `find`'s formal template parameter `InputIterator`. Equally clearly, `int` or `double` may not: `find` uses the expression `*first`, and the dereference operator makes no sense for an object of type `int` or of type `double`. The basic answer, then, is that `find` implicitly defines a set of requirements on types, and that it may be instantiated with any type that satisfies those requirements. Whatever type is substituted for `InputIterator` must provide certain operations: it must be possible to compare two objects of that type for equality, it must be possible to increment an object of that type, it must be possible to dereference an object of that type to obtain the object that it points to, and so on.

`Find` isn't the only STL algorithm that has such a set of requirements; the arguments to `for_each` and `count`, and other algorithms, must satisfy the same requirements. These requirements are sufficiently important that we give them a name: we call such a set of type requirements a *concept*, and we call this particular concept **Input Iterator**. We say that a type *conforms to a concept*, or that it *is a model of a concept*, if it satisfies all of those requirements. We say that `int*` is a model of

Input Iterator because `int*` provides all of the operations that are specified by the **Input Iterator** requirements.

Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation: the author of `find` only has to consider the interface specified by the concept **Input Iterator**, rather than the implementation of every possible type that conforms to that concept. Similarly, if you want to use `find`, you need only to ensure that the arguments you pass to it are models of **Input Iterator.** This is the reason why `find` and `reverse` can be used with `list`s, `vector`s, C arrays, and many other types: programming in terms of concepts, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

Refinement

Input Iterator is, in fact, a rather weak concept: that is, it imposes very few requirements. An **Input Iterator** must support a subset of pointer arithmetic (it must be possible to increment an **Input Iterator** using prefix and postfix `operator++`), but need not support all operations of pointer arithmetic. This is sufficient for `find`, but some other algorithms require that their arguments satisfy additional requirements. `Reverse`, for example, must be able to decrement its arguments as well as increment them; it uses the expression `--last`. In terms of concepts, we say that `reverse`'s arguments must be models of **Bidirectional Iterator** rather than **Input Iterator**.

The **Bidirectional Iterator** concept is very similar to the **Input Iterator** concept: it simply imposes some additional requirements. The types that are models of **Bidirectional Iterator** are a subset of the types that are models of**Input Iterator**: every type that is a model of **Bidirectional Iterator** is also a model of **Input Iterator**. `Int*`, for example, is both a model of **Bidirectional Iterator** and a model of **Input Iterator**, but `istream_iterator`, is only a model of **Input Iterator**: it does not conform to the more stringent **Bidirectional Iterator** requirements.

We describe the relationship between **Input Iterator** and **Bidirectional Iterator** by saying that **Bidirectional Iterator** is a *refinement* of **Input Iterator**. Refinement of concepts is very much like inheritance of C++ classes; the main reason we use a different word, instead of just calling it "inheritance", is to emphasize that refinement applies to concepts rather than to actual types.

There are actually three more iterator concepts in addition to the two that we have already discussed: the five iterator concepts are **Output Iterator**, **Input Iterator**, **Forward Iterator**, **Bidirectional Iterator**, and **Random Access Iterator; Forward Iterator** is a refinement of **Input Iterator**, **Bidirectional Iterator** is a refinement of **Forward Iterator**, and **Random Access Iterator** is a refinement of **Bidirectional Iterator**. (**Output Iterator** is related to the other four concepts, but it is not part of the hierarchy of refinement: it is not a refinement of any of the other iterator concepts, and none of the other iterator concepts are refinements of it.) The *Iterator Overview* has more information about iterators in general.

Container classes, like iterators, are organized into a hierarchy of concepts. All containers are models of the concept **Container**; more refined concepts, such as **Sequence** and **Associative Container**, describe specific types of containers.

Other parts of the STL

If you understand algorithms, iterators, and containers, then you understand almost everything there is to know about the STL. The STL does, however, include several other types of components.

First, the STL includes several *utilities*: very basic concepts and functions that are used in many different parts of the library. The concept**Assignable**, for example, describes types that have assignment operators and copy constructors; almost all STL classes are models of **Assignable**, and almost all STL algorithms require their arguments to be models of **Assignable**.

Second, the STL includes some low-level mechanisms for allocating and deallocating memory. *Allocators* are very specialized, and you can safely ignore them for almost all purposes.

Finally, the STL includes a large collection of *function objects*, also known as *functors*. Just as iterators are a generalization of pointers, function objects are a generalization of functions: a function object is anything that you can call using the ordinary function call syntax. There are several different concepts relating to function objects, including **Unary Function** (a function object that takes a single argument, *i.e.* one that is called as `f(x)`) and **Binary Function** (a function object that takes two arguments, *i.e.* one that is called as `f(x, y)`). Function objects are an important part of generic programming because they allow abstraction not only over the types of objects, but also over the operations that are being performed.

Chapter 6

How to use the STL documentation

This documentation assumes a general familiarity with C++, especially with C++ templates. Additionally, you should read Introduction to the Standard Template Library before proceeding to the pages that describe individual components: the introductory page defines several terms that are used throughout the documentation.

Classification of STL components

The STL components are divided into six broad categories on the basis of functionality: *Containers*, *Iterators*, *Algorithms*, *Function Objects*, *Utilities*, and *Allocators*; these categories are defined in the Introduction, and the Table of Contents is organized according to them.

The STL documentation contains two indices. One of them, the Main Index, lists all components in alphabetical order. The other, the Divided Index, contains a separate alphabetical listing for each category. The Divided Index includes one category that is not present in the Table of Contents: *Adaptors*. An adaptor is a class or a function that transforms one interface into a different one. The reason that adaptors don't appear in the Table of Contents is that no component is merely an adaptor, but always an adaptor and something else; `stack`, for example, is a container and an adaptor. Accordingly, `stack` appears in two different places in the Divided Index. There are several other components that appear in the Divided Index in more than one place.

The STL documentation classifies components in two ways.

1. *Categories* are a classification by functionality. The categories are:

- Container
- Iterator
- Algorithm
- Function Object
- Utility
- Adaptor
- Allocator.

2. *Component types* are a structural classification: one based on what kind of C++ entity (if any) a component is. The component types are:

   - Type (*i.e.* a `struct` or `class`)
   - Function
   - Concept (as defined in the Introduction).

These two classification schemes are independent, and each of them applies to every STL component; `vector`, for example, is a *type* whose category is *Containers*, and **Forward Iterator** is a *concept* whose category is *Iterators*.

Both of these classification schemes appear at the top of every page that documents an STL component. The upper left corner identifies the the component's category as *Containers*,*Iterators*, *Algorithms*, *Function Objects*, *Utilities*, *Adaptors*, or *Allocators*, and the upper right corner identifies the component as a *type*, a *function*, or a *concept*.

Using the STL documentation

The STL is a *generic* library: almost every class and function is a template. Accordingly, one of the most important purposes of the STL documentation is to provide a clear description of which types may be used to instantiate those templates. As described in the Introduction, a *concept* is a generic set of requirements that a type must satisfy: a type is said to be a *model of* a concept if it satisfies all of that concept's requirements.

Concepts are used very heavily in the STL documentation, both because they directly express type requirements, and because they are a tool for organizing types conceptually. (For example, the fact that `ostream_iterator` and `insert_iterator` are both models of **Output Iterator** is an important statement about what those two classes have in common.) Concepts are used for the documentation of both *types* and *functions*.

The format of a *concept* page

A page that documents a *concept* has the following sections.

- **Summary:** A description of the concept's purpose.
- **Refinement of:** A list of other concepts that this concept *refines*, with links to those concepts.
- **Associated types:** A concept is a set of requirements on some type. Frequently, however, some of those requirements involve some other type. For example, one of the**Unary Function** requirements is that a **Unary Function** must have an *argument type*; if `F` is a type that models **Unary Function** and `f` is an object of type `F`, then, in the expression `f(x)`, `x` must be of `F`'s argument type. If a concept does have any such associated types, then they are defined in this section.
- **Notation**: The next three sections, **definitions**, **valid expressions**, and **expression semantics**, present expressions involving types that model the concept being defined. This section defines the meaning of the variables and identifiers used in those expressions.
- **Definitions**: Some concepts, such as **LessThan Comparable**, use specialized terminology. If a concept requires any such terminology, it is defined in this section.
- **Valid Expressions**: A type that models a concept is required to support certain operations. In most cases, it doesn't make sense to describe this in terms of specific functions or member functions: it doesn't make any difference, for example, whether a type that models **Input Iterator** uses a global function or a member function to provide `operator++`. This section lists the expressions that a type modeling this concept must support. It includes any special requirements (if any) on the types of the expression's operands, and the expression's return type (if any).
- **Expression Semantics:** The previous section, **valid expressions**, lists which expressions involving a type must be supported; it doesn't, however, define the meaning of those expressions. This section does: it lists the semantics, preconditions, and postconditions for the expressions defined in the previous section.
- **Complexity Guarantees**: In some cases, the run-time complexity of certain operations is an important part of a concept's requirements. For example, one of the most significant distinctions between a **Bidirectional Iterator** and a **Random Access Iterator** is that, for random access iterators, expressions like `p + n` take constant time. Any such requirements on run-time complexity are listed in this section.
- **Invariants:** Many concepts require that some property is always true for objects of a type that models the concept being defined. For example, **LessThan**

Comparable imposes the requirement of *transitivity*: if `x < y` and `y < z`, then `x < z`. Some such properties are "axioms" (that is, they are independent of any other requirements) and some are "theorems" (that is, they follow either from requirements in the **expression semantics** section or from other requirements in the **invariants** section).

- **Models**: A list of examples of types that are models of this concept. Note that this list is not intended to be complete: in most cases a complete list would be impossible, because there are an infinite number of types that could model the concept.
- **Notes**: Footnotes (if any) that are referred to by other parts of the page.
- **See Also**: Links to other related pages.

The format of a *type* page

A page that documents a *type* has the following sections.

- **Description**. A summary of the type's properties.
- **Example of use**: A code fragment involving the type.
- **Definition**: A link to the source code where the type is defined.
- **Template parameters**: Almost all STL structs and classes are templates. This section lists the name of each template parameter, its purpose, and its default value (if any).
- **Model of**: A list of the concepts that this type is a model of, and links to those concepts. Note that a type may be a model of more than one concept: `vector`, for example, is a model of both **Random Access Container** and **Back Insertion Sequence**. If a type is a model of two different concepts, that simply means that it satisfies the requirements of both.
- **Type requirements**: The template parameters of a class template usually must satisfy a set of requirements. Many of these can simply be expressed by listing which concept a template parameter must conform to, but some type requirements are slightly more complicated, and involve a relationship between two different template parameters.
- **Public base classes**: If this class inherits from any other classes, they are listed in this section.
- **Members**: A list of this type's nested types, member functions, member variables, and associated non-member functions. In most cases these members are simply listed, rather than defined: since the type is a model of some concept, detailed definitions aren't usually necessary. For example, `vector` is a model of **Container**, so the description of the member function `begin()` in the **Container** page applies to `vector`, and there is no need to repeat it in the

vector page. Instead, the **Members** section provides a very brief description of each member and a link to whatever page defines that member more fully.

- **New Members:** A type might have some members that are not part of the requirements of any of the concepts that it models. For example, vector has a member function called `capacity()`, which is not part of the **Random Access Container** or **Back Insertion Sequence** requirements. These members are defined in the **New members** section.
- **Notes**: Footnotes (if any) that are referred to by other parts of the page.
- **See Also**: Links to other related pages.

The format of a *function* page

A page that documents a *function* has the following sections.

- **Prototype:** the function's declaration.
- **Description:** A summary of what the function does.
- **Definition**: A link to the source code where the function is defined.
- **Requirements on types:** Most functions in the STL are function templates. This section lists the requirements that must be satisfied by the function's template parameters. Sometimes the requirements can simply be expressed by listing which concept a template parameter must conform to, but sometimes they are more complicated and involve a relationship between two different template parameters. In the case of `find`, for example, the requirements are that the parameter `InputIterator` is a model of **Input Iterator**, that the parameter `EqualityComparable` is a model of **Equality Comparable**, and that comparison for equality is possible between objects of type `EqualityComparable` and objects of `InputIterator`'s value types.
- **Preconditions:** Functions usually aren't guaranteed to yield a well-defined result for any possible input, but only for valid input; it is an error to call a function with invalid input. This section describes the conditions for validity.
- **Complexity:** Guarantees on the function's run-time complexity. For example, `find`'s run-time complexity is linear in the length of the input range.
- **Example of use:** A code fragment that illustrates how to use the function.
- **Notes**: Footnotes (if any) that are referred to by other parts of the page.
- **See Also**: Links to other related pages.

Chapter 7

Containers

7.1 Concepts

7.1.1 General concepts

Container

Description

A Container is an object that stores other objects (its *elements*), and that has methods for accessing its elements. In particular, every type that is a model of Container has an associated iterator type that can be used to iterate through the Container's elements. There is no guarantee that the elements of a Container are stored in any definite order; the order might, in fact, be different upon each iteration through the Container. Nor is there a guarantee that more than one iterator into a Container may be active at any one time. (Specific types of Containers, such as Forward Container, do provide such guarantees.) A Container "owns" its elements: the lifetime of an element stored in a container cannot exceed that of the Container itself.

Refinement of

Assignable

Associated types

Value type	`X::value_type`	The type of the object stored in a container. The value type must be Assignable, but need not be DefaultConstructible.
Iterator type	`X::iterator`	The type of iterator used to iterate through a container's elements. The iterator's value type is expected to be the container's value type. A conversion from the iterator type to the const iterator type must exist. The iterator type must be an input iterator.
Const iterator type	`X::const_iterator`	A type of iterator that may be used to examine, but not to modify, a container's elements.
Reference type	`X::reference`	A type that behaves as a reference to the container's value type.
Const reference type	`X::const_reference`	A type that behaves as a const reference to the container's value type.
Pointer type	`X::pointer`	A type that behaves as a pointer to the container's value type.
Distance type	`X::difference_type`	A signed integral type used to represent the distance between two of the container's iterators. This type must be the same as the iterator's distance type.
Size type	`X::size_type`	An unsigned integral type that can represent any nonnegative value of the container's distance type.

Notation

`X`	A type that is a model of Container
`a`, `b`	Object of type `X`
`T`	The value type of `X`

Definitions

The *size* of a container is the number of elements it contains. The size is a nonnegative number. The *area* of a container is the total number of bytes that it occupies. More specifically, it is the sum of the elements' areas plus whatever overhead is associated with the container itself. If a container's value type `T` is a simple type (as opposed to a container type), then the container's area is bounded above by a constant times the container's size times `sizeof(T)`. That is, if `a` is a container with a simple value type, then `a`'s area is `O(a.size())`. A *variable sized* container is one that provides methods for inserting and/or removing elements; its size may vary during a container's lifetime. A *fixed size* container is one where the size is constant throughout the container's lifetime. In some fixed-size container types, the size is determined at compile time.

Valid expressions

In addition to the expressions defined in Assignable, EqualityComparable, and LessThanComparable, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Beginning of range	`a.begin()`		`iterator` if `a` is mutable, `const_iterator` otherwise
End of range	`a.end()`		`iterator` if `a` is mutable, `const_iterator` otherwise
Size	`a.size()`		`size_type`
Maximum size	`a.max_size()`		`size_type`
Empty container	`a.empty()`		Convertible to `bool`
Swap	`a.swap(b)`		`void`

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Assignable, Equality Comparable, or LessThan Comparable

Name	Expression	Precondition	Semantics	Postcondition
Copy constructor	`X(a)`			`X().size() == a.size()`. `X()` contains a copy of each of `a`'s elements.
Copy constructor	`X b(a);`			`b.size() == a.size()`. `b` contains a copy of each of `a`'s elements.
Assignment operator	`b = a`			`b.size() == a.size()`. `b` contains a copy of each of `a`'s elements.
Destructor	`a.~X()`		Each of `a`'s elements is destroyed, and memory allocated for them (if any) is deallocated.	
Beginning of range	`a.begin()`		Returns an iterator pointing to the first element in the container.	`a.begin()` is either dereferenceable or past-the-end. It is past-the-end if and only if `a.size() == 0`.
End of range	`a.end()`		Returns an iterator pointing one past the last element in the container.	`a.end()` is past-the-end.
Size	`a.size()`		Returns the size of the container, that is, its number of elements.	`a.size() >= 0 && a.size() <= max_size()`
Maximum size	`a.max_size()`		Returns the largest size that this container can ever have.	`a.max_size() >= 0 && a.max_size() >= a.size()`
Empty container	`a.empty()`		Equivalent to `a.size() == 0`. (But possibly faster.)	
Swap	`a.swap(b)`		Equivalent to `swap(a,b)`	

Complexity guarantees

The copy constructor, the assignment operator, and the destructor are linear in the container's size. `begin()` and `end()` are amortized constant time. `size()` is linear in the container's size. `max_size()` and `empty()` are amortized constant time. If you are testing whether a container is empty, you should always write `c.empty()`

instead of `c.size() == 0`. The two expressions are equivalent, but the former may be much faster. `swap()` is amortized constant time.

Invariants

Valid range	For any container `a`, `[a.begin(), a.end())` is a valid range.
Range size	`a.size()` is equal to the distance from `a.begin()` to `a.end()`.
Completeness	An algorithm that iterates through the range `[a.begin(), a.end())` will pass through every element of `a`.

Models

- vector

Notes

The fact that the lifetime of elements cannot exceed that of of their container may seem like a severe restriction. In fact, though, it is not. Note that pointers and iterators are objects; like any other objects, they may be stored in a container. The container, in that case, "owns" the pointers themselves, but not the objects that they point to. This expression must be a `typedef`, that is, a synonym for a type that already has some other name. This may either be a `typedef` for some other type, or else a unique type that is defined as a nested class within the class `X`. A container's iterator type and const iterator type may be the same: there is no guarantee that every container must have an associated mutable iterator type. For example, `set` defines `iterator` and `const_iterator` to be the same type. It is required that the reference type has the same semantics as an ordinary C++ reference, but it need not actually be an ordinary C++ reference. Some implementations, for example, might provide additional reference types to support non-standard memory models. Note, however, that "smart references" (user-defined reference types that provide additional functionality) are not a viable option. It is impossible for a user-defined type to have the same semantics as C++ references, because the C++ language does not support redefining the member access operator (`operator.`). As in the case of references , the pointer type must have the same semantics as C++ pointers but need not actually be a C++ pointer. "Smart pointers," however, unlike "smart references", are possible. This is because it is possible for user-defined types to define the dereference operator and the pointer member access operator, `operator*` and `operator->`. The iterator type need only be an *input iterator*, which provides a very weak set of guarantees; in particular, all algorithms on input iterators must be "single pass". It follows that only a single iterator into a container may be active at any one time. This restriction is removed in Forward Container. In the case of a fixed-size container, `size() == max_size()`. For any Assignable type, swap can be defined in terms of assignment. This requires three assignments, each of which, for a container type, is linear in the container's size. In a sense, then, `a.swap(b)` is redundant. It exists solely for the sake of efficiency: for many containers, such as vector and list, it is possible to implement `swap` such that its run-time complexity is constant rather than linear. If this is possible for some container type `X`, then the template specialization `swap(X&, X&)` can simply be written in terms of `X::swap(X&)`. The implication of this is that `X::swap(X&)` should **only** be defined if there exists such a constant-time implementation. Not every container class `X` need have such a member function, but if the member function exists at all then it is guaranteed to be amortized constant time. For many containers, such as `vector` and `deque`, `size` is *O(1)*. This satisfies the requirement that it be *O(N)*. Although `[a.begin(), a.end())` must be a valid range, and must include every element in the container, the order in which the elements appear in that range is unspecified. If you iterate through a container twice, it is not guaranteed that the order will be the same both times. This restriction is removed in Forward Container.

See also

The Iterator overview, Input Iterator, Sequence

Forward Container

Description

A Forward Container is a Container whose elements are arranged in a definite order: the ordering will not change spontaneously from iteration to iteration. The requirement of a definite ordering allows the definition of element-by-element equality (if the container's element type is Equality Comparable) and of lexicographical ordering (if the container's element type is LessThan Comparable). Iterators into a Forward Container satisfy the forward iterator requirements; consequently, Forward Containers support multipass algorithms and allow multiple iterators into the same container to be active at the same time.

Refinement of

Container, EqualityComparable, LessThanComparable

Associated types

No additional types beyond those defined in Container. However, the requirements for the iterator type are strengthened: the iterator type must be a model of Forward Iterator.

Notation

`X`	A type that is a model of Forward Container
`a, b`	Object of type `X`
`T`	The value type of `X`

Definitions

Valid expressions

In addition to the expressions defined in Container, EqualityComparable, and LessThanComparable, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Equality	`a == b`	`T` is EqualityComparable	Convertible to `bool`
Inequality	`a != b`	`T` is EqualityComparable	Convertible to `bool`
Less	`a < b`	`T` is LessThanComparable	Convertible to `bool`
Greater	`a > b`	`T` is LessThanComparable	Convertible to `bool`
Less or equal	`a <= b`	`T` is LessThanComparable	Convertible to `bool`
Greater or equal	`a >= b`	`T` is LessThanComparable	Convertible to `bool`

Expression semantics

Semantics of an expression is defined only where it is not defined in Container, EqualityComparable, or LessThanComparable, or where there is additional information.

Name	Expression	Precondition	Semantics	Postcondition
Equality	`a == b`		Returns `true` if `a.size() == b.size()` and if each element of `a` compares equal to the corresponding element of `b`. Otherwise returns `false`.	
Less	`a < b`		Equivalent to `lexicographical_compare(a,b)`	

Complexity guarantees

The equality and inequality operations are linear in the container's size.

Invariants

Ordering	Two different iterations through a forward container will access its elements in the same order, providing that there have been no intervening mutative operations.

Models

- vector
- list
- slist
- deque
- set
- map
- multiset

Notes

See also

The iterator overview, Forward Iterator,

Sequence

Reversible Container

Description

A Reversible Container is a Forward Container whose iterators are Bidirectional Iterators. It allows backwards iteration through the container.

Refinement of

Forward Container

Associated types

Two new types are introduced. In addition, the iterator type and the const iterator type must satisfy a more stringent requirement than for a Forward Container. The iterator and reverse iterator types must be Bidirectional Iterators, not merely Forward Iterators.

Reverse iterator type	`X::reverse_iterator`	A Reverse Iterator adaptor whose base iterator type is the container's iterator type. Incrementing an object of type `reverse_iterator` moves backwards through the container: the Reverse Iterator adaptor maps `operator++` to `operator--`.
Const reverse iterator type	`X::const_reverse_iterator`	A Reverse Iterator adaptor whose base iterator type is the container's const iterator type.

Notation

`X`	A type that is a model of Reversible Container
`a`, `b`	Object of type `X`

Definitions

Valid expressions

In addition to the expressions defined in Forward Container, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Beginning of range	`a.rbegin()`		`reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise
End of range	`a.rend()`		`reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise

Expression semantics

Semantics of an expression is defined only where it is not defined in Forward Container, or where there is additional information.

Name	Expression	Precondition	Semantics	Postcondition
Beginning of reverse range	`a.rbegin()`		Equivalent to `X::reverse_iterator(a.end())`.	`a.rbegin()` is dereferenceable or past-the-end. It is past-the-end if and only if `a.size() == 0`.
End of reverse range	`a.rend()`		Equivalent to `X::reverse_iterator (a.begin())`.	`a.end()` is past-the-end.

Complexity guarantees

The run-time complexity of `rbegin()` and `rend()` is amortized constant time.

Invariants

Valid range	`[a.rbegin(), a.rend())` is a valid range.
Equivalence of ranges	The distance from `a.begin()` to `a.end()` is the same as the distance from `a.rbegin()` to `a.rend()`.

Models

- vector
- list
- deque

Notes

A Container's iterator type and const iterator type may be the same type: a container need not provide mutable iterators. It follows from this that the reverse iterator type and the const reverse iterator type may also be the same.

See also

The Iterator overview, Bidirectional Iterator, Sequence

Random Access Container

Description

A Random Access Container is a Reversible Container whose iterator type is a Random Access Iterator. It provides amortized constant time access to arbitrary elements.

Refinement of

Reversible Container

Associated types

No additional types beyond those defined in Reversible Container. However, the requirements for the iterator type are strengthened: it must be a Random Access Iterator.

Notation

`X`	A type that is a model of Random Access Container
`a`, `b`	Object of type `X`
`T`	The value type of `X`

Definitions

Valid expressions

In addition to the expressions defined in Reversible Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Element access	`a[n]`	`n` is convertible to `size_type`	`reference` if `a` is mutable, `const_reference` otherwise

Expression semantics

Semantics of an expression is defined only where it is not defined in Reversible Container, or where there is additional information.

Name	Expression	Precondition	Semantics	Post-condition
Element access	`a[n]`	`0 <= n < a.size()`	Returns the `n`th element from the beginning of the container.	

Complexity guarantees

The run-time complexity of element access is amortized constant time.

Invariants

Element access	The element returned by `a[n]` is the same as the one obtained by incrementing `a.begin()` `n` times and then dereferencing the resulting iterator.

Models

- vector
- deque

Notes

See also

The Iterator overview, Random Access Iterator, Sequence

7.1.2 Sequences

Sequence

Description

A Sequence is a variable-sized Container whose elements are arranged in a strict linear order. It supports insertion and removal of elements.

Refinement of

Forward Container, Default Constructible

Associated types

None, except for those of Forward Container.

Notation

`X`	A type that is a model of Sequence
`a`, `b`	Object of type `X`
`T`	The value type of `X`
`t`	Object of type `T`
`p`, `q`	Object of type `X::iterator`
`n`	Object of a type convertible to `X::size_type`

Definitions

If `a` is a Sequence, then `p` is a *valid iterator in a* if it is a valid (nonsingular) iterator that is reachable from `a.begin()`. If `a` is a Sequence, then `[p, q)` is a *valid range in a* if `p` and `q` are valid iterators in `a` and if `q` is reachable from `p`.

Valid expressions

In addition to the expressions defined in Forward Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Fill constructor	`X(n, t)`		`X`
Fill constructor	`X a(n, t);`		
Default fill constructor	`X(n)`	`T` is DefaultConstructible.	`X`
Default fill constructor	`X a(n);`	`T` is DefaultConstructible.	
Range constructor	`X(i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `T`	`X`
Range constructor	`X a(i, j);`	`i` and `j` are Input Iterators whose value type is convertible to `T`	
Front	`a.front()`		`reference` if `a` is mutable, `const_reference` otherwise.
Insert	`a.insert(p, t)`		`X::iterator`
Fill insert	`a.insert(p, n, t)`	`a` is mutable	`void`
Range insert	`a.insert(p, i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `T` . `a` is mutable	`void`
Erase	`a.erase(p)`	`a` is mutable	`iterator`
Range erase	`a.erase(p,q)`	`a` is mutable	`iterator`
Clear	`a.clear()`	`a` is mutable	`void`
Resize	`a.resize(n, t)`	`a` is mutable	`void`
Resize	`a.resize(n)`	`a` is mutable	`void`

Expression semantics

Semantics of an expression is defined only where it is not defined in Forward Container, or where it differs.

Name	Expression	Precondition	Semantics	Postcondition
Fill constructor	`X(n, t)`	`n >= 0`	Creates a sequence with `n` copies of `t`	`size() == n`. Every element is a copy of `t`.
Fill constructor	`X a(n, t);`	`n >= 0`	Creates a sequence with `n` copies of `t`	`a.size() == n`. Every element of `a` is a copy of `t`.
Default fill constructor	`X(n)`	`n >= 0`	Creates a sequence of `n` elements initialized to a default value.	`size() == n`. Every element is a copy of `T()`.
Default fill constructor	`X a(n, t);`	`n >= 0`	Creates a sequence with `n` elements initialized to a default value.	`a.size() == n`. Every element of `a` is a copy of `T()`.
Default constructor	`X a;` or `X()`		Equivalent to `X(0)`.	`size() == 0`.
Range constructor	`X(i, j)`	`[i,j)` is a valid range.	Creates a sequence that is a copy of the range `[i,j)`	`size()` is equal to the distance from `i` to `j`. Each element is a copy of the corresponding element in the range `[i,j)`.
Range constructor	`X a(i, j);`	`[i,j)` is a valid range.	Creates a sequence that is a copy of the range `[i,j)`	`a.size()` is equal to the distance from `i` to `j`. Each element in `a` is a copy of the corresponding element in the range `[i,j)`.
Front	`a.front()`	`!a.empty()`	Equivalent to `*(a.first())`	
Insert	`a.insert(p, t)`	`p` is a valid iterator in `a`. `a.size() < a.max_size()`	A copy of `t` is inserted before `p`.	`a.size()` is incremented by 1. `*(a.insert(p,t))` is a copy of `t`. The relative order of elements already in the sequence is unchanged.

Name	Expression	Precondition	Semantics	Postcondition
Fill insert	`a.insert(p, n, t)`	`p` is a valid iterator in `a`. `n >= 0 && a.size() + n <= a.max_size()`.	`n` copies of `t` are inserted before `p`.	`a.size()` is incremented by n. The relative order of elements already in the sequence is unchanged.
Range insert	`a.insert(p, i, j)`	`[i,j)` is a valid range. `a.size()` plus the distance from `i` to `j` does not exceed `a.max_size()`.	Inserts a copy of the range `[i,j)` before `p`.	`a.size()` is incremented by the distance from `i` to `j`. The relative order of elements already in the sequence is unchanged.
Erase	`a.erase(p)`	`p` is a dereferenceable iterator in `a`.	Destroys the element pointed to by `p` and removes it from `a`.	`a.size()` is decremented by 1. The relative order of the other elements in the sequence is unchanged. The return value is an iterator to the element immediately following the one that was erased.
Range erase	`a.erase(p,q)`	`[p,q)` is a valid range in `a`.	Destroys the elements in the range `[p,q)` and removes them from `a`.	`a.size()` is decremented by the distance from `i` to `j`. The relative order of the other elements in the sequence is unchanged. The return value is an iterator to the element immediately following the ones that were erased.
Clear	`a.clear()`		Equivalent to `a.erase (a.begin(), a.end())`	

Name	Expression	Precondition	Semantics	Postcondition
Resize	`a.resize(n, t)`	`n <= a.max_size()`	Modifies the container so that it has exactly `n` elements, inserting elements at the end or erasing elements from the end if necessary. If any elements are inserted, they are copies of `t`. If `n > a.size()`, this expression is equivalent to `a.insert(a.end(), n - size(), t)`. If `n < a.size()`, it is equivalent to `a.erase(a.begin() + n, a.end())`.	`a.size() == n`
Resize	`a.resize(n)`	`n <= a.max_size()`	Equivalent to `a.resize(n, T())`.	`a.size() == n`

Complexity guarantees

The fill constructor, default fill constructor, and range constructor are linear. Front is amortized constant time. Fill insert, range insert, and range erase are linear. The complexities of single-element insert and erase are sequence dependent.

Invariants

Models

- vector
- deque
- list
- slist

Notes

At present (early 1998), not all compilers support "member templates". If your compiler supports member templates then `i` and `j` may be of any type that conforms to the Input Iterator requirements. If your compiler does not yet support member templates, however, then `i` and `j` must be of type `const T*` or of type `X::const_iterator`. Note that `p` equal to `a.begin()` means to insert something at the beginning of `a` (that is, before any elements already in `a`), and `p` equal to `a.end()` means to append something to the end of `a`. **Warning**: there is no guarantee that a valid iterator on `a` is still valid after an insertion or an erasure. In some cases iterators do remain valid, and in other cases they do not. The details are different for each sequence class. `a.insert(p, n, t)` is guaranteed to be no slower then calling `a.insert(p, t)` `n` times. In some cases it is significantly faster. Vector is usually preferable to deque and list. Deque is useful in the case of frequent insertions at both the beginning and end of the sequence, and list and slist are useful in the case of frequent insertions in the middle of the sequence. In almost all other situations, vector is more efficient.

See also

Container, Forward Container, Associative Container, Front Insertion Sequence, Back Insertion Sequence, `vector`, `deque`, `list`, `slist`

Front Insertion Sequence

Description

A Front Insertion Sequence is a Sequence where it is possible to insert an element at the beginning, or to access the first element, in amortized constant time. Front Insertion Sequences have special member functions as a shorthand for those operations.

Refinement of

Sequence

Associated types

None, except for those of Sequence.

Notation

`X`	A type that is a model of Front Insertion Sequence
`a`	Object of type `X`
`T`	The value type of `X`
`t`	Object of type `T`

Definitions

Valid expressions

In addition to the expressions defined in Sequence, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Front	`a.front()`		`reference` if `a` is mutable, otherwise `const_reference`.
Push front	`a.push_front(t)`	`a` is mutable.	`void`
Pop front	`a.pop_front()`	`a` is mutable.	`void`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Front	`a.front()`	`!a.empty()`	Equivalent to `*(a.begin())`.	
Push front	`a.push_front(t)`		Equivalent to `a.insert (a.begin(), t)`	`a.size` is incremented by 1. `a.front()` is a copy of `t`.
Pop front	`a.pop_front()`	`!a.empty()`	Equivalent to `a.erase (a.begin())`	`a.size()` is decremented by 1.

Complexity guarantees

Front, push front, and pop front are amortized constant time.

Invariants

Symmetry of push and pop	`push_front()` followed by `pop_front()` is a null operation.

Models

- list
- deque

Notes

Front is actually defined in Sequence, since it is always possible to implement it in amortized constant time. Its definition is repeated here, along with push front and pop front, in the interest of clarity. This complexity guarantee is the only reason that `front()`, `push_front()`, and `pop_front()` are defined: they provide no additional functionality. Not every sequence must define these operations, but it is guaranteed that they are efficient if they exist at all.

See also

Container, Sequence, Back Insertion Sequence, `deque`, `list`, `slist`

Back Insertion Sequence

Description

A Back Insertion Sequence is a Sequence where it is possible to append an element to the end, or to access the last element, in amortized constant time. Back Insertion Sequences have special member functions as a shorthand for those operations.

Refinement of

Sequence

Associated types

None, except for those of Sequence.

Notation

`X`	A type that is a model of Back Insertion Sequence
`a`	Object of type `X`
`T`	The value type of `X`
`t`	Object of type `T`

Definitions

Valid expressions

In addition to the expressions defined in Sequence, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Back	`a.back()`		`reference` if `a` is mutable, otherwise `const_reference`.
Push back	`a.push_back(t)`	`a` is mutable.	`void`
Pop back	`a.pop_back()`	`a` is mutable.	`void`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Back	`a.back()`	`!a.empty()`	Equivalent to `*(--a.end())`.	
Push back	`a.push_back(t)`		Equivalent to `a.insert (a.end(), t)`	`a.size` is incremented by 1. `a.back()` is a copy of `t`.
Pop back	`a.pop_back()`	`!a.empty()`	Equivalent to `a.erase (--a.end())`	`a.size()` is decremented by 1.

Complexity guarantees

Back, push back, and pop back are amortized constant time.

Invariants

Symmetry of push and pop	`push_back()` followed by `pop_back()` is a null operation.

Models

- vector
- list
- deque

Notes

This complexity guarantee is the only reason that `back()`, `push_back()`, and `pop_back()` are defined: they provide no additional functionality. Not every sequence must define these operations, but it is guaranteed that they are efficient if they exist at all.

See also

Container, Sequence, Front Insertion Sequence, `vector`, `deque`, `list`

7.1.3 Associative Containers

Associative Container

Description

An Associative Container is a variable-sized Container that supports efficient retrieval of elements (values) based on keys. It supports insertion and removal of elements, but differs from a Sequence in that it does not provide a mechanism for inserting an element at a specific position. As with all containers, the elements in an Associative Container are of type `value_type`. Additionally, each element in an Associative Container has a *key*, of type `key_type`. In some Associative Containers, Simple Associative Containers, the `value_type` and `key_type` are the same: elements are their own keys. In others, the key is some specific part of the value. Since elements are stored according to their keys, it is essential that the key associated with each element is immutable. In Simple Associative Containers this means that the elements themselves are immutable, while in other types of Associative Containers, such as Pair Associative Containers, the elements themselves are mutable but the part of an element that is its key cannot be modified. This means that an Associative Container's value type is not Assignable. The fact that the value type of an Associative Container is not Assignable has an important consequence: associative containers cannot have mutable iterators. This is simply because a mutable iterator (as defined in the Trivial Iterator requirements) must allow assignment. That is, if `i` is a mutable iterator and `t` is an object of `i`'s value type, then `*i = t` must be a valid expression. In Simple Associative Containers, where the elements are the keys, the elements are completely immutable; the nested types `iterator` and `const_iterator` are therefore the same. Other types of associative containers, however, do have mutable elements, and do provide iterators through which elements can be modified. Pair Associative Containers, for example, have two different nested types `iterator` and `const_iterator`. Even in this case, `iterator` is not a mutable iterator: as explained above, it does not provide the expression `*i = t`. It is, however, possible to modify an element through such an iterator: if, for example, `i` is of type `map<int, double>`, then `(*i).second = 3` is a valid expression. In some associative containers, Unique Associative Containers, it is guaranteed that no two elements have the same key. In other associative containers, Multiple Associative Containers, multiple elements with the same key are permitted.

Refinement of

Forward Container, Default Constructible

Associated types

One new type is introduced, in addition to the types defined in the Forward Container requirements.

Key type	`X::key_type`	The type of the key associated with `X::value_type`. Note that the key type and value type might be the same.

Notation

`X`	A type that is a model of Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`

Definitions

If `a` is an associative container, then `p` is a *valid iterator in a* if it is a valid iterator that is reachable from `a.begin()`. If `a` is an associative container, then `[p, q)` is a *valid range in a* if `[p, q)` is a valid range and `p` is a valid iterator in `a`.

Valid expressions

In addition to the expressions defined in Forward Container, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Default constructor	`X()` `X a;`		
Erase key	`a.erase(k)`		`size_type`
Erase element	`a.erase(p)`		`void`
Erase range	`a.erase(p, q)`		`void`
Clear	`a.clear()`		`void`
Find	`a.find(k)`		`iterator` if `a` is mutable, otherwise `const_iterator`
Count	`a.count(k)`		`size_type`
Equal range	`a.equal_range(k)`		`pair<iterator, iterator>` if `a` is mutable, otherwise `pair<const_iterator, const_iterator>`.

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default constructor	`X()` `X a;`		Creates an empty container.	The size of the container is `0`.
Erase key	`a.erase(k)`		Destroys all elements whose key is the same as `k`, and removes them from `a`. The return value is the number of elements that were erased, *i.e.* the old value of `a.count(k)`.	`a.size()` is decremented by `a.count(k)`. `a` contains no elements with key `k`.
Erase element	`a.erase(p)`	`p` is a dereferenceable iterator in `a`.	Destroys the element pointed to by `p`, and removes it from `a`.	`a.size()` is decremented by 1.
Erase range	`a.erase(p, q)`	`[p, q)` is a valid range in `a`.	Destroys the elements in the range `[p,q)` and removes them from `a`.	`a.size()` is decremented by the distance from `i` to `j`.
Clear	`a.clear()`		Equivalent to `a.erase(a.begin(), a.end())`	
Find	`a.find(k)`		Returns an iterator pointing to an element whose key is the same as `k`, or `a.end()` if no such element exists.	Either the return value is `a.end()`, or else the return value has a key that is the same as `k`.
Count	`a.count(k)`		Returns the number of elements in `a` whose keys are the same as `k`.	

Name	Expression	Precondition	Semantics	Postcondition
Equal range	`a.equal_range(k)`		Returns a pair `P` such that `[P.first, P.second)` is a range containing all elements in `a` whose keys are the same as `k`. If no elements have the same key as `k`, the return value is an empty range.	The distance between `P.first` and `P.second` is equal to `a.count(k)`. If `p` is a dereferenceable iterator in `a`, then either `p` lies in the range `[P.first, P.second)`, or else `*p` has a key that is not the same as `k`.

Complexity guarantees

Average complexity for erase key is at most `O(log(size()) + count(k))`. Average complexity for erase element is constant time. Average complexity for erase range is at most `O(log(size()) + N)`, where `N` is the number of elements in the range. Average complexity for count is at most `O(log(size()) + count(k))`. Average complexity for find is at most logarithmic. Average complexity for equal range is at most logarithmic.

Invariants

Contiguous storage	All elements with the same key are adjacent to each other. That is, if `p` and `q` are iterators that point to elements that have the same key, and if `p` precedes `q`, then every element in the range `[p, q)` has the same key as every other element.
Immutability of keys	Every element of an Associative Container has an immutable key. Objects may be inserted and erased, but an element in an Associative Container may not be modified in such a way as to change its key.

Models

- `set`
- `multiset`
- `map`

Notes

The reason there is no such mechanism is that the way in which elements are arranged in an associative container is typically a class invariant; elements in a Sorted Associative Container, for example, are always stored in ascending order, and elements in a Hashed Associative Container are always stored according to the hash function. It would make no sense to allow the position of an element to be chosen arbitrarily. Keys are not required to be Equality Comparable: associative containers do not necessarily use `operator==` to determine whether two keys are the same. In Sorted Associative Containers, for example, where keys are ordered by a comparison function, two keys are considered to be the same if neither one is less than the other. Note the implications of this member function: it means that if two elements have the same key, there must be no elements with different keys in between them. The requirement that elements with the same key be stored contiguously is an associative container invariant.

See also

Simple Associative Container, Pair Associative Container, Unique Associative Container, Multiple Associative Container, Sorted Associative Container, Unique Sorted Associative Container, Multiple Sorted Associative Container, Hashed Associative Container, Unique Hashed Associative Container, Multiple Hashed Associative Container.

Simple Associative Container

Description

A Simple Associative Container is an Associative Container where elements are their own keys. A key in a Simple Associative Container is not associated with any additional value.

Refinement of

Associative Container

Associated types

None, except for those described in the Associative Container requirements. Simple Associative Container, however, introduces two new type restrictions.

Key type	`X::key_type`	The type of the key associated with `X::value_type`. The types `key_type` and `value_type` must be the same type.
Iterator	`X::iterator`	The type of iterator used to iterate through a Simple Associative Container's elements. The types `X::iterator` and `X::const_iterator` must be the same type. That is, a Simple Associative Container does not provide mutable iterators.

Notation

`X`	A type that is a model of Simple Associative Container
`a`	Object of type `X`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`

Definitions

Valid expressions

None, except for those defined in the Associative Container requirements.

Expression semantics

Complexity guarantees

Invariants

Immutability of Elements	Every element of a Simple Associative Container is immutable. Objects may be inserted and erased, but not modified.

Models

- `set`
- `multiset`

Notes

This is a consequence of the Immutability of Keys invariant of Associative Container. Keys may never be modified; values in a Simple Associative Container are themselves keys, so it immediately follows that values in a Simple Associative Container may not be modified.

See also

Associative Container, Pair Associative Container

Pair Associative Container

Description

A Pair Associative Container is an Associative Container that associates a key with some other object. The value type of a Pair Associative Container is `pair<const key_type, data_type>`.

Refinement of

Associative Container

Associated types

One new type is introduced, in addition to the types defined in the Associative Container requirements. Additionally, Pair Associative Container introduces one new type restriction

Key type	`X::key_type`	The type of the key associated with `X::value_type`.
Data type	`X::data_type`	The type of the data associated with `X::value_type`. A Pair Associative Container can be thought of as a mapping from `key_type` to `data_type`.
Value type	`X::value_type`	The type of object stored in the container. The value type is required to be `pair<const key_type, data_type>`.

Notation

`X`	A type that is a model of Pair Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`d`	Object of type `X::data_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`

Definitions

Valid expressions

None, except for those defined in the Associative Container requirements.

Expression semantics

Complexity guarantees

Invariants

Models

- `map`

Notes

The value type must be `pair<const key_type, data_type>`, rather than `pair<key_type, data_type>`, because of the Associative Container invariant of key immutability. The `data_type` part of an object in a Pair Associative Container may be modified, but the `key_type` part may not be. Note the implication of this fact: a Pair Associative Container cannot provide mutable iterators (as defined in the Trivial Iterator requirements), because the value type of a mutable iterator must be Assignable, and `pair<const key_type, data_type>` is not Assignable. However, a Pair Associative Container can provide iterators that are not completely constant: iterators such that the expression `(*i).second = d` is valid.

See also

Associative Container, Simple Associative Container

Sorted Associative Container

Description

A Sorted Associative Container is a type of Associative Container. Sorted Associative Containers use an ordering relation on their keys; two keys are considered to be equivalent if neither one is less than the other. (If the ordering relation is case-insensitive string comparison, for example, then the keys "abcde" and "aBcDe" are equivalent.) Sorted Associative Containers guarantee that the complexity for most operations is never worse than logarithmic , and they also guarantee that their elements are always sorted in ascending order by key.

Refinement of

Reversible Container, Associative Container

Associated types

Two new types are introduced, in addition to the types defined in the Associative Container and Reversible Container requirements.

`X::key_compare`	The type of a Strict Weak Ordering used to compare keys. Its argument type must be `X::key_type`.
`X::value_compare`	The type of a Strict Weak Ordering used to compare values. Its argument type must be `X::value_type`, and it compares two objects of `value_type` by passing the keys associated with those objects to a function object of type `key_compare`.

Notation

`X`	A type that is a model of Sorted Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`
`c`	Object of type `X::key_compare`

Definitions

Valid expressions

In addition to the expressions defined in Associative Container and Reversible Container, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Default constructor	`X()` `X a;`		
Constructor with compare	`X(c)` `X a(c);`		
Key comparison	`a.key_comp()`		`X::key_compare`
Value comparison	`a::value_compare()`		`X::value_compare`
Lower bound	`a.lower_bound(k)`		`iterator` if `a` is mutable, otherwise `const_iterator`.
Upper bound	`a.upper_bound(k)`		`iterator` if `a` is mutable, otherwise `const_iterator`.
Equal range	`a.equal_range(k)`		`pair<iterator, iterator>` if `a` is mutable, otherwise `pair<const_iterator, const_iterator>`.

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default constructor	`X()` `X a;`		Creates an empty container, using `key_compare()` as the comparison object.	The size of the container is `0`.
Constructor with compare	`X(c)` `X a(c);`		Creates an empty container, using `c` as the comparison object.	The size of the container is `0`. `key_comp()` returns a function object that is equivalent to `c`.
Key comparison	`a.key_comp()`		Returns the key comparison object used by `a`.	
Value comparison	`a::value_compare()`		Returns the value comparison object used by `a`.	If `t1` and `t2` are objects of type `value_type`, and `k1` and `k2` are the keys associated with them, then `a.value_comp()(t1, t2)` is equivalent to `a.key_comp()(k1, k2)`.
Lower bound	`a.lower_bound(k)`		Returns an iterator pointing to the first element whose key is not less than `k`. Returns `a.end()` if no such element exists.	If `a` contains any elements that have the same key as `k`, then the return value of `lower_bound` points to the first such element.
Upper bound	`a.upper_bound(k)`		Returns an iterator pointing to the first element whose key is greater than `k`. Returns `a.end()` if no such element exists.	If `a` contains any elements that have the same key as `k`, then the return value of `upper_bound` points to one past the last such element.
Equal range	`a.equal_range(k)`		Returns a pair whose first element is `a.lower_bound(k)` and whose second element is `a.upper_bound(k)`.	

Complexity guarantees

`key_comp()` and `value_comp()` are constant time. Erase element is constant time. Erase key is `O(log(size()) + count(k))`. Erase range is `O(log(size()) + N)`, where `N` is the length of the range. Find is logarithmic. Count is `O(log(size()) + count(k))`. Lower bound, upper bound, and equal range are logarithmic.

Invariants

Definition of `value_comp`	If `t1` and `t2` are objects of type `X::value_type` and `k1` and `k2` are the keys associated with those objects, then `a.value_comp()` returns a function object such that `a.value_comp()(t1, t2)` is equivalent to `a.key_comp()(k1, k2)`.
Ascending order	The elements in a Sorted Associative Container are always arranged in ascending order by key. That is, if `a` is a Sorted Associative Container, then `is_sorted(a.begin(), a.end(), a.value_comp())` is always `true`.

Models

- `set`
- `multiset`
- `map`

Notes

This is a much stronger guarantee than the one provided by Associative Container. The guarantees in Associative Container only apply to average complexity; worst case complexity is allowed to be greater. Sorted Associative Container, however, provides an upper limit on worst case complexity. This definition is consistent with the semantics described in Associative Container. It is a stronger condition, though: if `a` contains no elements with the key `k`, then `a.equal_range(k)` returns an empty range that indicates the position where those elements would be if they did exist. The Associative Container requirements, however, merely state that the return value is an arbitrary empty range.

See also

Associative Container, Hashed Associative Container

Unique Associative Container

Description

A Unique Associative Container is an Associative Container with the property that each key in the container is unique: no two elements in a Unique Associative Container have the same key.

Refinement of

Associative Container

Associated types

None, except for those defined by Associative Container.

Notation

`X`	A type that is a model of Unique Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`

Definitions

Valid expressions

In addition to the expressions defined in Associative Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Range constructor	`X(i, j)` `X a(i, j);`	`i` and `j` are Input Iterators whose value type is convertible to `T`	
Insert element	`a.insert(t)`		`pair<X::iterator, bool>`
Insert range	`a.insert(i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `X::value_type`.	`void`
Count	`a.count(k)`		`size_type`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Range constructor	`X(i, j)` `X a(i, j);`	`[i,j)` is a valid range.	Creates an associative container that contains all of the elements in the range `[i,j)` that have unique keys.	`size()` is less than or equal to the distance from `i` to `j`.
Insert element	`a.insert(t)`		Inserts `t` into `a` if and only if `a` does not already contain an element whose key is the same as the key of `t`. The return value is a `pair P`. `P.first` is an iterator pointing to the element whose key is the same as the key of `t`. `P.second` is a `bool`: it is `true` if `t` was actually inserted into `a`, and `false` if `t` was not inserted into `a`, *i.e.* if `a` already contained an element with the same key as `t`.	`P.first` is a dereferenceable iterator. `*(P.first)` has the same key as `t`. The size of `a` is incremented by `1` if and only if `P.second` is `true`.
Insert range	`a.insert(i, j)`	`[i, j)` is a valid range.	Equivalent to `a.insert(t)` for each object `t` that is pointed to by an iterator in the range `[i, j)`. Each element is inserted into `a` if and only if `a` does not already contain an element with the same key.	The size of `a` is incremented by at most `j - i`.
Count	`a.count(k)`		Returns the number of elements in `a` whose keys are the same as `k`.	The return value is either `0` or `1`.

Complexity guarantees

Average complexity for insert element is at most logarithmic. Average complexity for insert range is at most `O(N * log(size() + N))`, where `N` is `j - i`.

Invariants

Uniqueness	No two elements have the same key. Equivalently, this means that for every object `k` of type `key_type`, `a.count(k)` returns either `0` or `1`.

Models

- `set`
- `map`

Notes

At present (early 1998), not all compilers support "member templates". If your compiler supports member templates then `i` and `j` may be of any type that conforms to the Input Iterator requirements. If your compiler does not yet support member templates, however, then `i` and `j` must be of type `const T*` or of type `X::const_iterator`.

See also

Associative Container, Multiple Associative Container, Unique Sorted Associative Container, Multiple Sorted Associative Container

Multiple Associative Container

Description

A Multiple Associative Container is an Associative Container in which there may be more than one element with the same key. That is, it is an Associative Container that does not have the restrictions of a Unique Associative Container.

Refinement of

Associative Container

Associated types

None, except for those defined by Associative Container

Notation

`X`	A type that is a model of Multiple Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`

Definitions

Valid expressions

In addition to the expressions defined in Associative Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Range constructor	`X(i, j)` `X a(i, j);`	`i` and `j` are Input Iterators whose value type is convertible to `T`	
Insert element	`a.insert(t)`		`X::iterator`
Insert range	`a.insert(i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `X::value_type`.	`void`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Range constructor	`X(i, j)` `X a(i, j);`	`[i,j)` is a valid range.	Creates an associative container that contains all elements in the range `[i,j)`.	`size()` is equal to the distance from `i` to `j`. Each element in `[i, j)` is present in the container.
Insert element	`a.insert(t)`		Inserts `t` into `a`.	The size of `a` is incremented by `1`. The value of `a.count(t)` is incremented by `a`.
Insert range	`a.insert(i, j)`	`[i, j)` is a valid range.	Equivalent to `a.insert(t)` for each object `t` that is pointed to by an iterator in the range `[i, j)`. Each element is inserted into `a`.	The size of `a` is incremented by `j - i`.

Complexity guarantees

Average complexity for insert element is at most logarithmic. Average complexity for insert range is at most `O(N * log(size() + N))`, where `N` is `j - i`.

Invariants

Models

- `multiset`

Notes

At present (early 1998), not all compilers support "member templates". If your compiler supports member templates then `i` and `j` may be of any type that conforms to the Input Iterator requirements. If your compiler does not yet support member templates, however, then `i` and `j` must be of type `const T*` or of type `X::const_iterator`.

See also

Associative Container, Unique Associative Container, Unique Sorted Associative Container, Multiple Sorted Associative Container

Unique Sorted Associative Container

Description

A Unique Sorted Associative Container is a Sorted Associative Container that is also a Unique Associative Container. That is, it is a Sorted Associative Container with the property that no two elements in the container have the same key.

Refinement of

Sorted Associative Container, Unique Associative Container

Associated types

None, except for those described in the Sorted Associative Container and Unique Associative Container requirements.

Notation

`X`	A type that is a model of Unique Sorted Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`
`c`	Object of type `X::key_compare`

Definitions

Valid expressions

In addition to the expressions defined in Sorted Associative Container and Unique Associative Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Range constructor	`X(i, j)` `X a(i, j);`	`i` and `j` are Input Iterators whose value type is convertible to `T` .	
Range constructor with compare	`X(i, j, c)` `X a(i, j, c);`	`i` and `j` are Input Iterators whose value type is convertible to `T` . `c` is an object of type `key_compare`.	
Insert with hint	`a.insert(p, t)`		`iterator`
Insert range	`a.insert(i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `X::value_type`.	`void`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Range constructor	`X(i, j)` `X a(i, j);`	`[i,j)` is a valid range.	Creates an associative container that contains all of the elements in the range `[i,j)` that have unique keys. The comparison object used by the container is `key_compare()`.	`size()` is less than or equal to the distance from `i` to `j`.
Range constructor with compare	`X(i, j, c)` `X a(i, j, c);`	`[i,j)` is a valid range.	Creates an associative container that contains all of the elements in the range `[i,j)` that have unique keys. The comparison object used by the container is `c`.	`size()` is less than or equal to the distance from `i` to `j`.
Insert with hint	`a.insert(p, t)`	`p` is a nonsingular iterator in `a`.	Inserts `t` into `a` if and only if `a` does not already contain an element whose key is equivalent to `t`'s key. The argument `p` is a hint: it points to the location where the search will begin. The return value is a dereferenceable iterator that points to the element with a key that is equivalent to that of `t`.	`a` contains an element whose key is the same as that of `t`. The size of `a` is incremented by either `1` or `0`.
Insert range	`a.insert(i, j)`	`[i, j)` is a valid range.	Equivalent to `a.insert(t)` for each object `t` that is pointed to by an iterator in the range `[i, j)`. Each element is inserted into `a` if and only if `a` does not already contain an element with an equivalent key.	The size of `a` is incremented by at most `j - i`.

Complexity guarantees

The range constructor, and range constructor with compare, are in general `O(N * log(N))`, where `N` is the size of the range. However, they are linear in `N` if the range is already sorted by `value_comp()`. Insert with hint is logarithmic in general, but it is amortized constant time if `t` is inserted immediately before `p`. Insert range is in general `O(N * log(N))`, where `N` is the size of the range. However, it is linear in `N` if the range is already sorted by `value_comp()`.

Invariants

Strictly ascending order	The elements in a Unique Sorted Associative Container are always arranged in strictly ascending order by key. That is, if `a` is a Unique Sorted Associative Container, then `is_sorted(a.begin(), a.end(), a.value_comp())` is always `true`. Furthermore, if `i` and `j` are dereferenceable iterators in `a` such that `i` precedes `j`, then `a.value_comp()(*i, *j)` is always `true`.

Models

- `map`
- `set`

Notes

At present (early 1998), not all compilers support "member templates". If your compiler supports member templates then `i` and `j` may be of any type that conforms to the Input Iterator requirements. If your compiler does not yet support member templates, however, then `i` and `j` must be of type `const T*` or of type `X::const_iterator`. This is a more stringent invariant than that of Sorted Associative Container. In a Sorted Associative Container we merely know that every element is less than or equal to its successor; in a Unique Sorted Associative Container, however, we know that it must be less than its successor.

See also

Associative Container, Sorted Associative Container, Multiple Sorted Associative Container, Hashed Associative Container

Multiple Sorted Associative Container

Description

A Multiple Sorted Associative Container is a Sorted Associative Container that is also a Multiple Associative Container. That is, it is a Sorted Associative Container with the property that any number of elements in the container may have equivalent keys.

Refinement of

Sorted Associative Container, Multiple Associative Container

Associated types

None, except for those described in the Sorted Associative Container and Multiple Associative Container requirements.

Notation

`X`	A type that is a model of Multiple Sorted Associative Container
`a`	Object of type `X`
`t`	Object of type `X::value_type`
`k`	Object of type `X::key_type`
`p`, `q`	Object of type `X::iterator`
`c`	Object of type `X::key_compare`

Definitions

Valid expressions

In addition to the expressions defined in Sorted Associative Container and Multiple Associative Container, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Range constructor	`X(i, j)` `X a(i, j);`	`i` and `j` are Input Iterators whose value type is convertible to `T` .	`X`
Range constructor with compare	`X(i, j, c)` `X a(i, j, c);`	`i` and `j` are Input Iterators whose value type is convertible to `T` . `c` is an object of type `key_compare`.	`X`
Insert with hint	`a.insert(p, t)`		`iterator`
Insert range	`a.insert(i, j)`	`i` and `j` are Input Iterators whose value type is convertible to `X::value_type`.	`void`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Range constructor	`X(i, j)` `X a(i, j);`	`[i,j)` is a valid range.	Creates an associative container that contains all of the elements in the range `[i,j)`. The comparison object used by the container is `key_compare()`.	`size()` is equal to the distance from `i` to `j`.
Range constructor with compare	`X(i, j, c)` `X a(i, j, c);`	`[i,j)` is a valid range.	Creates an associative container that contains all of the elements in the range `[i,j)`. The comparison object used by the container is `c`.	`size()` is equal to the distance from `i` to `j`.
Insert with hint	`a.insert(p, t)`	`p` is a nonsingular iterator in `a`.	Inserts `t` into `a`. The argument `p` is a hint: it points to the location where the search will begin. The return value is a dereferenceable iterator that points to the element that was just inserted.	`a` contains an element whose key is the same as that of `t`. The size of `a` is incremented by `1`.
Insert range	`a.insert(i, j)`	`[i, j)` is a valid range.	Equivalent to `a.insert(t)` for each object `t` that is pointed to by an iterator in the range `[i, j)`. Each element is inserted into `a`.	The size of `a` is incremented by `j - i`.

Complexity guarantees

The range constructor, and range constructor with compare, are in general `O(N * log(N))`, where `N` is the size of the range. However, they are linear in `N` if the range is already sorted by `value_comp()`. Insert with hint is logarithmic in general, but it is amortized constant time if `t` is inserted immediately before `p`. Insert range is in general `O(N * log(N))`, where `N` is the size of the range. However, it is linear in `N` if the range is already sorted by `value_comp()`.

Invariants

Models

- `multiset`

Notes

At present (early 1998), not all compilers support "member templates". If your compiler supports member templates then `i` and `j` may be of any type that conforms to the Input Iterator requirements. If your compiler does not yet support member templates, however, then `i` and `j` must be of type `const T*` or of type `X::const_iterator`.

See also

Associative Container, Sorted Associative Container, Unique Sorted Associative Container Hashed Associative Container

7.2 Container classes

7.2.1 Sequences

vector

Description

A `vector` is a Sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a `vector` may vary dynamically; memory management is automatic. `Vector` is the simplest of the STL container classes, and in many cases the most efficient.

Example

```
vector<int> V;
V.insert(V.begin(), 3);
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);
```

Definition

Defined in the standard header vector, and in the nonstandard backward-compatibility header vector.h.

Template parameters

Parameter	Description	Default
`T`	The vector's value type: the type of object that is stored in the vector.	
`Alloc`	The `vector`'s allocator, used for all internal memory management.	`alloc`

Model of

Random Access Container, Back Insertion Sequence.

Type requirements

None, except for those imposed by the requirements of Random Access Container and Back Insertion Sequence.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the vector.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `vector`.
`const_iterator`	Container	Const iterator used to iterate through a `vector`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `vector`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `vector`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `vector`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `vector`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `vector`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `vector`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed vector.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed vector.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed vector.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed vector.
`size_type size() const`	Container	Returns the size of the `vector`.
`size_type max_size() const`	Container	Returns the largest possible size of the `vector`.
`size_type capacity() const`	`vector`	See below.
`bool empty() const`	Container	`true` if the `vector`'s size is `0`.
`reference operator[] (size_type n)`	Random Access Container	Returns the `n`'th element.
`const_reference operator[] (size_type n) const`	Random Access Container	Returns the `n`'th element.
`vector()`	Container	Creates an empty vector.
`vector(size_type n)`	Sequence	Creates a vector with `n` elements.

Member	Where defined	Description
`vector(size_type n, const T& t)`	Sequence	Creates a vector with `n` copies of `t`.
`vector(const vector&)`	Container	The copy constructor.
`template <class InputIterator> vector (InputIterator, InputIterator)`	Sequence	Creates a vector with a copy of a range.
`~vector()`	Container	The destructor.
`vector& operator=(const vector&)`	Container	The assignment operator
`void reserve(size_t)`	`vector`	See below.
`reference front()`	Sequence	Returns the first element.
`const_reference front() const`	Sequence	Returns the first element.
`reference back()`	Back Insertion Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_back()`	Back Insertion Sequence	Removes the last element.
`void swap(vector&)`	Container	Swaps the contents of two vectors.
`iterator insert(iterator pos, const T& x)`	Sequence	Inserts `x` before `pos`.
`template <class InputIterator> void insert (iterator pos, InputIterator f, InputIterator l)`	Sequence	Inserts the range `[first, last)` before `pos`.

Member	Where defined	Description
`void insert (iterator pos, size_type n, const T& x)`	Sequence	Inserts `n` copies of `x` before `pos`.
`iterator erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`void resize(n, t = T())`	Sequence	Inserts or erases elements at the end such that the size becomes `n`.
`bool operator== (const vector&, const vector&)`	Forward Container	Tests two vectors for equality. This is a global function, not a member function.
`bool operator< (const vector&, const vector&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Random Access Container and Back Insertion Sequence requirements, but are specific to `vector`.

Member	Description
`size_type capacity() const`	Number of elements for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.
`void reserve(size_type n)`	If `n` is less than or equal to `capacity()`, this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then `capacity()` is greater than or equal to `n`; otherwise, `capacity()` is unchanged. In either case, `size()` is unchanged.

Notes

This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must be of type `const value_type*`. Memory will be reallocated automatically if more than `capacity() - size()` elements are inserted into the vector. Reallocation does not change `size()`, nor does it change the values of any elements of the vector. It does, however, increase `capacity()`, and it invalidates any iterators that point into the vector. When it is necessary to increase `capacity()`, `vector` usually increases it by a factor of two. It is crucial that the amount of growth is proportional to the current `capacity()`, rather than a fixed constant: in the former case inserting a series of elements into a vector is a linear time operation, and in the latter case it is quadratic. `Reserve()` causes a reallocation manually. The main reason for using `reserve()` is efficiency: if you know the capacity to which your `vector` must eventually grow, then it is usually more efficient to allocate that memory all at once rather than relying on the automatic reallocation scheme. The other reason for using `reserve()` is so that you can control the invalidation of iterators. A vector's iterators are invalidated when its memory is reallocated. Additionally, inserting or deleting an element in the middle of a vector invalidates all iterators that point to elements following the insertion or deletion point. It follows that you can prevent a vector's iterators from being invalidated if you use `reserve()` to preallocate as much memory as the vector will ever use, and if all insertions and deletions are at the vector's end.

See also

deque, list, slist

deque

Description

A `deque` is very much like a `vector`: like `vector`, it is a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle. The main way in which `deque` differs from `vector` is that `deque` also supports constant time insertion and removal of elements at the beginning of the sequence . Additionally, `deque` does not have any member functions analogous to `vector`'s `capacity()` and `reserve()`, and does not provide any of the guarantees on iterator validity that are associated with those member functions.

Example

```
deque<int> Q;
Q.push_back(3);
Q.push_front(1);
Q.insert(Q.begin() + 1, 2);
Q[2] = 0;
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Definition

Defined in the standard header deque, and in the nonstandard backward-compatibility header deque.h.

Template parameters

Parameter	Description	Default
`T`	The deque's value type: the type of object that is stored in the deque.	
`Alloc`	The `deque`'s allocator, used for all internal memory management.	`alloc`

Model of

Random access container, Front insertion sequence, Back insertion sequence.

Type requirements

None, except for those imposed by the requirements of Random access container, Front insertion sequence, and Back insertion sequence.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the deque.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `deque`.
`const_iterator`	Container	Const iterator used to iterate through a `deque`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `deque`.
`const_reverse_-iterator`	Reversible Container	Const iterator used to iterate backwards through a `deque`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `deque`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `deque`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `deque`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `deque`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed deque.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed deque.
`const_reverse_-iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed deque.
`const_reverse_-iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed deque.
`size_type size() const`	Container	Returns the size of the `deque`.
`size_type max_size() const`	Container	Returns the largest possible size of the `deque`.
`bool empty() const`	Container	`true` if the `deque`'s size is `0`.
`reference operator[] (size_type n)`	Random Access Container	Returns the `n`'th element.
`const_reference operator[] (size_type n) const`	Random Access Container	Returns the `n`'th element.
`deque()`	Container	Creates an empty deque.
`deque(size_type n)`	Sequence	Creates a deque with `n` elements.
`deque(size_type n, const T& t)`	Sequence	Creates a deque with `n` copies of `t`.
`deque(const deque&)`	Container	The copy constructor.

Member	Where defined	Description
`template <class InputIterator> deque (InputIterator f, InputIterator l)`	Sequence	Creates a deque with a copy of a range.
`~deque()`	Container	The destructor.
`deque& operator=(const deque&)`	Container	The assignment operator
`reference front()`	Front Insertion Sequence	Returns the first element.
`const_reference front() const`	Front Insertion Sequence	Returns the first element.
`reference back()`	Back Insertion Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_front(const T&)`	Front Insertion Sequence	Inserts a new element at the beginning.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_front()`	Front Insertion Sequence	Removes the first element.
`void pop_back()`	Back Insertion Sequence	Removes the last element.
`void swap(deque&)`	Container	Swaps the contents of two deques.
`iterator insert(iterator pos, const T& x)`	Sequence	Inserts `x` before `pos`.

Member	Where defined	Description
`template <class InputIterator> void insert(iterator pos, InputIterator f, InputIterator l)`	Sequence	Inserts the range `[f, l)` before `pos`.
`void insert(iterator pos, size_type n, const T& x)`	Sequence	Inserts `n` copies of `x` before `pos`.
`iterator erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`void resize(n, t = T())`	Sequence	Inserts or erases elements at the end such that the size becomes `n`.
`bool operator==(const deque&, const deque&)`	Forward Container	Tests two deques for equality. This is a global function, not a member function.
`bool operator<(const deque&, const deque&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

All of `deque`'s members are defined in the Random access container, Front insertion sequence, and Back insertion sequence requirements. `Deque` does not introduce any new members.

Notes

The name *deque* is pronounced "deck", and stands for "double-ended queue." Knuth (section 2.6) reports that the name was coined by E. J. Schweppe. See section 2.2.1 of Knuth for more information about deques. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.) Inserting an element at the beginning or end of a `deque` takes amortized constant time. Inserting an element in the middle is linear in `n`, where `n` is the smaller of the number of elements from the insertion point to the beginning, and the number of elements from the insertion point to the end. The semantics of iterator invalidation for `deque` is as follows. `Insert` (including `push_front` and `push_back`) invalidates all iterators that refer to a `deque`. `Erase` in the middle of a `deque` invalidates all iterators that refer to the `deque`. `Erase` at the beginning or end of a `deque` (including `pop_front` and `pop_back`) invalidates an iterator only if it points to the erased element. This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `deque::const_iterator`.

See also

`vector`, `list`, `slist`

list

Description

A `list` is a doubly linked list. That is, it is a Sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at the beginning or the end, or in the middle. `List`s have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, `list<T>::iterator` might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit. Note that singly linked lists, which only support forward traversal, are also sometimes useful. If you do not need backward traversal, then `slist` may be more efficient than `list`.

Definition

Defined in the standard header list, and in the nonstandard backward-compatibility header list.h.

Example

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Template parameters

Parameter	Description	Default
`T`	The `list`'s value type: the type of object that is stored in the list.	
`Alloc`	The `list`'s allocator, used for all internal memory management.	`alloc`

Model of

Reversible Container, Front Insertion Sequence, Back Insertion Sequence.

Type requirements

None, except for those imposed by the requirements of Reversible Container, Front Insertion Sequence, and Back Insertion Sequence.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the list.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `list`.
`const_iterator`	Container	Const iterator used to iterate through a `list`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `list`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `list`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `list`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `list`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `list`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `list`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed list.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed list.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed list.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed list.
`size_type size() const`	Container	Returns the size of the `list`. Note: you should not assume that this function is constant time. It is permitted to be *O(N)*, where *N* is the number of elements in the `list`. If you wish to test whether a `list` is empty, you should write `L.empty()` rather than `L.size() == 0`.
`size_type max_size() const`	Container	Returns the largest possible size of the `list`.
`bool empty() const`	Container	`true` if the `list`'s size is `0`.
`list()`	Container	Creates an empty list.
`list(size_type n)`	Sequence	Creates a list with `n` elements, each of which is a copy of `T()`.
`list(size_type n, const T& t)`	Sequence	Creates a list with `n` copies of `t`.
`list(const list&)`	Container	The copy constructor.

Member	Where defined	Description
`template <class InputIterator> list (InputIterator f, InputIterator l)`	Sequence	Creates a list with a copy of a range.
`~list()`	Container	The destructor.
`list& operator=(const list&)`	Container	The assignment operator
`reference front()`	Front Insertion Sequence	Returns the first element.
`const_reference front() const`	Front Insertion Sequence	Returns the first element.
`reference back()`	Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_front(const T&)`	Front Insertion Sequence	Inserts a new element at the beginning.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_front()`	Front Insertion Sequence	Removes the first element.
`void pop_back()`	Back Insertion Sequence	Removes the last element.
`void swap(list&)`	Container	Swaps the contents of two lists.
`iterator insert(iterator pos, const T& x)`	Sequence	Inserts `x` before `pos`.
`template <class InputIterator> void insert(iterator pos, InputIterator f, InputIterator l)`	Sequence	Inserts the range `[f, l)` before `pos`.

Member	Where defined	Description
`void insert(iterator pos, size_type n, const T& x)`	Sequence	Inserts `n` copies of `x` before `pos`.
`iterator erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`void resize(n, t = T())`	Sequence	Inserts or erases elements at the end such that the size becomes `n`.
`void splice(iterator pos, list& L)`	`list`	See below.
`void splice(iterator pos, list& L, iterator i)`	`list`	See below.
`void splice(iterator pos, list& L, iterator f, iterator l)`	`list`	See below.
`void remove(const T& value)`	`list`	See below.
`void unique()`	`list`	See below.
`void merge(list& L)`	`list`	See below.
`void sort()`	`list`	See below.
`bool operator==(const list&, const list&)`	Forward Container	Tests two lists for equality. This is a global function, not a member function.
`bool operator<(const list&, const list&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Reversible Container, Front Insertion Sequence, and Back Insertion Sequence requirements, but are specific to `list`.

Function	Description
`void splice(iterator position, list<T, Alloc>& x);`	`position` must be a valid iterator in `*this`, and `x` must be a list that is distinct from `*this`. (That is, it is required that `&x != this`.) All of the elements of `x` are inserted before `position` and removed from `x`. All iterators remain valid, including iterators that point to elements of `x`. This function is constant time.
`void splice(iterator position, list<T, Alloc>& x, iterator i);`	`position` must be a valid iterator in `*this`, and `i` must be a dereferenceable iterator in `x`. `Splice` moves the element pointed to by `i` from `x` to `*this`, inserting it before `position`. All iterators remain valid, including iterators that point to elements of `x`. If `position == i` or `position == ++i`, this function is a null operation. This function is constant time.
`void splice(iterator position, list<T, Alloc>& x, iterator f, iterator l);`	`position` must be a valid iterator in `*this`, and `[first, last)` must be a valid range in `x`. `position` may not be an iterator in the range `[first, last)`. `Splice` moves the elements in `[first, last)` from `x` to `*this`, inserting them before `position`. All iterators remain valid, including iterators that point to elements of `x`. This function is constant time.
`void remove(const T& val);`	Removes all elements that compare equal to `val`. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size()` comparisons for equality.
`template<class Predicate> void remove_if(Predicate p);`	Removes all elements `*i` such that `p(*i)` is true. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size()` applications of `p`.
`void unique();`	Removes all but the first element in every consecutive group of equal elements. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size() - 1` comparisons for equality.
`template<class BinaryPredicate> void unique (BinaryPredicate p);`	Removes all but the first element in every consecutive group of equivalent elements, where two elements `*i` and `*j` are considered equivalent if `p(*i, *j)` is true. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size() - 1` comparisons for equality.
`void merge(list<T, Alloc>& x);`	Both `*this` and `x` must be sorted according to `operator<`, and they must be distinct. (That is, it is required that `&x != this`.) This function removes all of `x`'s elements and inserts them in order into `*this`. The merge is stable; that is, if an element from `*this` is equivalent to one from `x`, then the element from `*this` will precede the one from `x`. All iterators to elements in `*this` and `x` remain valid. This function is linear time: it performs at most `size() + x.size() - 1` comparisons.

Function	Description
`template<class BinaryPredicate> void merge(list<T, Alloc>& x, BinaryPredicate Comp);`	`Comp` must be a comparison function that induces a strict weak ordering (as defined in the LessThan Comparable requirements) on objects of type `T`, and both `*this` and `x` must be sorted according to that ordering. The lists `x` and `*this` must be distinct. (That is, it is required that `&x != this`.) This function removes all of `x`'s elements and inserts them in order into `*this`. The merge is stable; that is, if an element from `*this` is equivalent to one from `x`, then the element from `*this` will precede the one from `x`. All iterators to elements in `*this` and `x` remain valid. This function is linear time: it performs at most `size() + x.size() - 1` applications of `Comp`.
`void reverse();`	Reverses the order of elements in the list. All iterators remain valid and continue to point to the same elements. This function is linear time.
`void sort();`	Sorts `*this` according to `operator<`. The sort is stable, that is, the relative order of equivalent elements is preserved. All iterators remain valid and continue to point to the same elements. The number of comparisons is approximately `N log N`, where `N` is the `list`'s size.
`template<class BinaryPredicate> void sort(BinaryPredicate comp);`	`Comp` must be a comparison function that induces a strict weak ordering (as defined in the LessThan Comparable requirements on objects of type `T`. This function sorts the list `*this` according to `Comp`. The sort is stable, that is, the relative order of equivalent elements is preserved. All iterators remain valid and continue to point to the same elements. The number of comparisons is approximately `N log N`, where `N` is the `list`'s size.

Notes

A comparison with `vector` is instructive. Suppose that `i` is a valid `vector<T>::iterator`. If an element is inserted or removed in a position that precedes `i`, then this operation will either result in `i` pointing to a different element than it did before, or else it will invalidate `i` entirely. (A `vector<T>::iterator` will be invalidated, for example, if an insertion requires a reallocation.) However, suppose that `i` and `j` are both iterators into a vector, and there exists some integer `n` such that `i == j + n`. In that case, even if elements are inserted into the vector and `i` and `j` point to different elements, the relation between the two iterators will still hold. A `list` is exactly the opposite: iterators will not be invalidated, and will not be made to point to different elements, but, for `list` iterators, the predecessor/successor relationship is not invariant. This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `list::const_iterator`. A similar property holds for all versions of `insert()` and `erase()`. `List<T, Alloc>::insert()` never invalidates any iterators, and `list<T, Alloc>::erase()` only invalidates iterators pointing to the elements that are actually being erased. This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. You can only use this member function if your compiler supports member templates. If `L` is a list, note that `L.reverse()` and `reverse(L.begin(), L.end())` are both correct ways of reversing the list. They differ in that `L.reverse()` will preserve the value that each iterator into `L` points to but will not preserve the iterators' predecessor/successor relationships, while `reverse(L.begin(), L.end())` will not preserve the value that each iterator points to but will preserve the iterators' predecessor/successor relationships. Note also that the algorithm `reverse(L.begin(), L.end())` will use `T`'s assignment operator, while the member function `L.reverse()` will not. The `sort` algorithm works only for random access iterators. In principle, however, it would be possible to write a sort algorithm that also accepted bidirectional iterators. Even if there were such a version of `sort`, it would still be useful for `list` to have a `sort` member function. That is, `sort` is provided as a member function not only for the sake of efficiency, but also because of the property that it preserves the values that list iterators point to.

See also

Bidirectional Iterator, Reversible Container, Sequence, `slist` `vector`.

bit_vector

Description

A `bit_vector` is essentially a `vector<bool>`: it is a Sequence that has the same interface as `vector`. The main difference is that `bit_vector` is optimized for space efficiency. A `vector` always requires at least one byte per element, but a `bit_vector` only requires one bit per element. **Warning**: The name `bit_vector` will be removed in a future release of the STL. The only reason that `bit_vector` is a separate class, instead of a template specialization of `vector<bool>`, is that this would require partial specialization of templates. On compilers that support partial specialization, `bit_vector` is a specialization of `vector<bool>`. The name `bit_vector` is a `typedef`. This `typedef` is not defined in the C++ standard, and is retained only for backward compatibility.

Example

```
bit_vector V(5);
V[0] = true;
V[1] = false;
V[2] = false;
V[3] = true;
V[4] = false;

for (bit_vector::iterator i = V.begin(); i < V.end(); ++i)
  cout << (*i ? '1' : '0');
cout << endl;
```

Definition

Defined in the standard header vector, and in the nonstandard backward-compatibility header bvector.h.

Template parameters

None. `Bit_vector` is not a class template.

Model of

Random access container, Back insertion sequence.

Type requirements

None.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object stored in the bit_vector: `bool`
`reference`	`bit_vector`	A proxy class that acts as a reference to a single bit. See below for details.
`const_reference`	Container	Const reference to `value_type`. In `bit_vector` this is simply defined to be `bool`.
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `bit_vector`.
`const_iterator`	Container	Const iterator used to iterate through a `bit_vector`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `bit_vector`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `bit_vector`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `bit_vector`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `bit_vector`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `bit_vector`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `bit_vector`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed bit_vector.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed bit_vector.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed bit_vector.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed bit_vector.
`size_type size() const`	Container	Returns the number of elements in the `bit_vector`.
`size_type max_size() const`	Container	Returns the largest possible size of the `bit_vector`.
`size_type capacity() const`	`bit_vector`	See below.
`bool empty() const`	Container	`true` if the `bit_vector`'s size is `0`.
`reference operator[] (size_type n)`	Random Access Container	Returns the `n`'th element.
`const_reference operator[] (size_type n) const`	Random Access Container	Returns the `n`'th element.
`bit_vector()`	Container	Creates an empty bit_vector.

Member	Where defined	Description
`bit_vector(size_type n)`	Sequence	Creates a bit_vector with `n` elements.
`bit_vector(size_type n, bool t)`	Sequence	Creates a bit_vector with `n` copies of `t`.
`bit_vector(const bit_vector&)`	Container	The copy constructor.
`template <class InputIterator> bit_vector (InputIterator, InputIterator)`	Sequence	Creates a bit_vector with a copy of a range.
`~bit_vector()`	Container	The destructor.
`bit_vector& operator=(const bit_vector&)`	Container	The assignment operator
`void reserve(size_t)`	`bit_vector`	See below.
`reference front()`	Sequence	Returns the first element.
`const_reference front() const`	Sequence	Returns the first element.
`reference back()`	Back Insertion Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_back()`	Back Insertion Sequence	Removes the last element.
`void swap(bit_vector&)`	Container	Swaps the contents of two bit_vectors.
`void swap (bit_vector::reference x, bit_vector::reference y)`	`bit_vector`	See below.
`iterator insert(iterator pos, bool x)`	Sequence	Inserts `x` before `pos`.
`template <class InputIterator> void insert(iterator pos, InputIterator f, InputIterator l)`	Sequence	Inserts the range `[f, l)` before `pos`.
`void insert(iterator pos, size_type n, bool x)`	Sequence	Inserts `n` copies of `x` before `pos`.

Member	Where defined	Description
`void erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`void erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`bool operator==(const bit_vector&, const bit_vector&)`	Forward Container	Tests two bit_vectors for equality. This is a global function, not a member function.
`bool operator<(const bit_vector&, const bit_vector&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Random access container and Back insertion sequence requirements, but are specific to `vector`.

Member	Description
`reference`	A proxy class that acts as a reference to a single bit; the reason it exists is to allow expressions like `V[0] = true`. (A proxy class like this is necessary, because the C++ memory model does not include independent addressing of objects smaller than one byte.) The public member functions of `reference` are `operator bool() const`, `reference& operator=(bool)`, and `void flip()`. That is, `reference` acts like an ordinary reference: you can convert a `reference` to `bool`, assign a `bool` value through a `reference`, or flip the bit that a `reference` refers to.
`size_type capacity() const`	Number of bits for which memory has been allocated. `capacity()` is always greater than or equal to `size()`.
`void reserve(size_type n)`	If `n` is less than or equal to `capacity()`, this call has no effect. Otherwise, it is a request for the allocation of additional memory. If the request is successful, then `capacity()` is greater than or equal to `n`; otherwise, `capacity()` is unchanged. In either case, `size()` is unchanged.
`void swap (bit_vector::reference x, bit_vector::reference y)`	Swaps the bits referred to by `x` and `y`. This is a global function, not a member function. It is necessary because the ordinary version of `swap` takes arguments of type `T&`, and `bit_vector::reference` is a class, not a built-in C++ reference.

Notes

This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const bool*` or of type `bit_vector::const_iterator`. Memory will be reallocated automatically if more than `capacity() - size()` bits are inserted into the bit_vector. Reallocation does not change `size()`, nor does it change the values of any bits of the bit_vector. It does, however, increase `capacity()`, and it invalidates any iterators that point into the bit_vector. When it is necessary to increase `capacity()`, `bit_vector` usually increases it by a factor of two. It is crucial that the amount of growth is proportional to the current `capacity()`, rather than a fixed constant: in the former case inserting a series of bits into a bit_vector is a linear time operation, and in the latter case it is quadratic. `reserve()` is used to cause a reallocation manually. The main reason for using `reserve()` is efficiency: if you know the capacity to which your `bit_vector` must eventually grow, then it is probably more efficient to allocate that memory all at once rather than relying on the automatic reallocation scheme. The other reason for using `reserve()` is to control the invalidation of iterators. A `bit_vector`'s iterators are invalidated when its memory is reallocated. Additionally, inserting or deleting a bit in the middle of a bit_vector invalidates all iterators that point to bits following the insertion or deletion point. It follows that you can prevent a bit_vector's iterators from being invalidated if you use `reserve()` to preallocate as much storage as the bit_vector will ever use, and if all insertions and deletions are at the bit_vector's end.

See also

vector

7.2.2 Associative Containers

set

Description

`Set` is a Sorted Associative Container that stores objects of type `Key`. `Set` is a Simple Associative Container, meaning that its value type, as well as its key type, is `Key`. It is also a Unique Associative Container, meaning that no two elements are the same. `Set` and `multiset` are particularly well suited to the set algorithms `includes`, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference`. The reason for this is twofold. First, the set algorithms require their arguments to be sorted ranges, and, since `set` and `multiset` are Sorted Associative Containers, their elements are always sorted in ascending order. Second, the output range of these algorithms is always sorted, and inserting a sorted range into a `set` or `multiset` is a fast operation: the Unique Sorted Associative Container and Multiple Sorted Associative Container requirements guarantee that inserting a range takes only linear time if the range is already sorted. `Set` has the important property that inserting a new element into a `set` does not invalidate iterators that point to existing elements. Erasing an element from a set also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  const int N = 6;
  const char* a[N] = {"isomer", "ephemeral", "prosaic",
                      "nugatory", "artichoke", "serif"};
  const char* b[N] = {"flat", "this", "artichoke",
                      "frigate", "prosaic", "isomer"};

  set<const char*, ltstr> A(a, a + N);
  set<const char*, ltstr> B(b, b + N);
  set<const char*, ltstr> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<const char*>(cout, " "),
            ltstr());
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<const char*>(cout, " "),
                   ltstr());
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()),
                 ltstr());
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
}
```

Definition

Defined in the standard header set, and in the nonstandard backward-compatibility header set.h.

Template parameters

Parameter	Description	Default
`Key`	The set's key type and value type. This is also defined as `set::key_type` and `set::value_type`	
`Compare`	The key comparison function, a Strict Weak Ordering whose argument type is `key_type`; it returns `true` if its first argument is less than its second argument, and `false` otherwise. This is also defined as `set::key_compare` and `set::value_compare`.	`less<Key>`
`Alloc`	The `set`'s allocator, used for all internal memory management.	`alloc`

Model of

Unique Sorted Associative Container, Simple Associative Container

Type requirements

- `Key` is Assignable.
- `Compare` is a Strict Weak Ordering whose argument type is `Key`.
- `Alloc` is an Allocator.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the set.
`key_type`	Associative Container	The key type associated with `value_type`.
`key_compare`	Sorted Associative Container	Function object that compares two keys for ordering.
`value_compare`	Sorted Associative Container	Function object that compares two values for ordering.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `set`.
`const_iterator`	Container	Const iterator used to iterate through a `set`. (`Iterator` and `const_iterator` are the same type.)
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `set`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `set`. (`Reverse_iterator` and `const_reverse_iterator` are the same type.)
`iterator begin() const`	Container	Returns an `iterator` pointing to the beginning of the `set`.
`iterator end() const`	Container	Returns an `iterator` pointing to the end of the `set`.
`reverse_iterator rbegin() const`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed set.
`reverse_iterator rend() const`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed set.
`size_type size() const`	Container	Returns the size of the `set`.
`size_type max_size() const`	Container	Returns the largest possible size of the `set`.
`bool empty() const`	Container	`true` if the `set`'s size is `0`.
`key_compare key_comp() const`	Sorted Associative Container	Returns the `key_compare` object used by the `set`.
`value_compare value_comp() const`	Sorted Associative Container	Returns the `value_compare` object used by the `set`.
`set()`	Container	Creates an empty `set`.
`set(const key_compare& comp)`	Sorted Associative Container	Creates an empty `set`, using `comp` as the `key_compare` object.
`template <class InputIterator> set (InputIterator f, InputIterator l)`	Unique Sorted Associative Container	Creates a set with a copy of a range.

Member	Where defined	Description
`template <class InputIterator> set (InputIterator f, InputIterator l, const key_compare& comp)`	Unique Sorted Associative Container	Creates a set with a copy of a range, using `comp` as the `key_compare` object.
`set(const set&)`	Container	The copy constructor.
`set& operator=(const set&)`	Container	The assignment operator
`void swap(set&)`	Container	Swaps the contents of two sets.
`pair<iterator, bool> insert(const value_type& x)`	Unique Associative Container	Inserts `x` into the `set`.
`iterator insert(iterator pos, const value_type& x)`	Unique Sorted Associative Container	Inserts `x` into the `set`, using `pos` as a hint to where it will be inserted.
`template <class InputIterator> void insert(InputIterator, InputIterator)`	Unique Sorted Associative Container	Inserts a range into the `set`.
`void erase(iterator pos)`	Associative Container	Erases the element pointed to by `pos`.
`size_type erase(const key_type& k)`	Associative Container	Erases the element whose key is `k`.
`void erase(iterator first, iterator last)`	Associative Container	Erases all elements in a range.
`void clear()`	Associative Container	Erases all of the elements.
`iterator find(const key_type& k) const`	Associative Container	Finds an element whose key is `k`.
`size_type count(const key_type& k) const`	Unique Associative Container	Counts the number of elements whose key is `k`.
`iterator lower_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key is not less than `k`.
`iterator upper_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key greater than `k`.
`pair<iterator, iterator> equal_range(const key_type& k) const`	Sorted Associative Container	Finds a range containing all elements whose key is `k`.
`bool operator==(const set&, const set&)`	Forward Container	Tests two sets for equality. This is a global function, not a member function.
`bool operator<(const set&, const set&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

All of `set`'s members are defined in the Unique Sorted Associative Container and Simple Associative Container requirements. `Set` does not introduce any new members.

Notes

This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `set::const_iterator`.

See also

Associative Container, Sorted Associative Container, Simple Associative Container, Unique Sorted Associative Container, `map`, `multiset`

map

Description

`Map` is a Sorted Associative Container that associates objects of type `Key` with objects of type `Data`. `Map` is a Pair Associative Container, meaning that its value type is `pair<const Key, Data>`. It is also a Unique Associative Container, meaning that no two elements have the same key. `Map` has the important property that inserting a new element into a `map` does not invalidate iterators that point to existing elements. Erasing an element from a `map` also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  map<const char*, int, ltstr> months;

  months["january"] = 31;
  months["february"] = 28;
  months["march"] = 31;
  months["april"] = 30;
  months["may"] = 31;
  months["june"] = 30;
  months["july"] = 31;
  months["august"] = 31;
  months["september"] = 30;
  months["october"] = 31;
  months["november"] = 30;
  months["december"] = 31;

  cout << "june -> " << months["june"] << endl;
  map<const char*, int, ltstr>::iterator cur  = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " << (*prev).first
       << endl;
  cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
```

Definition

Defined in the standard header map, and in the nonstandard backward-compatibility header map.h.

Template parameters

Parameter	Description	Default
`Key`	The map's key type. This is also defined as `map::key_type`.	
`Data`	The map's data type. This is also defined as `map::data_type`.	
`Compare`	The key comparison function, a Strict Weak Ordering whose argument type is `key_type`; it returns `true` if its first argument is less than its second argument, and `false` otherwise. This is also defined as `map::key_compare`.	`less<Key>`
`Alloc`	The `map`'s allocator, used for all internal memory management.	`alloc`

Model of

Unique Sorted Associative Container, Pair Associative Container

Type requirements

- `Data` is Assignable.
- `Compare` is a Strict Weak Ordering whose argument type is `Key`.
- `Alloc` is an Allocator.

Public base classes

None.

Members

Member	Where defined	Description
key_type	Associative Container	The map's key type, Key.
data_type	Pair Associative Container	The type of object associated with the keys.
value_type	Pair Associative Container	The type of object, pair<const key_type, data_type>, stored in the map.
key_compare	Sorted Associative Container	Function object that compares two keys for ordering.
value_compare	Sorted Associative Container	Function object that compares two values for ordering.
pointer	Container	Pointer to T.
reference	Container	Reference to T
const_reference	Container	Const reference to T
size_type	Container	An unsigned integral type.
difference_type	Container	A signed integral type.
iterator	Container	Iterator used to iterate through a map.
const_iterator	Container	Const iterator used to iterate through a map.
reverse_iterator	Reversible Container	Iterator used to iterate backwards through a map.
const_reverse_iterator	Reversible Container	Const iterator used to iterate backwards through a map.
iterator begin()	Container	Returns an iterator pointing to the beginning of the map.
iterator end()	Container	Returns an iterator pointing to the end of the map.
const_iterator begin() const	Container	Returns a const_iterator pointing to the beginning of the map.
const_iterator end() const	Container	Returns a const_iterator pointing to the end of the map.
reverse_iterator rbegin()	Reversible Container	Returns a reverse_iterator pointing to the beginning of the reversed map.
reverse_iterator rend()	Reversible Container	Returns a reverse_iterator pointing to the end of the reversed map.
const_reverse_iterator rbegin() const	Reversible Container	Returns a const_reverse_iterator pointing to the beginning of the reversed map.
const_reverse_iterator rend() const	Reversible Container	Returns a const_reverse_iterator pointing to the end of the reversed map.
size_type size() const	Container	Returns the size of the map.
size_type max_size() const	Container	Returns the largest possible size of the map.
bool empty() const	Container	true if the map's size is 0.

Member	Where defined	Description
`key_compare key_comp() const`	Sorted Associative Container	Returns the `key_compare` object used by the `map`.
`value_compare value_comp() const`	Sorted Associative Container	Returns the `value_compare` object used by the `map`.
`map()`	Container	Creates an empty `map`.
`map(const key_compare& comp)`	Sorted Associative Container	Creates an empty `map`, using `comp` as the `key_compare` object.
`template <class InputIterator> map(InputIterator f, InputIterator l)`	Unique Sorted Associative Container	Creates a map with a copy of a range.
`template <class InputIterator> map(InputIterator f, InputIterator l, const key_compare& comp)`	Unique Sorted Associative Container	Creates a map with a copy of a range, using `comp` as the `key_compare` object.
`map(const map&)`	Container	The copy constructor.
`map& operator=(const map&)`	Container	The assignment operator
`void swap(map&)`	Container	Swaps the contents of two maps.
`pair<iterator, bool> insert(const value_type& x)`	Unique Associative Container	Inserts `x` into the `map`.
`iterator insert(iterator pos, const value_type& x)`	Unique Sorted Associative Container	Inserts `x` into the `map`, using `pos` as a hint to where it will be inserted.
`template <class InputIterator> void insert(InputIterator, InputIterator)`	Unique Sorted Associative Container	Inserts a range into the `map`.
`void erase(iterator pos)`	Associative Container	Erases the element pointed to by `pos`.
`size_type erase(const key_type& k)`	Associative Container	Erases the element whose key is `k`.
`void erase(iterator first, iterator last)`	Associative Container	Erases all elements in a range.
`void clear()`	Associative Container	Erases all of the elements.
`iterator find(const key_type& k)`	Associative Container	Finds an element whose key is `k`.
`const_iterator find(const key_type& k) const`	Associative Container	Finds an element whose key is `k`.
`size_type count(const key_type& k)`	Unique Associative Container	Counts the number of elements whose key is `k`.

Member	Where defined	Description
`iterator lower_bound(const key_type& k)`	Sorted Associative Container	Finds the first element whose key is not less than `k`.
`const_iterator lower_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key is not less than `k`.
`iterator upper_bound(const key_type& k)`	Sorted Associative Container	Finds the first element whose key greater than `k`.
`const_iterator upper_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key greater than `k`.
`pair<iterator, iterator> equal_range(const key_type& k)`	Sorted Associative Container	Finds a range containing all elements whose key is `k`.
`pair<const_iterator, const_iterator> equal_range(const key_type& k) const`	Sorted Associative Container	Finds a range containing all elements whose key is `k`.
`data_type operator[](const key_type& k)`	`map`	See below.
`bool operator==(const map&, const map&)`	Forward Container	Tests two maps for equality. This is a global function, not a member function.
`bool operator<(const map&, const map&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Unique Sorted Associative Container and Pair Associative Container requirements, but are unique to `map`:

Member function	Description
`data_type operator[](const key_type& k)`	Returns a reference to the object that is associated with a particular key. If the `map` does not already contain such an object, `operator[]` inserts the default object `data_type()`.

Notes

`Map::iterator` is not a mutable iterator, because `map::value_type` is not Assignable. That is, if `i` is of type `map::iterator` and `p` is of type `map::value_type`, then `*i = p` is not a valid expression. However, `map::iterator` isn't a constant iterator either, because it can be used to modify the object that it points to.

Using the same notation as above, `(*i).second = p` is a valid expression. The same point applies to `map::reverse_iterator`. This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `map::const_iterator`. Since `operator[]` might insert a new element into the `map`, it can't possibly be a `const` member function. Note that the definition of `operator[]` is extremely simple: `m[k]` is equivalent to `(*((m.insert(value_type(k, data_type()))).first)).second`. Strictly speaking, this member function is unnecessary: it exists only for convenience.

See also

Associative Container, Sorted Associative Container, Pair Associative Container, Unique Sorted Associative Container, `set` `multiset`

multiset

Description

`Multiset` is a Sorted Associative Container that stores objects of type `Key`. `Multiset` is a Simple Associative Container, meaning that its value type, as well as its key type, is `Key`. It is also a Multiple Associative Container, meaning that two or more elements may be identical. `Set` and `multiset` are particularly well suited to the set algorithms `includes`, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference`. The reason for this is twofold. First, the set algorithms require their arguments to be sorted ranges, and, since `set` and `multiset` are Sorted Associative Containers, their elements are always sorted in ascending order. Second, the output range of these algorithms is always sorted, and inserting a sorted range into a `set` or `multiset` is a fast operation: the Unique Sorted Associative Container and Multiple Sorted Associative Container requirements guarantee that inserting a range takes only linear time if the range is already sorted. `Multiset` has the important property that inserting a new element into a `multiset` does not invalidate iterators that point to existing elements. Erasing an element from a multiset also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
int main()
{
  const int N = 10;
  int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
  int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

  multiset<int> A(a, a + N);
  multiset<int> B(b, b + N);
  multiset<int> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<int>(cout, " "));
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()));
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Definition

Defined in the standard header set, and in the nonstandard backward-compatibility header multiset.h.

Template parameters

Parameter	Description	Default
`Key`	The set's key type and value type. This is also defined as `multiset::key_type` and `multiset::value_type`	
`Compare`	The key comparison function, a Strict Weak Ordering whose argument type is `key_type`; it returns `true` if its first argument is less than its second argument, and `false` otherwise. This is also defined as `multiset::key_compare` and `multiset::value_compare`.	`less<Key>`
`Alloc`	The `multiset`'s allocator, used for all internal memory management.	`alloc`

Model of

Multiple Sorted Associative Container, Simple Associative Container

Type requirements

- `Key` is Assignable.
- `Compare` is a Strict Weak Ordering whose argument type is `Key`.
- `Alloc` is an Allocator.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the multiset.
`key_type`	Associative Container	The key type associated with `value_type`.
`key_compare`	Sorted Associative Container	Function object that compares two keys for ordering.
`value_compare`	Sorted Associative Container	Function object that compares two values for ordering.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `multiset`.
`const_iterator`	Container	Const iterator used to iterate through a `multiset`. (`Iterator` and `const_iterator` are the same type.)
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `multiset`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `multiset`. (`Reverse_iterator` and `const_reverse_iterator` are the same type.)
`iterator begin() const`	Container	Returns an `iterator` pointing to the beginning of the `multiset`.
`iterator end() const`	Container	Returns an `iterator` pointing to the end of the `multiset`.
`reverse_iterator rbegin() const`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed multiset.
`reverse_iterator rend() const`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed multiset.
`size_type size() const`	Container	Returns the size of the `multiset`.
`size_type max_size() const`	Container	Returns the largest possible size of the `multiset`.
`bool empty() const`	Container	`true` if the `multiset`'s size is `0`.
`key_compare key_comp() const`	Sorted Associative Container	Returns the `key_compare` object used by the `multiset`.
`value_compare value_comp() const`	Sorted Associative Container	Returns the `value_compare` object used by the `multiset`.
`multiset()`	Container	Creates an empty `multiset`.
`multiset(const key_compare& comp)`	Sorted Associative Container	Creates an empty `multiset`, using `comp` as the `key_compare` object.
`template <class InputIterator> multiset (InputIterator f, InputIterator l)`	Multiple Sorted Associative Container	Creates a multiset with a copy of a range.

Member	Where defined	Description
template <class InputIterator> multiset (InputIterator f, InputIterator l, const key_compare& comp)	Multiple Sorted Associative Container	Creates a multiset with a copy of a range, using comp as the key_compare object.
multiset(const multiset&)	Container	The copy constructor.
multiset& operator=(const multiset&)	Container	The assignment operator
void swap(multiset&)	Container	Swaps the contents of two multisets.
iterator insert(const value_type& x)	Multiple Associative Container	Inserts x into the multiset.
iterator insert(iterator pos, const value_type& x)	Multiple Sorted Associative Container	Inserts x into the multiset, using pos as a hint to where it will be inserted.
template <class InputIterator> void insert(InputIterator, InputIterator)	Multiple Sorted Associative Container	Inserts a range into the multiset.
void erase(iterator pos)	Associative Container	Erases the element pointed to by pos.
size_type erase(const key_type& k)	Associative Container	Erases the element whose key is k.
void erase(iterator first, iterator last)	Associative Container	Erases all elements in a range.
void clear()	Associative Container	Erases all of the elements.
iterator find(const key_type& k) const	Associative Container	Finds an element whose key is k.
size_type count(const key_type& k) const	Associative Container	Counts the number of elements whose key is k.
iterator lower_bound(const key_type& k) const	Sorted Associative Container	Finds the first element whose key is not less than k.
iterator upper_bound(const key_type& k) const	Sorted Associative Container	Finds the first element whose key greater than k.
pair<iterator, iterator> equal_range(const key_type& k) const	Sorted Associative Container	Finds a range containing all elements whose key is k.
bool operator==(const multiset&, const multiset&)	Forward Container	Tests two multisets for equality. This is a global function, not a member function.
bool operator<(const multiset&, const multiset&)	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

All of multiset's members are defined in the Multiple Sorted Associative Container and Simple Associative Container requirements. Multiset does not introduce any new members.

Notes

This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `multiset::const_iterator`.

See also

Associative Container, Sorted Associative Container, Simple Associative Container, Multiple Sorted Associative Container, `set`, `map`.

Character Traits

Description

Several library components, including strings, need to perform operations on characters. A Character Traits class is similar to a function object: it encapsulates some information about a particular character type, and some operations on that type. Note that every member of a Character Traits class is static. There is never any need to create a Character Traits object, and, in fact, there is no guarantee that creating such objects is possible.

Refinement of

Character Traits is not a refinement of any other concept.

Associated types

Value type	`X::char_type`	The character type described by this Character Traits type.
Int type	`X::int_type`	A type that is capable of representing every valid value of type `char_type`, and, additionally an end-of-file value. For `char`, for example, the int type may be `int`, and for `wchar_t` it may be `wint_t`.
Position type	`X::pos_type`	A type that can represent the position of a character of type `char_type` within a file. This type is usually `streampos`.
Offset type	`X::off_type`	An integer type that can represent the difference between two `pos_type` values. This type is usually `streamoff`.
State type	`X::state_type`	A type that can represent a state in a multibyte encoding scheme. This type, if used at all, is usually `mbstate_t`.

Notation

`X`	A type that is a model of Character Traits.
`c`, `c1`, `c2`	A value of `X`'s value type, `X::char_type`.
`e`, `e1`, `e2`	A value of `X`'s int type, `X::int_type`.
`n`	A value of type `size_t`.
`p`, `p1`, `p2`	A non-null pointer of type `const X::char_type*`.
`s`	A non-null pointer of type `X::char_type*`.

Valid Expressions

Name	Expression	Type requirements	Return type
Character assignment	`X::assign(c1, c2)`	`c1` is a modifiable lvalue.	`void`
Character equality	`X::eq(c1, c2)`		`bool`
Character comparison	`X::lt(c1, c2)`		`bool`
Range comparison	`X::compare(p1, p2, n)`		`int`
Length	`X::length(p)`		`size_t`
Find	`X::find(p, n, c)`		`const X::char_type*`
Move	`X::move(s, p, n)`		`X::char_type*`
Copy	`X::copy(s, p, n)`		`X::char_type*`
Range assignment	`X::assign(s, n, c)`		`X::char_type*`
EOF value	`X::eof()`		`X::int_type`
Not EOF	`X::not_eof(e)`		`X::int_type`
Convert to value type	`X::to_char_type(e)`		`X::char_type`
Convert to int type	`X::to_int_type(c)`		`X::int_type`
Equal int type values	`X::eq_int_type(e1, e2)`		`bool`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Character assignment	`X::assign(c1, c2)`		Performs the assignment `c1 = c2`	`X::eq(c1, c2)` is `true`.
Character equality	`X::eq(c1, c2)`		Returns `true` if and only if `c1` and `c2` are equal.	
Character comparison	`X::lt(c1, c2)`		Returns `true` if and only if `c1` is less than `c2`. Note that for any two value values `c1` and `c2`, exactly one of `X::lt(c1, c2)`, `X::lt(c2, c1)`, and `X::eq(c1, c2)` should be `true`.	
Range comparison	`X::compare(p1, p2, n)`	`[p1, p1+n)` and `[p2, p2+n)` are valid ranges.	Generalization of `strncmp`. Returns `0` if every element in `[p1, p1+n)` is equal to the corresponding element in `[p2, p2+n)`, a negative value if there exists an element in `[p1, p1+n)` less than the corresponding element in `[p2, p2+n)` and all previous elements are equal, and a positive value if there exists an element in `[p1, p1+n)` greater than the corresponding element in `[p2, p2+n)` and all previous elements are equal.	
Length	`X::length(p)`		Generalization of `strlen`. Returns the smallest non-negative number `n` such that `X::eq(p+n, X::char_type())` is true. Behavior is undefined if no such `n` exists.	

Name	Expression	Pre-condition	Semantics	Postcondition
Find	`X::find(p, n, c)`	`[p, p+n)` is a valid range.	Generalization of `strchr`. Returns the first pointer `q` in `[p, p+n)` such that `X::eq(*q, c)` is true. Returns a null pointer if no such pointer exists. (Note that this method for indicating a failed search differs from that is `find`.)	
Move	`X::move(s, p, n)`	`[p, p+n)` and `[s, s+n)` are valid ranges (possibly overlapping).	Generalization of `memmove`. Copies values from the range `[p, p+n)` to the range `[s, s+n)`, and returns `s`.	
Copy	`X::copy(s, p, n)`	`[p, p+n)` and `[s, s+n)` are valid ranges which do not overlap.	Generalization of `memcpy`. Copies values from the range `[p, p+n)` to the range `[s, s+n)`, and returns `s`.	
Range assignment	`X::assign(s, n, c)`	`[s, s+n)` is a valid range.	Generalization of `memset`. Assigns the value `c` to each pointer in the range `[s, s+n)`, and returns `s`.	
EOF value	`X::eof()`		Returns a value that can represent EOF.	`X::eof()` is distinct from every valid value of type `X::char_type`. That is, there exists no value `c` such that `X::eq_int_type(X::to_int_type(c), X::eof())` is `true`.

Name	Expression	Precondition	Semantics	Postcondition
Not EOF	`X:: not_eof(e)`		Returns `e` if `e` represents a valid `char_type` value, and some non-EOF value if `e` is `X::eof()`.	
Convert to value type	`X:: to_char_type (e)`		Converts `e` to `X`'s int type. If `e` is a representation of some `char_type` value then it returns that value; if `e` is `X::eof()` then the return value is unspecified.	
Convert to int type	`X:: to_int_type (c)`		Converts `c` to `X`'s int type.	`X::to_char_type (X::to_int_type (c))` is a null operation.
Equal int type values	`X:: eq_int_type (e1, e2)`		Compares two int type values. If there exist values of type `X::char_type` such that `e1` is `X:: to_int_type(c1))` and `e2` is `X:: to_int_type(c2))`, then `X::eq_int_type (e1, e2)` is the same as `X::eq(c1, c2)`. Otherwise, `eq_int_type` returns `true` if `e1` and `e2` are both EOF and `false` if one of `e1` and `e2` is EOF and the other is not.	

Complexity guarantees

`length`, `find`, `move`, `copy`, and the range version of `assign` are linear in `n`. All other operations are constant time.

Models

- `char_traits<char>`
- `char_traits<wchar_t>`

Notes

See also

`string`

char_traits

Description

The `char_traits` class is the default Character Traits class used by the library; it is the only predefined Character Traits class.

Example

The `char_traits` class is of no use by itself. It is used as a template parameter of other classes, such as the `basic_string` template.

Definition

Defined in the standard header string.

Template parameters

Parameter	Description	Default
`charT`	`char_traits`'s value type, *i.e.* `char_traits<>::char_type`.	

Model of

Character Traits

Type requirements

`charT` is either `char` or `wchar_t`. (All of `char_traits`'s member functions are defined for arbitrary types, but some of `char_traits`'s members must be explicitly specialized if `char_traits` is to be useful for other types than `char` and `wchar_t`.

Public base classes

None.

Members

All of `char_traits`'s members are static. There is never any reason to create an object of type `char_traits`.

Member	Where defined	Description
`char_type`	Character Traits	`char_traits`'s value type: `charT`.
`int_type`	Character Traits	`char_traits`'s int type.
`pos_type`	Character Traits	`char_traits`'s position type.
`off_type`	Character Traits	`char_traits`'s offset type
`state_type`	Character Traits	`char_traits`'s state type.
`static void assign(char_type& c1, const char_type& c2)`	Character Traits	Assigns `c2` to `c1`.
`static bool eq(const char_type& c1, const char_type& c2)`	Character Traits	Character equality.
`static bool lt(const char_type& c1, const char_type& c2)`	Character Traits	Returns `true` if `c1` is less than `c2`.

Member	Where defined	Description
static int compare(const char_type* p1, const char_type* p2, size_t n)	Character Traits	Three-way lexicographical comparison, much like strncmp.
static size_t length(const char* p)	Length	Returns length of a null-terminated array of characters.
static const char_type* find(const char_type* p, size_t n, const char_type& c)	Character Traits	Finds c in [p, p+n), returning 0 if not found.
static char_type* move(char_type* s, const char_type* p, size_t n)	Character Traits	Copies characters from [p, p+n) to the (possibly overlapping) range [s, s+n).
static char_type* copy(char_type* s, const char_type* p, size_t n)	Character Traits	Copies characters from [p, p+n) to the (non-overlapping) range [s, s+n).
static char_type* assign(char_type* s, size_t n, char_type c)	Character Traits	Assigns the value c to every element in the range [s, s+n).
static int_type eof()	Character Traits	Returns the value used as an EOF indicator.
static int_type not_eof(const int_type& c)	Character Traits	Returns a value that is not equal to eof(). Returns c unless c is equal to eof().
static char_type to_char_type(const int_type& c)	Character Traits	Returns the char_type value corresponding to c, if such a value exists.
static int_type to_int_type(const char_type& c)	Character Traits	Returns a int_type representation of c.
static bool eq_int_type(cosnt int_type& c1, const int_type& c1)	Character Traits	Tests whether two int_type values are equal. If the values can also be represented as char_type, then eq and eq_int_type must be consistent with each other.

New members

None. All of char_traits's members are defined in the Character Traits requirements.

Notes

See also

Character Traits, `string`

basic_string

Description

The `basic_string` class represents a Sequence of characters. It contains all the usual operations of a Sequence, and, additionally, it contains standard string operations such as search and concatenation. The `basic_string` class is parameterized by character type, and by that type's Character Traits. Most of the time, however, there is no need to use the `basic_string` template directly. The types `string` and `wstring` are typedefs for, respectively, `basic_string<char>` and `basic_string<wchar_t>`. Some of `basic_string`'s member functions use an unusual method of specifying positions and ranges. In addition to the conventional method using iterators, many of `basic_string`'s member functions use a single value `pos` of type `size_type` to represent a position (in which case the position is `begin() + pos`, and many of `basic_string`'s member functions use two values, `pos` and `n`, to represent a range. In that case `pos` is the beginning of the range and `n` is its size. That is, the range is `[begin() + pos, begin() + pos + n)`.

Example

```
int main() {
  string s(10u, ' ');           // Create a string of ten blanks.

  const char* A = "this is a test";
  s += A;
  cout << "s = " << (s + '\n');
  cout << "As a null-terminated sequence: " << s.c_str() << endl;
  cout << "The sixteenth character is " << s[15] << endl;

  reverse(s.begin(), s.end());
  s.push_back('\n');
  cout << s;
}
```

Definition

Defined in the standard header string.

Template parameters

Parameter	Description	Default
`charT`	The string's value type: the type of character it contains.	
`traits`	The Character Traits type, which encapsulates basic character operations.	`char_traits<charT>`
`Alloc`	The string's allocator, used for internal memory management.	`alloc`

Model of

Random Access Container, Sequence.

Type requirements

In addition to the type requirements imposed by Random Access Container and Sequence:

- `charT` is a POD ("plain ol' data") type.
- `traits` is a Character Traits type whose value type is `charT`

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `CharT`, stored in the string.
`pointer`	Container	Pointer to `CharT`.
`reference`	Container	Reference to `CharT`
`const_reference`	Container	Const reference to `CharT`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`static const size_type npos`	`basic_string`	The largest possible value of type `size_type`. That is, `size_type(-1)`.
`iterator`	Container	Iterator used to iterate through a string. A `basic_string` supplies Random Access Iterators.
`const_iterator`	Container	Const iterator used to iterate through a string.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a string.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a string.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the string.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the string.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the string.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the string.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed string.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed string.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed string.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed string.
`size_type size() const`	Container	Returns the size of the string.
`size_type length() const`	`basic_string`	Synonym for `size()`.
`size_type max_size() const`	Container	Returns the largest possible size of the string.
`size_type capacity() const`	`basic_string`	See below.
`bool empty() const`	Container	`true` if the string's size is `0`.

Member	Where defined	Description
reference operator[] (size_type n)	Random Access Container	Returns the n'th character.
const_reference operator[] (size_type n) const	Random Access Container	Returns the n'th character.
const charT* c_str() const	basic_string	Returns a pointer to a null-terminated array of characters representing the string's contents.
const charT* data() const	basic_string	Returns a pointer to an array of characters (not necessarily null-terminated) representing the string's contents.
basic_string()	Container	Creates an empty string.
basic_string(const basic_string& s, size_type pos = 0, size_type n = npos)	Container, basic_string	Generalization of the copy constructor.
basic_string(const charT*)	basic_string	Construct a string from a null-terminated character array.
basic_string(const charT* s, size_type n)	basic_string	Construct a string from a character array and a length.
basic_string(size_type n, charT c)	Sequence	Create a string with n copies of c.
template <class InputIterator> basic_string(InputIterator first, InputIterator last)	Sequence	Create a string from a range.
~basic_string()	Container	The destructor.
basic_string& operator=(const basic_string&)	Container	The assignment operator
basic_string& operator=(const charT* s)	basic_string	Assign a null-terminated character array to a string.
basic_string& operator=(charT c)	basic_string	Assign a single character to a string.
void reserve(size_t)	basic_string	See below.
void swap(basic_string&)	Container	Swaps the contents of two strings.
iterator insert(iterator pos, const T& x)	Sequence	Inserts x before pos.
template <class InputIterator> void insert(iterator pos, InputIterator f, InputIterator l)	Sequence	Inserts the range [first, last) before pos.
void insert(iterator pos, size_type n, const T& x)	Sequence	Inserts n copies of x before pos.
basic_string& insert(size_type pos, const basic_string& s)	basic_string	Inserts s before pos.

Member	Where defined	Description
`basic_string& insert(size_type pos, const basic_string& s, size_type pos1, size_type n)`	`basic_string`	Inserts a substring of `s` before `pos`.
`basic_string& insert(size_type pos, const charT* s)`	`basic_string`	Inserts `s` before `pos`.
`basic_string& insert(size_type pos, const charT* s, size_type n)`	`basic_string`	Inserts the first n characters of `s` before `pos`.
`basic_string& insert(size_type pos, size_type n, charT c)`	`basic_string`	Inserts `n` copies of `c` before `pos`.
`basic_string& append(const basic_string& s)`	`basic_string`	Append `s` to `*this`.
`basic_string& append(const basic_string& s, size_type pos, size_type n)`	`basic_string`	Append a substring of `s` to `*this`.
`basic_string& append(const charT* s)`	`basic_string`	Append `s` to `*this`.
`basic_string& append(const charT* s, size_type n)`	`basic_string`	Append the first `n` characters of `s` to `*this`.
`basic_string& append(size_type n, charT c)`	`basic_string`	Append `n` copies of `c` to `*this`.
`template <class InputIterator> basic_string& append(InputIterator first, InputIterator last)`	`basic_string`	Append a range to `*this`.
`void push_back(charT c)`	`basic_string`	Append a single character to `*this`.
`basic_string& operator+=(const basic_string& s)`	`basic_string`	Equivalent to `append(s)`.
`basic_string& operator+=(const charT* s)`	`basic_string`	Equivalent to `append(s)`
`basic_string& operator+=(charT c)`	`basic_string`	Equivalent to `push_back(c)`
`iterator erase(iterator p)`	Sequence	Erases the character at position `p`
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`basic_string& erase(size_type pos = 0, size_type n = npos)`	`basic_string`	Erases a range.
`void clear()`	Sequence	Erases the entire container.
`void resize(size_type n, charT c = charT())`	Sequence	Appends characters, or erases characters from the end, as necessary to make the string's length exactly `n` characters.
`basic_string& assign(const basic_string&)`	`basic_string`	Synonym for `operator=`

Member	Where defined	Description
basic_string& assign(const basic_string& s, size_type pos, size_type n)	basic_string	Assigns a substring of s to *this
basic_string& assign(const charT* s, size_type n)	basic_string	Assigns the first n characters of s to *this.
basic_string& assign(const charT* s)	basic_string	Assigns a null-terminated array of characters to *this.
basic_string& assign(size_type n, charT c)	Sequence	Erases the existing characters and replaces them by n copies of c.
template <class InputIterator> basic_string& assign(InputIterator first, InputIterator last)	Sequence	Erases the existing characters and replaces them by [first, last)
basic_string& replace(size_type pos, size_type n, const basic_string& s)	basic_string	Replaces a substring of *this with the string s.
basic_string& replace(size_type pos, size_type n, const basic_string& s, size_type pos1, size_type n1)	basic_string	Replaces a substring of *this with a substring of s.
basic_string& replace(size_type pos, size_type n, const charT* s, size_type n1)	basic_string	Replaces a substring of *this with the first n1 characters of s.
basic_string& replace(size_type pos, size_type n, const charT* s)	basic_string	Replaces a substring of *this with a null-terminated character array.
basic_string& replace(size_type pos, size_type n, size_type n1, charT c)	basic_string	Replaces a substring of *this with n1 copies of c.
basic_string& replace(iterator first, iterator last, const basic_string& s)	basic_string	Replaces a substring of *this with the string s.
basic_string& replace(iterator first, iterator last, const charT* s, size_type n)	basic_string	Replaces a substring of *this with the first n characters of s.
basic_string& replace(iterator first, iterator last, const charT* s)	basic_string	Replaces a substring of *this with a null-terminated character array.
basic_string& replace(iterator first, iterator last, size_type n, charT c)	basic_string	Replaces a substring of *this with n copies of c.

Member	Where defined	Description
template <class InputIterator> basic_string& replace(iterator first, iterator last, InputIterator f, InputIterator l)	basic_string	Replaces a substring of *this with the range [f, l)
size_type copy(charT* buf, size_type n, size_type pos = 0) const	basic_string	Copies a substring of *this to a buffer.
size_type find(const basic_string& s, size_type pos = 0) const	basic_string	Searches for s as a substring of *this, beginning at character pos of *this.
size_type find(const charT* s, size_type pos, size_type n) const	basic_string	Searches for the first n characters of s as a substring of *this, beginning at character pos of *this.
size_type find(const charT* s, size_type pos = 0) const	basic_string	Searches for a null-terminated character array as a substring of *this, beginning at character pos of *this.
size_type find(charT c, size_type pos = 0) const	basic_string	Searches for the character c, beginning at character position pos.
size_type rfind(const basic_string& s, size_type pos = npos) const	basic_string	Searches backward for s as a substring of *this, beginning at character position min(pos, size())
size_type rfind(const charT* s, size_type pos, size_type n) const	basic_string	Searches backward for the first n characters of s as a substring of *this, beginning at character position min(pos, size())
size_type rfind(const charT* s, size_type pos = npos) const	basic_string	Searches backward for a null-terminated character array as a substring of *this, beginning at character min(pos, size())
size_type rfind(charT c, size_type pos = npos) const	basic_string	Searches backward for the character c, beginning at character position min(pos, size().
size_type find_first_of(const basic_string& s, size_type pos = 0) const	basic_string	Searches within *this, beginning at pos, for the first character that is equal to any character within s.
size_type find_first_of(const charT* s, size_type pos, size_type n) const	basic_string	Searches within *this, beginning at pos, for the first character that is equal to any character within the first n characters of s.
size_type find_first_of(const charT* s, size_type pos = 0) const	basic_string	Searches within *this, beginning at pos, for the first character that is equal to any character within s.
size_type find_first_of(charT c, size_type pos = 0) const	basic_string	Searches within *this, beginning at pos, for the first character that is equal to c.
size_type find_first_not_of(const basic_string& s, size_type pos = 0) const	basic_string	Searches within *this, beginning at pos, for the first character that is not equal to any character within s.

Member	Where defined	Description
`size_type find_first_not_of(const charT* s, size_type pos, size_type n) const`	`basic_string`	Searches within `*this`, beginning at `pos`, for the first character that is not equal to any character within the first `n` characters of `s`.
`size_type find_first_not_of(const charT* s, size_type pos = 0) const`	`basic_string`	Searches within `*this`, beginning at `pos`, for the first character that is not equal to any character within `s`.
`size_type find_first_not_of(charT c, size_type pos = 0) const`	`basic_string`	Searches within `*this`, beginning at `pos`, for the first character that is not equal to `c`.
`size_type find_last_of(const basic_string& s, size_type pos = npos) const`	`basic_string`	Searches backward within `*this`, beginning at `min(pos, size())`, for the first character that is equal to any character within `s`.
`size_type find_last_of(const charT* s, size_type pos, size_type n) const`	`basic_string`	Searches backward within `*this`, beginning at `min(pos, size())`, for the first character that is equal to any character within the first `n` characters of `s`.
`size_type find_last_of(const charT* s, size_type pos = npos) const`	`basic_string`	Searches backward `*this`, beginning at `min(pos, size())`, for the first character that is equal to any character within `s`.
`size_type find_last_of(charT c, size_type pos = npos) const`	`basic_string`	Searches backward `*this`, beginning at `min(pos, size())`, for the first character that is equal to `c`.
`size_type find_last_not_of(const basic_string& s, size_type pos = npos) const`	`basic_string`	Searches backward within `*this`, beginning at `min(pos, size())`, for the first character that is not equal to any character within `s`.
`size_type find_last_not_of(const charT* s, size_type pos, size_type n) const`	`basic_string`	Searches backward within `*this`, beginning at `min(pos, size())`, for the first character that is not equal to any character within the first `n` characters of `s`.
`size_type find_last_not_of(const charT* s, size_type pos = npos) const`	`basic_string`	Searches backward `*this`, beginning at `min(pos, size())`, for the first character that is not equal to any character within `s`.
`size_type find_last_not_of(charT c, size_type pos = npos) const`	`basic_string`	Searches backward `*this`, beginning at `min(pos, size())`, for the first character that is not equal to `c`.
`basic_string substr(size_type pos = 0, size_type n = npos) const`	`basic_string`	Returns a substring of `*this`.
`int compare(const basic_string& s) const`	`basic_string`	Three-way lexicographical comparison of `s` and `*this`.
`int compare(size_type pos, size_type n, const basic_string& s) const`	`basic_string`	Three-way lexicographical comparison of `s` and a substring of `*this`.

Member	Where defined	Description
int compare(size_type pos, size_type n, const basic_string& s, size_type pos1, size_type n1) const	basic_string	Three-way lexicographical comparison of a substring of s and a substring of *this.
int compare(const charT* s) const	basic_string	Three-way lexicographical comparison of s and *this.
int compare(size_type pos, size_type n, const charT* s, size_type len = npos) const	basic_string	Three-way lexicographical comparison of the first min(len, traits::length(s) characters of s and a substring of *this.
template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, const basic_string<charT, traits, Alloc>& s2)	basic_string	String concatenation. A global function, not a member function.
template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const charT* s1, const basic_string<charT, traits, Alloc>& s2)	basic_string	String concatenation. A global function, not a member function.
template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, const charT* s2)	basic_string	String concatenation. A global function, not a member function.
template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(charT c, const basic_string<charT, traits, Alloc>& s2)	basic_string	String concatenation. A global function, not a member function.
template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, charT c)	basic_string	String concatenation. A global function, not a member function.
template <class charT, class traits, class Alloc> bool operator==(const basic_string<charT, traits, Alloc>& s1, const basic_string<charT, traits, Alloc>& s2)	Container	String equality. A global function, not a member function.

Member	Where defined	Description
`template <class charT, class traits, class Alloc> bool operator==(const charT* s1, const basic_string<charT, traits, Alloc>& s2)`	`basic_string`	String equality. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator==(const basic_string<charT, traits, Alloc>& s1, const charT* s2)`	`basic_string`	String equality. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator!=(const basic_string<charT, traits, Alloc>& s1, const basic_string<charT, traits, Alloc>& s2)`	Container	String inequality. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator!=(const charT* s1, const basic_string<charT, traits, Alloc>& s2)`	`basic_string`	String inequality. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator!=(const basic_string<charT, traits, Alloc>& s1, const charT* s2)`	`basic_string`	String inequality. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator<(const basic_string<charT, traits, Alloc>& s1, const basic_string<charT, traits, Alloc>& s2)`	Container	String comparison. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator<(const charT* s1, const basic_string<charT, traits, Alloc>& s2)`	`basic_string`	String comparison. A global function, not a member function.
`template <class charT, class traits, class Alloc> bool operator<(const basic_string<charT, traits, Alloc>& s1, const charT* s2)`	`basic_string`	String comparison. A global function, not a member function.
`template <class charT, class traits, class Alloc> void swap(basic_string<charT, traits, Alloc>& s1, basic_string<charT, traits, Alloc>& s2)`	Container	Swaps the contents of two strings.

Member	Where defined	Description
`template <class charT, class traits, class Alloc> basic_istream<charT, traits> operator>>(basic_istream<charT, traits>& is, basic_string<charT, traits, Alloc>& s)`	`basic_string`	Reads `s` from the input stream `is`
`template <class charT, class traits, class Alloc> basic_ostream<charT, traits> operator<<(basic_istream<charT, traits>& os, const basic_string<charT, traits, Alloc>& s)`	`basic_string`	Writes `s` to the output stream `os`
`template <class charT, class traits, class Alloc> basic_istream<charT, traits> getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Alloc>& s, charT delim)`	`basic_string`	Reads a string from the input stream `is`, stopping when it reaches `delim`
`template <class charT, class traits, class Alloc> basic_istream<charT, traits> getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Alloc>& s)`	`basic_string`	Reads a single line from the input stream `is`

New members

These members are not defined in the Random Access Container and Sequence: requirements, but are specific to `basic_string`.

Member	Description
`static const size_type npos`	The largest possible value of type `size_type`. That is, `size_type(-1)`.
`size_type length() const`	Equivalent to `size()`.
`size_type capacity() const`	Number of elements for which memory has been allocated. That is, the size to which the string can grow before memory must be reallocated. `capacity()` is always greater than or equal to `size()`.
`const charT* c_str() const`	Returns a pointer to a null-terminated array of characters representing the string's contents. For any string `s` it is guaranteed that the first `s.size()` characters in the array pointed to by `s.c_str()` are equal to the character in `s`, and that `s.c_str()[s.size()]` is a null character. Note, however, that it not necessarily the first null character. Characters within a string are permitted to be null.
`const charT* data() const`	Returns a pointer to an array of characters, not necessarily null-terminated, representing the string's contents. `data()` is permitted, but not required, to be identical to `c_str()`. The first `size()` characters of that array are guaranteed to be identical to the characters in `*this`. The return value of `data()` is never a null pointer, even if `size()` is zero.
`basic_string(const basic_string& s, size_type pos = 0, size_type n = npos)`	Constructs a string from a substring of `s`. The substring begins at character position `pos` and terminates at character position `pos + n` or at the end of `s`, whichever comes first. This constructor throws `out_of_range` if `pos > s.size()`. Note that when `pos` and `n` have their default values, this is just a copy constructor.
`basic_string(const charT* s)`	Equivalent to `basic_string(s, s + traits::length(s))`.
`basic_string(const charT* s, size_type n)`	Equivalent to `basic_string(s, s + n)`.
`basic_string& operator=(const charT* s)`	Equivalent to `operator=(basic_string(s))`.
`basic_string& operator=(charT c)`	Assigns to `*this` a string whose size is `1` and whose contents is the single character `c`.
`void reserve(size_t n)`	Requests that the string's capacity be changed; the postcondition for this member function is that, after it is called, `capacity() >= n`. You may request that a string decrease its capacity by calling `reserve()` with an argument less than the current capacity. (If you call `reserve()` with an argument less than the string's size, however, the capacity will only be reduced to `size()`. A string's size can never be greater than its capacity.) `reserve()` throws `length_error` if `n > max_size()`.
`basic_string& insert(size_type pos, const basic_string& s)`	If `pos > size()`, throws `out_of_range`. Otherwise, equivalent to `insert(begin() + pos, s.begin(), s.end())`.

Member	Description
`basic_string& insert(size_type pos, const basic_string& s, size_type pos1, size_type n)`	If `pos > size()` or `pos1 > s.size()`, throws `out_of_range`. Otherwise, equivalent to `insert(begin() + pos, s.begin() + pos1, s.begin() + pos1 + min(n, s.size() - pos1))`.
`basic_string& insert(size_type pos, const charT* s)`	If `pos > size()`, throws `out_of_range`. Otherwise, equivalent to `insert(begin() + pos, s, s + traits::length(s))`
`basic_string& insert(size_type pos, const charT* s, size_type n)`	If `pos > size()`, throws `out_of_range`. Otherwise, equivalent to `insert(begin() + pos, s, s + n)`.
`basic_string& insert(size_type pos, size_type n, charT c)`	If `pos > size()`, throws `out_of_range`. Otherwise, equivalent to `insert(begin() + pos, n, c)`.
`basic_string& append(const basic_string& s)`	Equivalent to `insert(end(), s.begin(), s.end())`.
`basic_string& append(const basic_string& s, size_type pos, size_type n)`	If `pos > s.size()`, throws `out_of_range`. Otherwise, equivalent to `insert(end(), s.begin() + pos, s.begin() + pos + min(n, s.size() - pos))`.
`basic_string& append(const charT* s)`	Equivalent to `insert(end(), s, s + traits::length(s))`.
`basic_string& append(const charT* s, size_type n)`	Equivalent to `insert(end(), s, s + n)`.
`basic_string& append(size_type n, charT c)`	Equivalent to `insert(end(), n, c)`.
`template <class InputIterator> basic_string& append(InputIterator first, InputIterator last)`	Equivalent to `insert(end(), first, last)`.
`void push_back(charT c)`	Equivalent to `insert(end(), c)`
`basic_string& operator+=(const basic_string& s)`	Equivalent to `append(s)`.
`basic_string& operator+=(const charT* s)`	Equivalent to `append(s)`
`basic_string& operator+=(charT c)`	Equivalent to `push_back(c)`
`basic_string& erase(size_type pos = 0, size_type n = npos)`	If `pos > size()`, throws `out_of_range`. Otherwise, equivalent to `erase(begin() + pos, begin() + pos + min(n, size() - pos))`.
`basic_string& assign(const basic_string& s)`	Synonym for `operator=`
`basic_string& assign(const basic_string& s, size_type pos, size_type n)`	Equivalent to (but probably faster than) `clear()` followed by `insert(0, s, pos, n)`.
`basic_string& assign(const charT* s, size_type n)`	Equivalent to (but probably faster than) `clear()` followed by `insert(0, s, n)`.

Member	Description
`basic_string& assign(const charT* s)`	Equivalent to (but probably faster than) `clear()` followed by `insert(0, s)`.
`basic_string& replace(size_type pos, size_type n, const basic_string& s)`	Equivalent to `erase(pos, n)` followed by `insert(pos, s)`.
`basic_string& replace(size_type pos, size_type n, const basic_string& s, size_type pos1, size_type n1)`	Equivalent to `erase(pos, n)` followed by `insert(pos, s, pos1, n1)`.
`basic_string& replace(size_type pos, size_type n, const charT* s, size_type n1)`	Equivalent to `erase(pos, n)` followed by `insert(pos, s, n1)`.
`basic_string& replace(size_type pos, size_type n, const charT* s)`	Equivalent to `erase(pos, n)` followed by `insert(pos, s)`.
`basic_string& replace(size_type pos, size_type n, size_type n1, charT c)`	Equivalent to `erase(pos, n)` followed by `insert(pos, n1, c)`.
`basic_string& replace(iterator first, iterator last, const basic_string& s)`	Equivalent to `insert(erase(first, last), s.begin(), s.end())`.
`basic_string& replace(iterator first, iterator last, const charT* s, size_type n)`	Equivalent to `insert(erase(first, last), s, s + n)`.
`basic_string& replace(iterator first, iterator last, const charT* s)`	Equivalent to `insert(erase(first, last), s, s + traits::length(s))`.
`basic_string& replace(iterator first, iterator last, size_type n, charT c)`	Equivalent to `insert(erase(first, last), n, c)`.
`template <class InputIterator> basic_string& replace(iterator first, iterator last, InputIterator f, InputIterator l)`	Equivalent to `insert(erase(first, last), f, l)`.
`size_type copy(charT* buf, size_type n, size_type pos = 0) const`	Copies at most `n` characters from `*this` to a character array. Throws `out_of_range` if `pos > size()`. Otherwise, equivalent to `copy(begin() + pos, begin() + pos + min(n, size()), buf)`. Note that this member function does nothing other than copy characters from `*this` to `buf`; in particular, it does not terminate `buf` with a null character.

Member	Description
`size_type find(const basic_string& s, size_type pos = 0) const`	Searches for `s` as a substring of `*this`, beginning at character position `pos`. It is almost the same as `search`, except that `search` tests elements for equality using `operator==` or a user-provided function object, while this member function uses `traits::eq`. Returns the lowest character position `N` such that `pos <= N` and `pos + s.size() <= size()` and such that, for every `i` less than `s.size()`, `(*this)[N + i]` compares equal to `s[i]`. Returns `npos` if no such position `N` exists. Note that it is legal to call this member function with arguments such that `s.size() > size() - pos`, but such a search will always fail.
`size_type find(const charT* s, size_type pos, size_type n) const`	Searches for the first `n` characters of `s` as a substring of `*this`, beginning at character `pos` of `*this`. This is equivalent to `find(basic_string(s, n), pos)`.
`size_type find(const charT* s, size_type pos = 0) const`	Searches for a null-terminated character array as a substring of `*this`, beginning at character `pos` of `*this`. This is equivalent to `find(basic_string(s), pos)`.
`size_type find(charT c, size_type pos = 0) const`	Searches for the character `c`, beginning at character position `pos`. That is, returns the first character position `N` greater than or equal to `pos`, and less than `size()`, such that `(*this)[N]` compares equal to `c`. Returns `npos` if no such character position `N` exists.
`size_type rfind(const basic_string& s, size_type pos = npos) const`	Searches backward for `s` as a substring of `*this`. It is almost the same as `find_end`, except that `find_end` tests elements for equality using `operator==` or a user-provided function object, while this member function uses `traits::eq`. This member function returns the largest character position `N` such that `N <= pos` and `N + s.size() <= size()`, and such that, for every `i` less than `s.size()`, `(*this)[N + i]` compares equal to `s[i]`. Returns `npos` if no such position `N` exists. Note that it is legal to call this member function with arguments such that `s.size() > size()`, but such a search will always fail.
`size_type rfind(const charT* s, size_type pos, size_type n) const`	Searches backward for the first `n` characters of `s` as a substring of `*this`. Equivalent to `rfind(basic_string(s, n), pos)`.
`size_type rfind(const charT* s, size_type pos = npos) const`	Searches backward for a null-terminated character array as a substring of `*this`. Equivalent to `rfind(basic_string(s), pos)`.
`size_type rfind(charT c, size_type pos = npos) const`	Searches backward for the character `c`. That is, returns the largest character position `N` such that `N <= pos` and `N < size()`, and such that `(*this)[N]` compares equal to `c`. Returns `npos` if no such character position exists.

Member	Description
size_type find_first_of(const basic_string& s, size_type pos = 0) const	Searches within *this, beginning at pos, for the first character that is equal to any character within s. This is similar to the standard algorithm find_first_of, but differs because find_first_of compares characters using operator== or a user-provided function object, while this member function uses traits::eq. Returns the smallest character position N such that pos <= N < size(), and such that (*this)[N] compares equal to some character within s. Returns npos if no such character position exists.
size_type find_first_of(const charT* s, size_type pos, size_type n) const	Searches within *this, beginning at pos, for the first character that is equal to any character within the range [s, s+n). That is, returns the smallest character position N such that pos <= N < size(), and such that (*this)[N] compares equal to some character in [s, s+n). Returns npos if no such character position exists.
size_type find_first_of(const charT* s, size_type pos = 0) const	Equivalent to find_first_of(s, pos, traits::length(s)).
size_type find_first_of(charT c, size_type pos = 0) const	Equivalent to find(c, pos).
size_type find_first_not_of(const basic_string& s, size_type pos = 0) const	Searches within *this, beginning at pos, for the first character that is not equal to any character within s. Returns the smallest character position N such that pos <= N < size(), and such that (*this)[N] does not compare equal to any character within s. Returns npos if no such character position exists.
size_type find_first_not_of(const charT* s, size_type pos, size_type n) const	Searches within *this, beginning at pos, for the first character that is not equal to any character within the range [s, s+n). That is, returns the smallest character position N such that pos <= N < size(), and such that (*this)[N] does not compare equal to any character in [s, s+n). Returns npos if no such character position exists.
size_type find_first_not_of(const charT* s, size_type pos = 0) const	Equivalent to find_first_not_of(s, pos, traits::length(s)).
size_type find_first_not_of(charT c, size_type pos = 0) const	Returns the smallest character position N such that pos <= N < size(), and such that (*this)[N] does not compare equal to c. Returns npos if no such character position exists.
size_type find_last_of(const basic_string& s, size_type pos = npos) const	Searches backward within *this for the first character that is equal to any character within s. That is, returns the largest character position N such that N <= pos and N < size(), and such that (*this)[N] compares equal to some character within s. Returns npos if no such character position exists.

Member	Description
`size_type find_last_of(const charT* s, size_type pos, size_type n) const`	Searches backward within `*this` for the first character that is equal to any character within the range `[s, s+n)`. That is, returns the largest character position `N` such that `N <= pos` and `N < size()`, and such that `(*this)[N]` compares equal to some character within `[s, s+n)`. Returns `npos` if no such character position exists.
`size_type find_last_of(const charT* s, size_type pos = npos) const`	Equivalent to `find_last_of(s, pos, traits::length(s))`.
`size_type find_last_of(charT c, size_type pos = npos) const`	Equivalent to `rfind(c, pos)`.
`size_type find_last_not_of(const basic_string& s, size_type pos = npos) const`	Searches backward within `*this` for the first character that is not equal to any character within `s`. That is, returns the largest character position `N` such that `N <= pos` and `N < size()`, and such that `(*this)[N]` does not compare equal to any character within `s`. Returns `npos` if no such character position exists.
`size_type find_last_not_of(const charT* s, size_type pos, size_type n) const`	Searches backward within `*this` for the first character that is not equal to any character within `[s, s+n)`. That is, returns the largest character position `N` such that `N <= pos` and `N < size()`, and such that `(*this)[N]` does not compare equal to any character within `[s, s+n)`. Returns `npos` if no such character position exists.
`size_type find_last_not_of(const charT* s, size_type pos = npos) const`	Equivalent to `find_last_of(s, pos, traits::length(s))`.
`size_type find_last_not_of(charT c, size_type pos = npos) const`	Searches backward `*this` for the first character that is not equal to `c`. That is, returns the largest character position `N` such that `N <= pos` and `N < size()`, and such that `(*this)[N]` does not compare equal to `c`.
`basic_string substr(size_type pos = 0, size_type n = npos) const`	Equivalent to `basic_string(*this, pos, n)`.
`int compare(const basic_string& s) const`	Three-way lexicographical comparison of `s` and `*this`, much like `strcmp`. If `traits::compare(data, s.data(), min(size(), s.size()))` is nonzero, then it returns that nonzero value. Otherwise returns a negative number if `size() < s.size()`, a positive number if `size() > s.size()`, and zero if the two are equal.
`int compare(size_type pos, size_type n, const basic_string& s) const`	Three-way lexicographical comparison of `s` and a substring of `*this`. Equivalent to `basic_string(*this, pos, n).compare(s)`.

Member	Description
`int compare(size_type pos, size_type n, const basic_string& s, size_type pos1, size_type n1) const`	Three-way lexicographical comparison of a substring of `s` and a substring of `*this`. Equivalent to `basic_string(*this, pos, n).compare(basic_string(s, pos1, n1))`.
`int compare(const charT* s) const`	Three-way lexicographical comparison of `s` and `*this`. Equivalent to `compare(basic_string(s))`.
`int compare(size_type pos, size_type n, const charT* s, size_type len = npos) const`	Three-way lexicographical comparison of the first `min(len, traits::length(s)` characters of `s` and a substring of `*this`. Equivalent to `basic_string(*this, pos, n).compare(basic_string(s, min(len, traits::length(s))))`.
`template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, const basic_string<charT, traits, Alloc>& s2)`	String concatenation. Equivalent to creating a temporary copy of `s`, appending `s2`, and then returning the temporary copy.
`template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const charT* s1, const basic_string<charT, traits, Alloc>& s2)`	String concatenation. Equivalent to creating a temporary `basic_string` object from `s1`, appending `s2`, and then returning the temporary object.
`template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, const charT* s2)`	String concatenation. Equivalent to creating a temporary copy of `s`, appending `s2`, and then returning the temporary copy.
`template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(charT c, const basic_string<charT, traits, Alloc>& s2)`	String concatenation. Equivalent to creating a temporary object with the constructor `basic_string(1, c)`, appending `s2`, and then returning the temporary object.
`template <class charT, class traits, class Alloc> basic_string<charT, traits, Alloc> operator+(const basic_string<charT, traits, Alloc>& s1, charT c)`	String concatenation. Equivalent to creating a temporary object, appending `c` with `push_back`, and then returning the temporary object.
`template <class charT, class traits, class Alloc> bool operator==(const charT* s1, const basic_string<charT, traits, Alloc>& s2)`	String equality. Equivalent to `basic_string(s1).compare(s2) == 0`.

Member	Description
template <class charT, class traits, class Alloc> bool operator==(const basic_string<charT, traits, Alloc>& s1, const charT* s2)	String equality. Equivalent to basic_string(s1).compare(s2) == 0.
template <class charT, class traits, class Alloc> bool operator!=(const charT* s1, const basic_string<charT, traits, Alloc>& s2)	String inequality. Equivalent to basic_string(s1).compare(s2) == 0.
template <class charT, class traits, class Alloc> bool operator!=(const basic_string<charT, traits, Alloc>& s1, const charT* s2)	String inequality. Equivalent to !(s1 == s2).
template <class charT, class traits, class Alloc> bool operator<(const charT* s1, const basic_string<charT, traits, Alloc>& s2)	String comparison. Equivalent to !(s1 == s2).
template <class charT, class traits, class Alloc> bool operator<(const basic_string<charT, traits, Alloc>& s1, const charT* s2)	String comparison. Equivalent to !(s1 == s2).
template <class charT, class traits, class Alloc> basic_istream<charT, traits> operator>>(basic_istream <charT, traits>& is, basic_string<charT, traits, Alloc>& s)	Reads s from the input stream is. Specifically, it skips whitespace, and then replaces the contents of s with characters read from the input stream. It continues reading characters until it encounters a whitespace character (in which case that character is not extracted), or until end-of-file, or, if is.width() is nonzero, until it has read is.width() characters. This member function resets is.width() to zero.
template <class charT, class traits, class Alloc> basic_ostream<charT, traits> operator>>(basic_istream <charT, traits>& is, const basic_string<charT, traits, Alloc>& s)	Writes s to the output stream is. It writes max(s.size(), is.width()) characters, padding as necessary. This member function resets is.width() to zero.
template <class charT, class traits, class Alloc> basic_istream<charT, traits> getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Alloc>& s, charT delim)	Replaces the contents of s with characters read from the input stream. It continues reading characters until it encounters the character delim (in which case that character is extracted but not stored in s), or until end of file. Note that getline, unlike operator>>, does not skip whitespace. As the name suggests, it is most commonly used to read an entire line of text precisely as the line appears in an input file.

Member	Description
`template <class charT, class traits, class Alloc> basic_istream<charT, traits> getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Alloc>& s)`	Equivalent to `getline(is, s, is.widen('\n'))`.

Notes

See also

`vector`, Character Traits

7.2.3 Container adaptors

stack

Description

A `stack` is an adaptor that provides a restricted subset of Container functionality: it provides insertion, removal, and inspection of the element at the top of the stack. `Stack` is a "last in first out" (LIFO) data structure: the element at the top of a `stack` is the one that was most recently added. `Stack` does not allow iteration through its elements. `Stack` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is `deque`, but a different type may be selected explicitly.

Example

```
int main() {
  stack<int> S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

Definition

Defined in the standard header stack, and in the nonstandard backward-compatibility header stack.h.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the stack.	
`Sequence`	The type of the underlying container used to implement the stack.	`deque<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Back Insertion Sequence.
- `Sequence::value_type` is the same type as `T`.
- If `operator==` is used, then `T` is a model of Equality Comparable
- If `operator<` is used, then `T` is a model of LessThan Comparable.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`stack`	See below.
`size_type`	`stack`	See below.
`stack()`	Default Constructible	The default constructor. Creates an empty `stack`.
`stack(const stack&)`	Assignable	The copy constructor.
`stack& operator=(const stack&)`	Assignable	The assignment operator.
`bool empty() const`	`stack`	See below.
`size_type size() const`	`stack`	See below.
`value_type& top()`	`stack`	See below.
`const value_type& top() const`	`stack`	See below.
`void push(const value_type&)`	`stack`	See below.
`void pop()`	`stack`	See below.
`bool operator==(const stack&, const stack&)`	`stack`	See below.
`bool operator<(const stack&, const stack&)`	`stack`	See below.

New members

These members are not defined in the Assignable and Default Constructible requirements, but are specific to `stack`.

Member	Description
`value_type`	The type of object stored in the `stack`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`bool empty() const`	Returns `true` if the `stack` contains no elements, and `false` otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `stack`.
`value_type& top()`	Returns a mutable reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`const value_type& top() const`	Returns a const reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the top of the stack. Postconditions: `size()` will be incremented by `1`, and `top()` will be equal to `x`.
`void pop()`	Removes the element at the top of the stack. Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator==(const stack&, const stack&)`	Compares two stacks for equality. Two stacks are equal if they contain the same number of elements and if they are equal element-by-element. This is a global function, not a member function.
`bool operator<(const stack&, const stack&)`	Lexicographical ordering of two stacks. This is a global function, not a member function.

Notes

Stacks are a standard data structure, and are discussed in all algorithm books. See, for example, section 2.2.1 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.) This restriction is the only reason for `stack` to exist at all. Note that any Front Insertion Sequence or Back Insertion Sequence can be used as a stack; in the case of vector, for example, the stack operations are the member functions `back`, `push_back`, and `pop_back`. The only reason to use the container adaptor `stack` instead is to make it clear that you are performing only stack operations, and no other operations. One might wonder why `pop()` returns `void`, instead of `value_type`. That is, why must one use `top()` and `pop()` to examine and remove the top element, instead of combining the two in a single member function? In fact, there is a good reason for this design. If `pop()` returned the top element, it would have to return by value rather than by reference: return by reference would create a dangling pointer. Return by value, however, is inefficient: it involves at least one redundant copy constructor call. Since it is impossible for `pop()` to return a value in such a way as to be both efficient and correct, it is more sensible for it to return no value at all and to require clients to use `top()` to inspect the value at the top of the stack.

See also

`queue`, `priority_queue`, Container, Sequence

queue

Description

A `queue` is an adaptor that provides a restricted subset of Container functionality A `queue` is a "first in first out" (FIFO) data structure. That is, elements are added to the back of the `queue` and may be removed from the front; `Q.front()` is the element that was added to the `queue` least recently. `Queue` does not allow iteration through its elements. `Queue` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is `deque`, but a different type may be selected explicitly.

Example

```
int main() {
  queue<int> Q;
  Q.push(8);
  Q.push(7);
  Q.push(6);
  Q.push(2);

  assert(Q.size() == 4);
  assert(Q.back() == 2);

  assert(Q.front() == 8);
  Q.pop();

  assert(Q.front() == 7);
  Q.pop();

  assert(Q.front() == 6);
  Q.pop();

  assert(Q.front() == 2);
  Q.pop();

  assert(Q.empty());
}
```

Definition

Defined in the standard header queue, and in the nonstandard backward-compatibility header stack.h.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the queue.	
`Sequence`	The type of the underlying container used to implement the queue.	`deque<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Front Insertion Sequence.
- `Sequence` is a model of Back Insertion Sequence.
- `Sequence::value_type` is the same type as `T`.
- If `operator==` is used, then `T` is a model of

  Equality Comparable
- If `operator<` is used, then `T` is a model of LessThan Comparable.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`queue`	See below.
`size_type`	`queue`	See below.
`queue()`	Default Constructible	The default constructor. Creates an empty `queue`.
`queue(const queue&)`	Assignable	The copy constructor.
`queue& operator=(const queue&)`	Assignable	The assignment operator.
`bool empty() const`	`queue`	See below.
`size_type size() const`	`queue`	See below.
`value_type& front()`	`queue`	See below.
`const value_type& front() const`	`queue`	See below.
`value_type& back()`	`queue`	See below.
`const value_type& back() const`	`queue`	See below.
`void push(const value_type&)`	`queue`	See below.
`void pop()`	`queue`	See below.
`bool operator==(const queue&, const queue&)`	`queue`	See below.
`bool operator<(const queue&, const queue&)`	`queue`	See below.

New members

These members are not defined in the Assignable and Default Constructible requirements, but are specific to `queue`.

Member	Description
`value_type`	The type of object stored in the `queue`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`bool empty() const`	Returns `true` if the `queue` contains no elements, and `false` otherwise. `Q.empty()` is equivalent to `Q.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `queue`.
`value_type& front()`	Returns a mutable reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`const value_type& front() const`	Returns a const reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`value_type& back()`	Returns a mutable reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`const value_type& back() const`	Returns a const reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the back of the queue. Postconditions: `size()` will be incremented by `1`, and `back()` will be equal to `x`.
`void pop()`	Removes the element at the front of the queue. Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator==(const queue&, const queue&)`	Compares two queues for equality. Two queues are equal if they contain the same number of elements and if they are equal element-by-element. This is a global function, not a member function.
`bool operator<(const queue&, const queue&)`	Lexicographical ordering of two queues. This is a global function, not a member function.

Notes

Queues are a standard data structure, and are discussed in all algorithm books. See, for example, section 2.2.1 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.) This restriction is the only reason for `queue` to exist at all. Any container that is both a front insertion sequence and a back insertion sequence can be used as a queue; `deque`, for example, has member functions `front`, `back`, `push_front`, `push_back`, `pop_front`, and `pop_back` The only reason to use the container adaptor `queue` instead of the container `deque` is to make it clear that you are performing only queue operations, and no other operations. One might wonder why `pop()` returns `void`, instead of `value_type`. That is, why must one use `front()` and `pop()` to examine and remove the element at the front of the `queue`, instead of combining the two in a single member function? In fact, there is a good reason for this design. If `pop()` returned the front element, it would have to return by value rather than by reference: return by reference would create a dangling pointer. Return by value, however, is inefficient: it involves at least one redundant copy constructor call. Since it is impossible for `pop()` to return a value in such a way as to be both efficient and correct, it is more sensible for it to return no value at all and to require clients to use `front()` to inspect the value at the front of the `queue`.

See also

`stack`, `priority_queue`, `deque`, Container, Sequence

priority_queue

Description

A `priority_queue` is an adaptor that provides a restricted subset of Container functionality: it provides insertion of elements, and inspection and removal of the top element. It is guaranteed that the top element is the largest element in the `priority_queue`, where the function object `Compare` is used for comparisons. `Priority_queue` does not allow iteration through its elements. `Priority_queue` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is `vector`, but a different type may be selected explicitly.

Example

```
int main() {
  priority_queue<int> Q;
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);

  assert(Q.top() == 8);
  Q.pop();

  assert(Q.top() == 7);
  Q.pop();

  assert(Q.top() == 5);
  Q.pop();

  assert(Q.top() == 4);
  Q.pop();

  assert(Q.top() == 2);
  Q.pop();

  assert(Q.top() == 1);
  Q.pop();

  assert(Q.empty());
}
```

Definition

Defined in the standard header queue, and in the nonstandard backward-compatibility header stack.h.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the priority queue.	
`Sequence`	The type of the underlying container used to implement the priority queue.	`vector<T>`
`Compare`	The comparison function used to determine whether one element is smaller than another element. If `Compare(x,y)` is `true`, then `x` is smaller than `y`. The element returned by `Q.top()` is the largest element in the priority queue. That is, it has the property that, for every other element `x` in the priority queue, `Compare(Q.top(), x)` is `false`.	`less<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Sequence.
- `Sequence` is a model of Random Access Container
- `Sequence::value_type` is the same type as `T`.
- `Compare` is a model of Binary Predicate
- `Compare` induces a strict weak ordering, as defined in the LessThan Comparable requirements, on its argument type.
- `T` is convertible to `Compare`'s argument type.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`priority_queue`	See below.
`size_type`	`priority_queue`	See below.
`priority_queue()`	Default Constructible	The default constructor. Creates an empty `priority_queue`, using `Compare()` as the comparison function.
`priority_queue(const priority_queue&)`	Assignable	The copy constructor.
`priority_queue(const Compare&)`	`priority_queue`	See below.
`priority_queue(const value_type*, const value_type*)`	`priority_queue`	See below.
`priority_queue(const value_type*, const value_type*, const Compare&)`	`priority_queue`	See below.
`priority_queue operator=(const priority_queue&)`	Assignable	The assignment operator.
`bool empty() const`	`priority_queue`	See below.
`size_type size() const`	`priority_queue`	See below.
`const value_type& top() const`	`priority_queue`	See below.
`void push(const value_type&)`	`priority_queue`	See below.
`void pop()`	`priority_queue`	See below.

New members

These members are not defined in the Assignable and Default Constructible requirements, but are specific to `priority_queue`.

Member	Description
`value_type`	The type of object stored in the `priority_queue`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`priority_queue(const Compare& comp)`	The constructor. Creates an empty `priority_queue`, using `comp` as the comparison function. The default constructor uses `Compare()` as the comparison function.
`priority_queue(const value_type* first, const value_type* last)`	The constructor. Creates a `priority_queue` initialized to contain the elements in the range `[first, last)`, and using `Compare()` as the comparison function.
`priority_queue(const value_type* first, const value_type* last, const Compare& comp)`	The constructor. Creates a `priority_queue` initialized to contain the elements in the range `[first, last)`, and using `comp` as the comparison function.
`bool empty() const`	Returns `true` if the `priority_queue` contains no elements, and `false` otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `priority_queue`.
`const value_type& top() const`	Returns a const reference to the element at the top of the priority_queue. The element at the top is guaranteed to be the largest element in the priority queue, as determined by the comparison function `Compare`. That is, for every other element `x` in the `priority_queue`, `Compare(Q.top(), x)` is `false`. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` into the priority_queue. Postcondition: `size()` will be incremented by `1`.
`void pop()`	Removes the element at the top of the priority_queue, that is, the largest element in the priority_queue. Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.

Notes

Priority queues are discussed in all algorithm books; see, for example, section 5.2.3 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1975.) This restriction is the only reason for `priority_queue` to exist at all. If iteration through elements is important, you can either use a `vector` that is maintained in sorted order, or a `set`, or a `vector` that is maintained as a heap using `make_heap`, `push_heap`, and `pop_heap`. `Priority_queue` is, in fact, implemented as a random access container that is maintained as a heap. The only reason to use the container adaptor `priority_queue`, instead of performing the heap operations manually, is to make it clear that you are never performing any operations that might violate the heap invariant. One might wonder why `pop()` returns `void`, instead of `value_type`. That is, why must one use `top()` and `pop()` to examine and remove the element at the top of the `priority_queue`, instead of combining the two in a single member function? In fact, there is a good reason for this design. If `pop()` returned the top element, it would have to return by value rather than by reference: return by reference would create a dangling pointer.

Return by value, however, is inefficient: it involves at least one redundant copy constructor call. Since it is impossible for `pop()` to return a value in such a way as to be both efficient and correct, it is more sensible for it to return no value at all and to require clients to use `top()` to inspect the value at the top of the `priority_queue`.

See also

`stack`, `queue`, `set`, `make_heap`, `push_heap`, `pop_heap`, `is_heap`, `sort`, is_sorted, Container, Sorted Associative Container, Sequence

7.2.4 bitset

Description

`Bitset` is very similar to vector¡bool¿ (also known as bit_vector): it contains a collection of bits, and provides constant-time access to each bit. There are two main differences between `bitset` and `vector<bool>`. First, the size of a `bitset` cannot be changed: `bitset`'s template parameter `N`, which specifies the number of bits in the bitset, must be an integer constant. Second, `bitset` is not a Sequence; in fact, it is not an STL Container at all. It does not have iterators, for example, or `begin()` and `end()` member functions. Instead, `bitset`'s interface resembles that of unsigned integers. It defines bitwise arithmetic operators such as `&=`, `|=`, and `^=`. In general, bit `0` is the least significant bit and bit `N-1` is the most significant bit.

Example

```
int main() {
  const bitset<12> mask(2730ul);
  cout << "mask =      " << mask << endl;

  bitset<12> x;

  cout << "Enter a 12-bit bitset in binary: " << flush;
  if (cin >> x) {
    cout << "x =        " << x << endl;
    cout << "As ulong:  " << x.to_ulong() << endl;
    cout << "And with mask: " << (x & mask) << endl;
    cout << "Or with mask:  " << (x | mask) << endl;
  }
}
```

Definition

Defined in the standard header bitset.

Template parameters

Parameter	Description	Default
`N`	A nonzero constant of type `size_t`: the number of bits that the bitset contains.	

Model of

Assignable, Default Constructible, Equality Comparable

Type requirements

`N` is a constant integer expression of a type convertible to `size_t`, and `N` is a positive number.

Public base classes

None.

Members

Member	Where defined	Description
`reference`	`bitset`	A proxy class that acts as a reference to a single bit.
`bitset()`	Default Constructible	The default constructor. All bits are initially zero.
`bitset(unsigned long val)`	`bitset`	Conversion from unsigned long.
`bitset(const bitset&)`	Assignable	Copy constructor.
`bitset& operator=(const bitset&)`	Assignable	Assignment operator.
`template<class Char, class Traits, class Alloc> explicit bitset(const basic_string<Char,Traits,Alloc>& s, size_t pos = 0, size_t n = basic_string <Char,Traits,Alloc>::npos)`	`bitset`	Conversion from string.
`bitset& operator&=(const bitset&)`	`bitset`	Bitwise and.
`bitset& operator|=(const bitset&)`	`bitset`	Bitwise inclusive or.
`bitset& operator^=(const bitset&)`	`bitset`	Bitwise exclusive or.
`bitset& operator<<=(size_t)`	`bitset`	Left shift.
`bitset& operator>>=(size_t)`	`bitset`	Right shift.
`bitset operator<<(size_t n) const`	`bitset`	Returns a copy of `*this` shifted left by `n` bits.
`bitset operator>>(size_t n) const`	`bitset`	Returns a copy of `*this` shifted right by `n` bits.
`bitset& set()`	`bitset`	Sets every bit.
`bitset& flip()`	`bitset`	Flips the value of every bit.
`bitset operator~() const`	`bitset`	Returns a copy of `*this` with all of its bits flipped.
`bitset& reset()`	`bitset`	Clears every bit.
`bitset& set(size_t n, int val = 1)`	`bitset`	Sets bit `n` if `val` is nonzero, and clears bit `n` if `val` is zero.
`bitset& reset(size_t n)`	`bitset`	Clears bit `n`.
`bitset flip(size_t n)`	`bitset`	Flips bit `n`.
`size_t size() const`	`bitset`	Returns `N`.
`size_t count() const`	`bitset`	Returns the number of bits that are set.
`bool any() const`	`bitset`	Returns `true` if any bits are set.
`bool none() const`	`bitset`	Returns `true` if no bits are set.
`bool test(size_t n) const`	`bitset`	Returns `true` if bit `n` is set.
`reference operator[](size_t n)`	`bitset`	Returns a `reference` to bit `n`.
`bool operator[](size_t n) const`	`bitset`	Returns `true` if bit `n` is set.
`unsigned long to_ulong() const`	`bitset`	Returns an `unsigned long` corresponding to the bits in `*this`.

Member	Where defined	Description
`template<class Char, class Traits, class Alloc> basic_string<Char,Traits,Alloc> to_string() const`	`bitset`	Returns a string representation of `*this`.
`bool operator==(const bitset&) const`	Equality Comparable	The equality operator.
`bool operator!=(const bitset&) const`	Equality Comparable	The inequality operator.
`bitset operator&(const bitset&, const bitset&)`	`bitset`	Bitwise and of two bitsets. This is a global function, not a member function.
`bitset operator|(const bitset&, const bitset&)`	`bitset`	Bitwise or of two bitsets. This is a global function, not a member function.
`bitset operator^(const bitset&, const bitset&)`	`bitset`	Bitwise exclusive or of two bitsets. This is a global function, not a member function.
`template <class Char, class Traits, size_t N> basic_istream<Char,Traits> operator>> (basic_istream<Char,Traits>&, bitset<N>&)`	`bitset`	Extract a `bitset` from an input stream.
`template <class Char, class Traits, size_t N> basic_ostream<Char,Traits> operator>> (basic_ostream<Char,Traits>&, const bitset<N>&)`	`bitset`	Output a `bitset` to an output stream.

New members

These members are not defined in the Assignable, Default Constructible, or Equality Comparable requirements, but are specific to `bitset`.

Member	Description
reference	A proxy class that acts as a reference to a single bit. It contains an assignment operator, a conversion to bool, an operator~, and a member function flip. It exists only as a helper class for bitset's operator[]. That is, it supports the expressions x = b[i], b[i] = x, b[i] = b[j], x = ~b[i], and b[i].flip(). (Where b is a bitset and x is a bool.)
bitset(unsigned long val)	Conversion from unsigned long. Constructs a bitset, initializing the first min(N, sizeof(unsigned long) * CHAR_BIT) bits to the corresponding bits in val and all other bits, if any, to zero.
template<class Char, class Traits, class Alloc> explicit bitset(const basic_string <Char,Traits,Alloc>& s, size_t pos = 0, size_t n = basic_string <Char,Traits,Alloc>:: npos)	Conversion from string. Constructs a bitset, initializing the first M bits to the corresponding characters in s, where M is defined as min(N, min(s.size() - pos, n)). Note that the *highest* character position in s, not the lowest, corresponds to the least significant bit. That is, character position pos + M - 1 - i corresponds to bit i. So, for example, bitset(string("1101")) is the same as bitset(13ul). This function throws out_of_range if pos > s.size(), and invalid_argument if any of the characters used to initialize the bits are anything other than 0 or 1.
bitset& operator&=(const bitset&)	Bitwise and.
bitset& operator|=(const bitset&)	Bitwise inclusive or.
bitset& operator^=(const bitset&)	Bitwise exclusive or.
bitset& operator<<=(size_t n)	Left shift, where bit 0 is considered the least significant bit. Bit i takes on the previous value of bit i - n, or zero if no such bit exists.
bitset& operator>>=(size_t n)	Right shift, where bit 0 is considered the least significant bit. Bit i takes on the previous value of bit i + n, or zero if no such bit exists.
bitset operator<<(size_t n) const	Returns a copy of *this shifted left by n bits. Note that the expression b << n is equivalent to constructing a temporary copy of b and then using operator<<=.
bitset operator>>(size_t n) const	Returns a copy of *this shifted right by n bits. Note that the expression b >> n is equivalent to constructing a temporary copy of b and then using operator>>=.
bitset& set()	Sets every bit.
bitset& flip()	Flips the value of every bit.
bitset operator~() const	Returns a copy of *this with all of its bits flipped.
bitset& reset()	Clears every bit.
bitset& set(size_t n, int val = 1)	Sets bit n if val is nonzero, and clears bit n if val is zero. Throws out_of_range if n >= N.
bitset& reset(size_t n)	Clears bit n. Throws out_of_range if n >= N.

Member	Description
bitset flip(size_t n)	Flips bit n. Throws out_of_range if n >= N.
size_t size() const	Returns N.
size_t count() const	Returns the number of bits that are set.
bool any() const	Returns true if any bits are set.
bool none() const	Returns true if no bits are set.
bool test(size_t n) const	Returns true if bit n is set. Throws out_of_range if n >= N.
reference operator[](size_t n)	Returns a reference to bit n. Note that reference is a proxy class with an assignment operator and a conversion to bool, which allows you to use operator[] for assignment. That is, you can write both x = b[n] and b[n] = x.
bool operator[](size_t n) const	Returns true if bit n is set.
unsigned long to_ulong() const	Returns an unsigned long corresponding to the bits in *this. Throws overflow_error if it is impossible to represent *this as an unsigned long. (That is, if N is larger than the number of bits in an unsigned long and if any of the high-order bits are set.
template<class Char, class Traits, class Alloc> basic_string <Char,Traits,Alloc> to_string() const	Returns a string representation of *this: each character is 1 if the corresponding bit is set, and 0 if it is not. In general, character position i corresponds to bit position N - 1 - i. Note that this member function relies on two language features, *member templates* and *explicit function template argument specification*, that are not yet universally available; this member function is disabled for compilers that do not support those features. Note also that the syntax for calling this member function is somewhat cumbersome. To convert a bitset b to an ordinary string, you must write b.template to_string<char, char_traits<char>, allocator<char> >()
bitset operator&(const bitset&, const bitset&)	Bitwise and of two bitsets. This is a global function, not a member function. Note that the expression b1 & b2 is equivalent to creating a temporary copy of b1, using operator&=, and returning the temporary copy.
bitset operator|(const bitset&, const bitset&)	Bitwise or of two bitsets. This is a global function, not a member function. Note that the expression b1 | b2 is equivalent to creating a temporary copy of b1, using operator|=, and returning the temporary copy.
bitset operator^(const bitset&, const bitset&)	Bitwise exclusive or of two bitsets. This is a global function, not a member function. Note that the expression b1 ^b2 is equivalent to creating a temporary copy of b1, using operator^=, and returning the temporary copy.

Member	Description
template <class Char, class Traits, size_t N> basic_istream<Char, Traits> operator>> (basic_istream <Char,Traits>& is, bitset<N>& x)	Extract a bitset from an input stream. This function first skips whitespace, then extracts up to N characters from the input stream. It stops either when it has successfully extracted N character, or when extraction fails, or when it sees a character that is something other than 1 (in which case it does not extract that character). It then assigns a value to the bitset in the same way as if it were initializing the bitset from a string. So, for example, if the input stream contains the characters "1100abc", it will assign the value 12ul to the bitset, and the next character read from the input stream will be a.
template <class Char, class Traits, size_t N> basic_ostream<Char,Traits> operator>> (basic_ostream <Char,Traits>& os, const bitset<N>& x)	Output a bitset to an output stream. This function behaves as if it converts the bitset to a string and then writes that string to the output stream. That is, it is equivalent to os << x.template to_string<Char,Traits,allocator<Char> >()

Notes

See also

vector, bit_vector, string

Chapter 8

Iterators

8.1 Introduction

Summary

Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element. Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector and a doubly linked list. The STL defines several different concepts related to iterators, several predefined iterators, and a collection of types and functions for manipulating iterators.

Description

Iterators are in fact not a single concept, but six concepts that form a hierarchy: some of them define only a very restricted set of operations, while others define additional functionality. The five concepts that are actually used by algorithms are Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, and Random Access Iterator. A sixth concept, Trivial Iterator, is introduced only to clarify the definitions of the other iterator concepts. The most restricted sorts of iterators are Input Iterators and Output Iterators, both of which permit "single pass" algorithms but do not necessarily support "multi-pass" algorithms. Input iterators only guarantee read access: it is possible to dereference an Input Iterator to obtain the value it points to, but not it is not necessarily possible to assign a new value through an input iterator. Similarly, Output Iterators only guarantee write access: it is possible to assign a value through an Output Iterator, but not necessarily possible to refer to that value. Forward Iterators are a refinement of Input Iterators and

Output Iterators: they support the Input Iterator and Output Iterator operations and also provide additional functionality. In particular, it is possible to use "multi-pass" algorithms with Forward Iterators. A Forward Iterator may be *constant*, in which case it is possible to access the object it points to but not to to assign a new value through it, or *mutable*, in which case it is possible to do both. Bidirectional Iterators, like Forward Iterators, allow multi-pass algorithms. As the name suggests, they are different in that they support motion in both directions: a Bidirectional Iterator may be incremented to obtain the next element or decremented to obtain the previous element. A Forward Iterator, by contrast, is only required to support forward motion. An iterator used to traverse a singly linked list, for example, would be a Forward Iterator, while an iterator used to traverse a doubly linked list would be a Bidirectional Iterator. Finally, Random Access Iterators allow the operations of pointer arithmetic: addition of arbitrary offsets, subscripting, subtraction of one iterator from another to find a distance, and so on. Most algorithms are expressed not in terms of a single iterator but in terms of a *range* of iterators ; the notation `[first, last)` refers to all of the iterators from `first` up to, but **not including**, `last`. Note that a range may be empty, *i.e.* `first` and `last` may be the same iterator. Note also that if there are `n` iterators in a range, then the notation `[first, last)` represents `n+1` positions. This is crucial: algorithms that operate on `n` things frequently require `n+1` positions. Linear search, for example (`find`) must be able to return some value to indicate that the search was unsuccessful. Sometimes it is important to be able to infer some properties of an iterator: the type of object that is returned when it is dereferenced, for example. There are two different mechanisms to support this sort of inferrence: an older mechanism called Iterator Tags, and a newer mechanism called `iterator_traits` .

Concepts

- Trivial Iterator
- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

Types

- `istream_iterator`
- `ostream_iterator`
- `reverse_iterator`

- `reverse_bidirectional_iterator`
- `insert_iterator`
- `front_insert_iterator`
- `back_insert_iterator`

- `iterator_traits`

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`

Functions

- `distance_type`
- `value_type`
- `iterator_category`

- `distance`
- `advance`

- `inserter`
- `front_inserter`
- `back_inserter`

Notes

Ranges are not a well-defined concept for Trivial Iterators, because a Trivial Iterator cannot be incremented: there is no such thing as a next element. They are also not a well-defined concept for Output Iterators, because it is impossible to compare two Output Iterators for equality. Equality is crucial to the definition of a range, because only by comparing an iterator for equality with the last element is it possible to step through a range. Sometimes the notation `[first, last)` refers to the iterators `first`, `first+1`, ..., `last-1` and sometimes it refers to the objects pointed to by those iterators: `*first`, `*(first+1)`, ..., `*(last-1)`. In most cases it will be obvious from context which of these is meant; where the distinction is important, the notation will be qualified explicitly as "range of iterators" or "range of objects". The `iterator_traits` class relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use `iterator_traits`, and you will instead have to continue using the functions `iterator_category`, `distance_type`, and `value_type`.

See also

8.2 Concepts

8.2.1 Trivial Iterator

Description

A Trivial Iterator is an object that may be dereferenced to refer to some other object. Arithmetic operations (such as increment and comparison) are not guaranteed to be supported.

Refinement of

Assignable, Equality Comparable, Default Constructible

Associated types

Value type	The type of the value obtained by dereferencing a Trivial Iterator

Notation

`X`	A type that is a model of Trivial Iterator
`T`	The value type of `X`
`x, y`	Object of type `X`
`t`	Object of type `T`

Definitions

A type that is a model of Trivial Iterator may be *mutable*, meaning that the values referred to by objects of that type may be modified, or *constant*, meaning that they may not. For example, `int*` is a mutable iterator type and `const int*` is a constant iterator type. If an iterator type is mutable, this implies that its value type is a model of Assignable; the converse, though, is not necessarily true. A Trivial Iterator may have a *singular* value, meaning that the results of most operations, including comparison for equality, are undefined. The only operation that a is guaranteed to be supported is assigning a nonsingular iterator to a singular iterator. A Trivial Iterator may have a *dereferenceable* value, meaning that dereferencing it yields a well-defined value. Dereferenceable iterators are always nonsingular, but the converse is not true. For example, a null pointer is nonsingular (there are well defined operations involving null pointers) even thought it is not dereferenceable. *Invalidating* a dereferenceable iterator means performing an operation after which the iterator might be nondereferenceable or singular. For example, if `p` is a pointer, then `delete p` invalidates `p`.

Valid expressions

In addition to the expressions defined in Assignable, Equality Comparable, and Default Constructible, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Default constructor	`X x`		
Dereference	`*x`		Convertible to `T`
Dereference assignment	`*x = t`	`X` is mutable	
Member access	`x->m`	`T` is a type for which `x.m` is defined	

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default constructor	`X x`			`x` is singular
Dereference	`*x`	`x` is dereferenceable		
Dereference assignment	`*x = t`	`x` is dereferenceable		`*x` is a copy of t
Member access	`x->m`	`x` is dereferenceable	Equivalent to `(*x).m`	

Complexity guarantees

The complexity of operations on trivial iterators is guaranteed to be amortized constant time.

Invariants

Identity	`x == y` if and only if `&*x == &*y`

Models

- A pointer to an object that is not part of an array.

Notes

The requirement for the return type of `*x` is specified as "convertible to `T`", rather than simply `T`, because it sometimes makes sense for an iterator to return some sort of proxy object instead of the object that the iterator conceptually points to. Proxy objects are implementation details rather than part of an interface (one use of them, for example, is to allow an iterator to behave differently depending on whether its value is being read or written), so the value type of an iterator that returns a proxy is still `T`. Defining `operator->` for iterators depends on a feature that is part of the C++ language but that is not yet implemented by all C++ compilers. If your compiler does not yet support this feature, the workaround is to use `(*it).m` instead of `it->m`.

See also

Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, Random Access Iterator, Iterator Overview

8.2.2 Input Iterator

Description

An Input Iterator is an iterator that may be dereferenced to refer to some object, and that may be incremented to obtain the next iterator in a sequence. Input Iterators are not required to be mutable.

Refinement of

Trivial iterator.

Associated types

Value type	The type of the value obtained by dereferencing an Input Iterator
Distance type	A signed integral type used to represent the distance from one iterator to another, or the number of elements in a range.

Notation

`X`	A type that is a model of Input Iterator
`T`	The value type of `X`
`i`, `j`	Object of type `X`
`t`	Object of type `T`

Definitions

An iterator is *past-the-end* if it points beyond the last element of a container. Past-the-end values are nonsingular and nondereferenceable. An iterator is *valid* if it is dereferenceable or past-the-end. An iterator `i` is *incrementable* if there is a "next" iterator, that is, if `++i` is well-defined. Past-the-end iterators are not incrementable. An Input Iterator `j` is *reachable* from an Input Iterator `i` if, after applying `operator++` to `i` a finite number of times, `i == j`. The notation `[i,j)` refers to a *range* of iterators beginning with `i` and up to but not including `j`. The range `[i,j)` is a *valid range* if both `i` and `j` are valid iterators, and `j` is reachable from `i` .

Valid expressions

In addition to the expressions defined in Trivial Iterator, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Preincrement	`++i`		`X&`
Postincrement	`(void)i++`		
Postincrement and dereference	`*i++`		`T`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Dereference	`*t`	`i` is incrementable		
Preincrement	`++i`	`i` is dereferenceable		`i` is dereferenceable or past-the-end
Postincrement	`(void)i++`	`i` is dereferenceable	Equivalent to `(void)++i`	`i` is dereferenceable or past-the-end
Postincrement and dereference	`*i++`	`i` is dereferenceable	Equivalent to `T t = *i; ++i; return t;`	`i` is dereferenceable or past-the-end

Complexity guarantees

All operations are amortized constant time.

Invariants

Models

- istream_iterator

Notes

`i == j` does not imply `++i == ++j`. Every iterator in a valid range `[i, j)` is dereferenceable, and `j` is either dereferenceable or past-the-end. The fact that every iterator in the range is dereferenceable follows from the fact that incrementable iterators must be deferenceable. After executing `++i`, it is not required that copies of the old value of `i` be dereferenceable or that they be in the domain of `operator==`. It is not guaranteed that it is possible to pass through the same input iterator twice.

See also

Output Iterator, Iterator overview

8.2.3 Output Iterator

Description

An Output Iterator is a type that provides a mechanism for storing (but not necessarily accessing) a sequence of values. Output Iterators are in some sense the converse of Input Iterators, but they have a far more restrictive interface: they do not necessarily support member access or equality, and they do not necessarily have either an associated distance type or even a value type . Intuitively, one picture of an Output Iterator is a tape: you can write a value to the current location and you can advance to the next location, but you cannot read values and you cannot back up or rewind.

Refinement of

Assignable, DefaultConstructible

Associated types

None.

Notation

`X`	A type that is a model of Output Iterator
`x`, `y`	Object of type `X`

Definitions

If `x` is an Output Iterator of type `X`, then the expression `*x = t;` stores the value `t` into `x`. Note that `operator=`, like other C++ functions, may be overloaded; it may, in fact, even be a template function. In general, then, `t` may be any of several different types. A type `T` belongs to the *set of value types* of `X` if, for an object `t` of type `T`, `*x = t;` is well-defined and does not require performing any non-trivial conversions on `t`. An Output Iterator may be *singular*, meaning that the results of most operations, including copying and dereference assignment, are undefined. The only operation that is guaranteed to be supported is assigning a nonsingular iterator to a singular iterator. An Output Iterator may be *dereferenceable*, meaning that assignment through it is defined. Dereferenceable iterators are always nonsingular, but nonsingular iterators are not necessarily dereferenceable.

Valid expressions

Name	Expression	Type requirements	Return type
Default constructor	`X x;` `X()`		
Copy constructor	`X(x)`		`X`
Copy constructor	`X y(x);` or `X y = x;`		
Dereference assignment	`*x = t`	`t` is convertible to a type in the set of value types of `X`.	Result is not used
Preincrement	`++x`		`X&`
Postincrement	`(void) x++`		`void`
Postincrement and assign	`*x++ = t;`		Result is not used

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default constructor	`X x;` `X()`			`x` may be singular
Copy constructor	`X(x)`	`x` is nonsingular		`*X(x) = t` is equivalent to `*x = t`
Copy constructor	`X x(y);` or `X x = y;`	`y` is nonsingular		`*y = t` is equivalent to `*x = t`
Dereference assignment	`*x = t`	`x` is dereferenceable. If there has been a previous assignment through `x`, then there has been an intervening increment.		
Preincrement	`++x`	`x` is dereferenceable. `x` has previously been assigned through. If `x` has previously been incremented, then there has been an intervening assignment through `x`		`x` points to the next location into which a value may be stored
Postincrement	`(void) x++`	`x` is dereferenceable. `x` has previously been assigned through.	Equivalent to `(void) ++x`	`x` points to the next location into which a value may be stored
Postincrement and assign	`*x++ = t;`	`x` is dereferenceable. If there has been a previous assignment through `x`, then there has been an intervening increment.	Equivalent to `*x = t; ++x;`	`x` points to the next location into which a value may be stored

Complexity guarantees

The complexity of operations on output iterators is guaranteed to be amortized constant time.

Invariants

Models

- ostream_iterator
- insert_iterator
- front_insert_iterator
- back_insert_iterator

Notes

Other iterator types, including Trivial Iterator and Input Iterator, define the notion of a *value type*, the type returned when an iterator is dereferenced. This notion does not apply to Output Iterators, however, since the dereference operator (unary `operator*`) does not return a usable value for Output Iterators. The only context in which the dereference operator may be used is assignment through an output iterator: `*x = t`. Although Input Iterators and output iterators are roughly symmetrical concepts, there is an important sense in which accessing and storing values are not symmetrical: for an Input Iterator `operator*` must return a unique type, but, for an Output Iterator, in the expression `*x = t`, there is no reason why `operator=` must take a unique type. Consequently, there need not be any unique "value type" for Output Iterators. There should be only one active copy of a single Output Iterator at any one time. That is: after creating and using a copy `x` of an Output Iterator `y`, the original output iterator `y` should no longer be used. Assignment through an Output Iterator `x` is expected to alternate with incrementing `x`, and there must be an assignment through `x` before `x` is ever incremented. Any other order of operations results in undefined behavior. That is: `*x = t; ++x; *x = t2; ++xis acceptable, but` *x = t; `++x; ++x; *x = t2;` is not. Note that an Output Iterator need not define comparison for equality. Even if an `operator==` is defined, `x == y` need not imply `++x == ++y`. If you are implementing an Output Iterator class `X`, one sensible way to define `*x = t` is to define `X::operator*()` to return an object of some private class `X_proxy`, and then to define `X_proxy::operator=`. Note that you may overload `X_proxy::operator=`, or even define it as a member template; this allows assignment of more than one type through Output Iterators of class `X`.

See also

Trivial Iterator, Input Iterator, Iterator overview

8.2.4 Forward Iterator

Description

A Forward Iterator is an iterator that corresponds to the usual intuitive notion of a linear sequence of values. It is possible to use Forward Iterators (unlike Input Iterators and Output Iterators) in multipass algorithms. Forward Iterators do not, however, allow stepping backwards through a sequence, but only, as the name suggests, forward. A type that is a model of Forward Iterator may be either *mutable* or *immutable*, as defined in the Trivial Iterators requirements.

Refinement of

Input Iterator, Output Iterator

Associated types

The same as for Input Iterator

Notation

`X`	A type that is a model of Forward Iterator
`T`	The value type of `X`
`i`, `j`	Object of type `X`
`t`	Object of type `T`

Definitions

Valid expressions

Forward Iterator does not define any new expressions beyond those defined in Input Iterator. However, some of the restrictions described in Input Iterator are relaxed.

Name	Expression	Type reqs	Return type
Preincrement	`++i`		`X&`
Postincrement	`i++`		`X`

Expression semantics

Forward Iterator does not define any new expressions beyond those defined in Input Iterator. However, some of the restrictions described in Input Iterator are relaxed.

Name	Expression	Precondition	Semantics	Postcondition
Preincrement	`++i`	`i` is dereferenceable	`i` points to the next value	`i` is dereferenceable or past-the-end. `&i == &++i`. If `i == j`, then `++i == ++j`.
Postincrement	`i++`	`i` is dereferenceable	Equivalent to `{ X tmp = i; ++i; return tmp; }`	`i` is dereferenceable or past-the-end.

Complexity guarantees

The complexity of operations on Forward Iterators is guaranteed to be amortized constant time.

Invariants

Models

Notes

The restrictions described in Input Iterator have been removed. Incrementing a forward iterator does not invalidate copies of the old value and it is guaranteed that, if `i` and `j` are dereferenceable and `i == j`, then `++i == ++j`. As a consequence of these two facts, it is possible to pass through the same Forward Iterator twice.

See also

Input Iterator, Output Iterator, Bidirectional Iterator, Random Access Iterator, Iterator overview

8.2.5 Bidirectional Iterator

Description

A Bidirectional Iterator is an iterator that can be both incremented and decremented. The requirement that a Bidirectional Iterator can be decremented is the only thing that distinguishes Bidirectional Iterators from Forward Iterators.

Refinement of

Forward Iterator

Associated types

The same as for Forward Iterator.

Notation

`X`	A type that is a model of Bidirectional Iterator
`T`	The value type of `X`
`i`, `j`	Object of type `X`
`t`	Object of type `T`

Definitions

Valid expressions

In addition to the expressions defined in Forward Iterator, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Predecrement	`--i`		`X&`
Postdecrement	`i--`		`X`

Expression Semantics

Semantics of an expression is defined only where it is not defined in Forward Iterator.

Name	Expression	Precondition	Semantics	Postcondition
Predecrement	`--i`	`i` is dereferenceable or past-the-end. There exists a dereferenceable iterator `j` such that `i == ++j`.	`i` is modified to point to the previous element.	`i` is dereferenceable. `&i = &--i`. If `i == j`, then `--i == --j`. If `j` is dereferenceable and `i == ++j`, then `--i == j`.
Postdecrement	`i--`	`i` is dereferenceable or past-the-end. There exists a dereferenceable iterator `j` such that `i == ++j`.	Equivalent to `{ X tmp = i; --i; return tmp; }`	

Complexity guarantees

The complexity of operations on bidirectional iterators is guaranteed to be amortized constant time.

Invariants

Symmetry of increment and decrement	If `i` is dereferenceable, then `++i; --i;` is a null operation. Similarly, `--i; ++i;` is a null operation.

Models

- `T*`
- `list<T>::iterator`

Notes

See also

Input Iterator, Output Iterator, Forward Iterator, Random Access Iterator, Iterator overview

8.2.6 Random Access Iterator

Description

A Random Access Iterator is an iterator that provides both increment and decrement (just like a Bidirectional Iterator), and that also provides constant-time methods for moving forward and backward in arbitrary-sized steps. Random Access Iterators provide essentially all of the operations of ordinary C pointer arithmetic.

Refinement of

Bidirectional Iterator, LessThan Comparable

Associated types

The same as for Bidirectional Iterator

Notation

`X`	A type that is a model of Random Access Iterator
`T`	The value type of `X`
`Distance`	The distance type of `X`
`i`, `j`	Object of type `X`
`t`	Object of type `T`
`n`	Object of type `Distance`

Definitions

Valid expressions

In addition to the expressions defined in Bidirectional Iterator, the following expressions must be valid.

Name	Expression	Type reqs	Return type
Iterator addition	`i += n`		`X&`
Iterator addition	`i + n` or `n + i`		`X`
Iterator subtraction	`i -= n`		`X&`
Iterator subtraction	`i - n`		`X`
Difference	`i - j`		`Distance`
Element operator	`i[n]`		Convertible to `T`
Element assignment	`i[n] = t`	`X` is mutable	Convertible to `T`

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Bidirectional Iterator or LessThan Comparable.

Name	Expression	Precondition	Semantics	Postcondition
Forward motion	`i += n`	Including `i` itself, there must be `n` dereferenceable or past-the-end iterators following or preceding `i`, depending on whether `n` is positive or negative.	If `n > 0`, equivalent to executing `++i` `n` times. If `n < 0`, equivalent to executing `--i` `n` times. If `n == 0`, this is a null operation.	`i` is dereferenceable or past-the-end.
Iterator addition	`i + n` or `n + i`	Same as for `i += n`	Equivalent to `X tmp = i; return tmp += n;` . The two forms `i + n` and `n + i` are identical.	Result is dereferenceable or past-the-end
Iterator subtraction	`i -= n`	Including `i` itself, there must be `n` dereferenceable or past-the-end iterators preceding or following `i`, depending on whether `n` is positive or negative.	Equivalent to `i += (-n)`.	`i` is dereferenceable or past-the-end.
Iterator subtraction	`i - n`	Same as for `i -= n`	Equivalent to `X tmp = i; return tmp -= n;` .	Result is dereferenceable or past-the-end
Difference	`i - j`	Either `i` is reachable from `j` or `j` is reachable from `i`, or both.	Returns a number `n` such that `i == j + n`	
Element operator	`i[n]`	`i + n` exists and is dereferenceable.	Equivalent to `*(i + n)`	
Element assignment	`i[n] = t`	`i + n` exists and is dereferenceable.	Equivalent to `*(i + n) = t`	`i[n]` is a copy of `t`.
Less	`i < j`	Either `i` is reachable from `j` or `j` is reachable from `i`, or both.	As described in LessThan Comparable	

Complexity guarantees

All operations on Random Access Iterators are amortized constant time.

Invariants

Symmetry of addition and subtraction	If `i + n` is well-defined, then `i += n; i -= n;` and `(i + n) - n` are null operations. Similarly, if `i - n` is well-defined, then `i -= n; i += n;` and `(i - n) + n` are null operations.
Relation between distance and addition	If `i - j` is well-defined, then `i == j + (i - j)`.
Reachability and distance	If `i` is reachable from `j`, then `i - j >= 0`.
Ordering	`operator <` is a strict weak ordering, as defined in LessThan Comparable.

Models

- `T*`
- `vector<T>::iterator`
- `vector<T>::const_iterator`
- `deque<T>::iterator`
- `deque<T>::const_iterator`

Notes

"Equivalent to" merely means that `i += n` yields the same iterator as if `i` had been incremented (decremented) `n` times. It does not mean that this is how `operator+=` should be implemented; in fact, this is not a permissible implementation. It is guaranteed that `i += n` is amortized constant time, regardless of the magnitude of `n`. One minor syntactic oddity: in C, if `p` is a pointer and `n` is an int, then `p[n]` and `n[p]` are equivalent. This equivalence is not guaranteed, however, for Random Access Iterators: only `i[n]` need be supported. This isn't a terribly important restriction, though, since the equivalence of `p[n]` and `n[p]` has essentially no application except for obfuscated C contests. The precondition defined in LessThan Comparable is that `i` and `j` be in the domain of `operator <`. Essentially, then, this is a definition of that domain: it is the set of pairs of iterators such that one iterator is reachable from the other. All of the other comparison operators have the same domain and are defined in terms of `operator <`, so they have exactly the same semantics as described in LessThan Comparable. This complexity guarantee is in fact the only reason why Random Access Iterator exists as a distinct concept. Every operation in iterator arithmetic can be defined for Bidirectional Iterators; in fact, that is exactly what the algorithms `advance` and `distance` do. The distinction is simply that the Bidirectional Iterator implementations are linear time, while Random Access Iterators are required to support random access to elements in amortized constant time. This has major implications for the sorts of algorithms that can sensibly be written using the two types of iterators.

See also

LessThan Comparable, Trivial Iterator, Bidirectional Iterator, Iterator overview

8.3 Iterator Tags

8.3.1 Introduction

Summary

Iterator tag functions are a method for accessing information that is associated with iterators. Specifically, an iterator type must, as discussed in the Input Iterator requirements, have an associated *distance type* and *value type*. It is sometimes important for an algorithm parameterized by an iterator type to be able to determine the distance type and value type. Iterator tags also allow algorithms to determine an iterator's category, so that they can take different actions depending on whether an iterator is an Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, or Random Access Iterator. Note that the iterator tag functions `distance_type`, `value_type`, and `iterator_category` are an older method of accessing the type information associated with iterators: they were defined in the original STL. The draft C++ standard, however, defines a different and more convenient mechanism: `iterator_traits`. Both mechanisms are supported , for reasons of backwards compatibility, but the older mechanism will eventually be removed.

Description

The basic idea of the iterator tag functions, and of `iterator_traits`, is quite simple: iterators have associated type information, and there must be a way to access that information. Specifically, iterator tag functions and `iterator_traits` are used to determine an iterator's value type, distance type, and iterator category. An iterator's *category* is the most specific concept that it is a model of: Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, or Random Access Iterator. This information is expressed in the C++ type system by defining five category tag types, `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, and `random_access_iterator_tag`, each of which corresponds to one of those concepts. The function `iterator_category` takes a single argument, an iterator, and returns the tag corresponding to that iterator's category. That is, it returns a `random_access_iterator_tag` if its argument is a pointer, a bidirectional_iterator_tag if its argument is a list::iterator, and so on. Iterator_traits provides the same information in a slightly different way: if I is an iterator, then iterator_traits<I>::iterator_category is a nested typedef: it is one of the five category tag types. An iterator's *value type* is the type of object that is returned when the iterator is dereferenced. (See the discussion in the Input Iterator requirements.) Ideally, one might want value_type to take a single argument, an iterator, and return the iterator's value type. Unfortunately, that's impossible: a function must return an object, and types aren't objects. Instead, value_type returns the value (T*) 0, where T is the argument's value type. The iterator_traits class, however, does not have this restriction: iterator_traits<I>::value_type is a type, not a value. It is a nested typedef, and it can be used in declarations of variables, as an function's argument type or return type, and in any other ways that C++ types can be used. (Note that the function value_type need not be defined for Output Iterators, since an Output Iterator need not have a value type. Similarly, iterator_traits<I>::value_type is typically defined as void when I is an output iterator) An iterator's *distance type*, or *difference type* (the terms are synonymous) is the type that is used to represent the distance between two iterators. (See the discussion in the Input Iterator requirements.) The function distance_type returns this information in the same form that value_type does: its argument is an iterator, and it returns the value (Distance*) 0, where Distance is the iterator's distance type. Similarly, iterator_traits<I>::difference_type is I's distance type. Just as with value_type, the function distance_type need not be defined for Output Iterators, and, if I is an Output Iterator, iterator_traits<I>::difference_type may be defined as void. An Output Iterator need not have a distance type. The functions iterator_category, value_type, and distance_type must be provided for every type of iterator. (Except, as noted above, that value_type and distance_type need not be provided for Output Iterators.) In principle, this is simply a matter of overloading: anyone who defines a new iterator type must define those three functions for it. In practice, there's a slightly more convenient method. The STL defines five base classes, output_iterator, input_iterator, forward_iterator, bidirectional_iterator, and random_access_iterator. The functions iterator_category, value_type, and distance_type are defined for those base classes. The effect, then, is that if you are defining a new type of iterator you can simply derive it from one of those base classes, and the iterator tag functions will automatically be defined correctly. These base classes contain no member functions or member variables, so deriving from one of them ought not to incur any overhead. (Again, note that base classes are provided solely for the convenience of people who define iterators. If you define a class Iter that is a new kind of Bidirectional Iterator, you do not have to derive it from the base class bidirectional_iterator. You do, however, have to make sure that iterator_category, value_type, and distance_type are defined correctly for arguments of type Iter, and deriving Iter from bidirectional_iterator is usually the most convenient way to do that.)

Examples

This example uses the value_type iterator tag function in order to declare a temporary variable of an iterator's value type. Note the use of an auxiliary function, `__iter_swap`. This is a very common idiom: most uses of iterator tags involve auxiliary functions.

```
template <class ForwardIterator1, class ForwardIterator2,
          class ValueType>
inline void __iter_swap(ForwardIterator1 a, ForwardIterator2 b,
                        ValueType*) {
   ValueType tmp = *a;
   *a = *b;
   *b = tmp;
}

template <class ForwardIterator1, class ForwardIterator2>
inline void iter_swap(ForwardIterator1 a, ForwardIterator2 b) {
   __iter_swap(a, b, value_type(a));
}
```

This example does exactly the same thing, using `iterator_traits` instead. Note how much simpler it is: the auxiliary function is no longer required.

```
template <class ForwardIterator1, class ForwardIterator2>
inline void iter_swap(ForwardIterator1 a, ForwardIterator2 b) {
   iterator_traits<ForwardIterator1>::value_type tmp = *a;
   *a = *b;
   *b = tmp;
}
```

This example uses the `iterator_category` iterator tag function: `reverse` can be implemented for either Bidirectional Iterators or for Random Access Iterators, but the algorithm for Random Access Iterators is more efficient. Consequently, `reverse` is written to dispatch on the iterator category. This dispatch takes place at compile time, and should not incur any run-time penalty.

```
template <class BidirectionalIterator>
void __reverse(BidirectionalIterator first, BidirectionalIterator last,
           bidirectional_iterator_tag) {
   while (true)
     if (first == last || first == --last)
        return;
     else
        iter_swap(first++, last);
}

template <class RandomAccessIterator>
void __reverse(RandomAccessIterator first, RandomAccessIterator last,
               random_access_iterator_tag) {
   while (first < last) iter_swap(first++, --last);
}

template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first,
                    BidirectionalIterator last) {
   __reverse(first, last, iterator_category(first));
}
```

In this case, `iterator_traits` would not be different in any substantive way: it would still be necessary to use auxiliary functions to dispatch on the iterator category. The only difference is changing the top-level function to

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first,
                    BidirectionalIterator last) {
   __reverse(first, last,
             iterator_traits<first>::iterator_category());
}
```

Concepts

Types

- `output_iterator`
- `input_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`

- `output_iterator_tag`
- `input_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

- `iterator_traits`

Functions

- `iterator_category`
- `value_type`
- `distance_type`

Notes

Output Iterators have neither a distance type nor a value type; in many ways, in fact, Output Iterators aren't really iterators. Output iterators do not have a value type, because it is impossible to obtain a value from an output iterator but only to write a value through it. They do not have a distance type, similarly, because it is impossible to find the distance from one output iterator to another. Finding a distance requires a comparison for equality, and output iterators do not support `operator==`. The `iterator_traits` class relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use `iterator_traits`, and you will have to continue to use the older iterator tag functions. Note that Trivial Iterator does not appear in this list. The Trivial Iterator concept is introduced solely for conceptual clarity; the STL does not actually define any Trivial Iterator types, so there is no need for a Trivial Iterator tag. There is, in fact, a strong reason not to define one: the C++ type system does not provide any way to distinguish between a pointer that is being used as a trivial iterator (that is, a pointer to an object that isn't part of an array) and a pointer that is being used as a Random Access Iterator into an array.

See also

Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, Random Access Iterator, `iterator_traits`, Iterator Overview

8.3.2 iterator_traits

Description

As described in the Iterator Overview, one of the most important facts about iterators is that they have associated types. An iterator type, for example, has an associated *value type*: the type of object that the iterator points to. It also has an associated *distance type*, or *difference type*, a signed integral type that can be used to represent the distance between two iterators. (Pointers, for example, are iterators; the value type of `int*` is `int`. Its distance type is `ptrdiff_t`, because, if `p1` and `p2` are pointers, the expression `p1 - p2` has type `ptrdiff_t`.) Generic algorithms often need to have access to these associated types; an algorithm that takes a range of iterators, for example, might need to declare a temporary variable whose type is the iterators' value type. The class `iterator_traits` is a mechanism that allows such declarations. The most obvious way to allow declarations of that sort would be to require that all iterators declare nested types; an iterator `I`'s value type, for example, would be `I::value_type`. That can't possibly work, though. Pointers are iterators, and pointers aren't classes; if `I` is (say) `int*`, then it's impossible to define `I::value_type` to be `int`. Instead, `I`'s value type is written `iterator_traits<I>::value_type`. `iterator_traits` is a template class that contains nothing but nested `typedef`s; in addition to `value_type`, `iterator_traits` defines the nested types `iterator_category`, `difference_type`, `pointer`, and `reference`. The library contains two definitions of `iterator_traits`: a fully generic one, and a specialization that is used whenever the template argument is a pointer type . The fully generic version defines `iterator_traits<I>::value_type` as a synonym for `I::value_type`, `iterator_traits<I>::difference_type` as a synonym for `I::difference_type`, and so on. Since pointers don't have nested types, `iterator_traits<T*>` has a different definition.

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type        value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer           pointer;
  typedef typename Iterator::reference         reference;
};

template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

If you are defining a new iterator type `I`, then you must ensure that `iterator_traits<I>` is defined properly. There are two ways to do this. First, you can define your iterator so that it has nested types `I::value_type`, `I::difference_type`, and so on. Second, you can explicitly specialize `iterator_traits` for your type. The first way is almost always more convenient, however, especially since you can easily ensure that your iterator has the appropriate nested types just by inheriting from one of the base classes `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, or `random_access_iterator`. Note that `iterator_traits` is new; it was added to the draft C++ standard relatively recently. Both the old iterator tags mechanism and the new `iterator_traits` mechanism are currently supported , but the old iterator tag functions are no longer part of the standard C++ library and they will eventually be removed.

Example

This generic function returns the last element in a non-empty range. Note that there is no way to define a function with this interface in terms of the old `value_type` function, because the function's return type must be declared to be the iterator's value type.

```
template <class InputIterator>
iterator_traits<InputIterator>::value_type
last_value(InputIterator first, InputIterator last) {
  iterator_traits<InputIterator>::value_type result = *first;
  for (++first; first != last; ++first)
    result = *first;
  return result;
}
```

(Note: this is an example of how to use `iterator_traits`; it is not an example of good code. There are better ways of finding the last element in a range of bidirectional iterators, or even forward iterators.)

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`Iterator`	The iterator type whose associated types are being accessed.	

Model of

Default Constructible, Assignable

Type requirements

- `Iterator` is a model of one of the iterator concepts. (Input Iterator, Output Iterator, Forward Iterator,

  Bidirectional Iterator, or Random Access Iterator.)

Public base classes

None.

Members

None, except for nested types.

Member	Description
`iterator_category`	One of the types `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, or `random_access_iterator_tag`. An iterator's category is the *most specific* iterator concept that it is a model of.
`value_type`	`Iterator`'s value type, as defined in the Trivial Iterator requirements.
`difference_type`	`Iterator`'s distance type, as defined in the Input Iterator requirements.
`pointer`	`Iterator`'s pointer type: a pointer to its value type.
`reference`	`Iterator`'s reference type: a reference to its value type.

Notes

The `iterator_traits` class relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use `iterator_traits`, and you will have to continue using the old iterator tag functions `iterator_category`, `distance_type`, and `value_type`. This is one reason that those functions have not yet been removed.

See also

The iterator overview, iterator tags, `input_iterator_tag`, output_iterator_tag, `forward_iterator_tag`, bidirectional_iterator_tag, `random_access_iterator_tag`, input_iterator, `output_iterator`, forward_iterator, `bidirectional_iterator`, random_access_iterator

8.3.3 Iterator tag classes

input_iterator_tag

Description

`Input_iterator_tag` is an empty class: it has no member functions, member variables, or nested types. It is used solely as a "tag": a representation of the Input Iterator concept within the C++ type system. Specifically, it is used as a return value for the function `iterator_category`. `Iterator_category` takes a single argument, an iterator, and returns an object whose type depends on the iterator's category. `Iterator_category`'s return value is of type `input_iterator_tag` if its argument is an Input Iterator.

Example

See `iterator_category`

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

None.

Model of

Assignable

Type requirements

None.

Public base classes

None.

Members

None.

New Members

None.

Notes

See also

`iterator_category`, Iterator Tags, `iterator_traits`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, `random_access_iterator_tag`

output_iterator_tag

Description

`Output_iterator_tag` is an empty class: it has no member functions, member variables, or nested types. It is used solely as a "tag": a representation of the Output Iterator concept within the C++ type system. Specifically, it is used as a return value for the function `iterator_category`. `Iterator_category` takes a single argument, an iterator, and returns an object whose type depends on the iterator's category. `Iterator_category`'s return value is of type `output_iterator_tag` if its argument is an Output Iterator.

Example

See `iterator_category`

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

None.

Model of

Assignable

Type requirements

None.

Public base classes

None.

Members

None.

New Members

None.

Notes

See also

`iterator_category`, Iterator Tags, `iterator_traits`, `input_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, `random_access_iterator_tag`

forward_iterator_tag

Description

`Forward_iterator_tag` is an empty class: it has no member functions, member variables, or nested types. It is used solely as a "tag": a representation of the Forward Iterator concept within the C++ type system. Specifically, it is used as a return value for the function `iterator_category`. `Iterator_category` takes a single argument, an iterator, and returns an object whose type depends on the iterator's category. `Iterator_category`'s return value is of type `forward_iterator_tag` if its argument is a Forward Iterator.

Example

See `iterator_category`

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

None.

Model of

Assignable

Type requirements

None.

Public base classes

None.

Members

None.

New Members

None.

Notes

See also

`iterator_category`, Iterator Tags, `iterator_traits`, `output_iterator_tag`, `input_iterator_tag`, `bidirectional_iterator_tag`, `random_access_iterator_tag`

bidirectional_iterator_tag

Description

`Bidirectional_iterator_tag` is an empty class: it has no member functions, member variables, or nested types. It is used solely as a "tag": a representation of the Bidirectional Iterator concept within the C++ type system. Specifically, it is used as a return value for the function `iterator_category`. `Iterator_category` takes a single argument, an iterator, and returns an object whose type depends on the iterator's category. `Iterator_category`'s return value is of type `bidirectional_iterator_tag` if its argument is a Bidirectional Iterator.

Example

See `iterator_category`

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

None.

Model of

Assignable

Type requirements

None.

Public base classes

None.

Members

None.

New Members

None.

Notes

See also

`iterator_category`, Iterator Tags, `iterator_traits`, `output_iterator_tag`, `input_iterator_tag`, `forward_iterator_tag` `random_access_iterator_tag`

random_access_iterator_tag

Description

Random_access_iterator_tag is an empty class: it has no member functions, member variables, or nested types. It is used solely as a "tag": a representation of the Random Access Iterator concept within the C++ type system. Specifically, it is used as a return value for the function iterator_category. Iterator_category takes a single argument, an iterator, and returns an object whose type depends on the iterator's category. Iterator_category's return value is of type random_access_iterator_tag if its argument is a Random Access Iterator.

Example

See iterator_category

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

None.

Model of

Assignable

Type requirements

None.

Public base classes

None.

Members

None.

New Members

None.

Notes

See also

iterator_category, Iterator Tags, iterator_traits, output_iterator_tag, input_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag

8.4 Iterator functions

8.4.1 distance

Prototype

Distance is an overloaded name; there are actually two distance functions.

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);

template <class InputIterator, class Distance>
void distance(InputIterator first, InputIterator last, Distance& n);
```

Description

Finds the distance between first and last, *i.e.* the number of times that first must be incremented until it is equal to last. The first version of distance, which takes two arguments, simply returns that distance; the second version, which takes three arguments and which has a return type of void, increments n by that distance. The second version of distance was the one defined in the original STL, and the first version is the one defined in the draft C++ standard; the definition was changed because the older interface was clumsy and error-prone. The older interface required the use of a temporary variable, and it has semantics that are somewhat nonintuitive: it increments n by the distance from first to last, rather than storing that distance in n. Both interfaces are currently supported , for reasons of backward compatibility, but eventually the older version will be removed.

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Requirements on types

For the first version:

- `InputIterator` is a model of Input Iterator.

For the second version:

- `InputIterator` is a model of Input Iterator.
- `Distance` is an integral type that is able to represent a distance between iterators of type `InputIterator`.

Preconditions

- `[first, last)` is a *valid range*, as defined in the

  Input Iterator requirements.

Complexity

Constant time if `InputIterator` is a model of random access iterator, otherwise linear time.

Example

```
int main() {
  list<int> L;
  L.push_back(0);
  L.push_back(1);

  assert(distance(L.begin(), L.end()) == L.size());
}
```

Notes

This is the reason that `distance` is not defined for output iterators: it is impossible to compare two output iterators for equality. Forgetting to initialize `n` to 0 is a common mistake. The new `distance` interface uses the `iterator_traits` class, which relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use the newer version of `distance`, or any other STL components that involve `iterator_traits`.

See also

`distance_type`, `advance`, Input iterator, Random access iterator, Iterator tags, `iterator_traits`, Iterator overview.

8.4.2 advance

Prototype

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

Description

`Advance(i, n)` increments the iterator `i` by the distance `n`. If `n > 0` it is equivalent to executing `++i` `n` times, and if `n < 0` it is equivalent to executing `--i` `n` times. If `n == 0`, the call has no effect.

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `Distance` is an integral type that is convertible to `InputIterator`'s distance type.

Preconditions

- `i` is nonsingular.
- Every iterator between `i` and `i+n` (inclusive) is nonsingular.
- If `InputIterator` is a model of input iterator or forward iterator, then `n` must be nonnegative. If `InputIterator` is a model of

  bidirectional iterator or random access iterator, then this precondition does not apply.

Complexity

Constant time if `InputIterator` is a model of random access iterator, otherwise linear time.

Example

```
list<int> L;
L.push_back(0);
L.push_back(1);

list<int>::iterator i = L.begin();
advance(i, 2);
assert(i == L.end());
```

Notes

See also

`distance`, Input iterator, Bidirectional Iterator, Random access iterator, `iterator_traits`, Iterator overview.

8.5 Iterator classes

8.5.1 istream_iterator

Description

An `istream_iterator` is an Input Iterator that performs formatted input of objects of type `T` from a particular `istream`. When end of stream is reached, the `istream_iterator` takes on a special *end of stream* value, which is a past-the-end iterator. Note that all of the restrictions of an Input Iterator must be obeyed, including the restrictions on the ordering of `operator*` and `operator++` operations.

Example

Fill a `vector` with values read from standard input.

```
vector<int> V;
copy(istream_iterator<int>(cin), istream_iterator<int>(),
     back_inserter(V));
```

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`T`	The `istream_iterator`'s value type. `Operator*` returns a `const T&`.	
`Distance`	The `istream_iterator`'s distance type.	`ptrdiff_t`

Model of

Input Iterator

Type requirements

The value type `T` must be a type such that `cin >> T` is a valid expression. The value type `T` must be a model of Default Constructible. The distance type must, as described in the Input Iterator requirements, be a signed integral type.

Public base classes

None.

Members

Member	Where defined	Description
istream_iterator()	istream_iterator	See below.
istream_iterator(istream&)	istream_iterator	See below.
istream_iterator(const istream_iterator&)	Trivial Iterator	The copy constructor
istream_iterator& operator=(const istream_iterator&)	Trivial Iterator	The assignment operator
const T& operator*() const	Input Iterator	Returns the next object in the stream.
istream_iterator& operator++()	Input Iterator	Preincrement.
istream_iterator& operator++(int)	Input Iterator	Postincrement.
bool operator==(const istream_iterator&, const istream_iterator&)	Trivial iterator	The equality operator. This is a global function, not a member function.
input_iterator_tag iterator_category(const istream_iterator&)	iterator tags	Returns the iterator's category.
T* value_type(const istream_iterator&)	iterator tags	Returns the iterator's value type.
Distance* distance_type(const istream_iterator&)	iterator tags	Returns the iterator's distance type. ¡

New members

These members are not defined in the Input Iterator requirements, but are specific to `istream_iterator`.

Function	Description
istream_iterator()	The default constructor: Constructs an end-of-stream iterator. This is a past-the-end iterator, and it is useful when constructing a "range".
istream_iterator(istream& s)	Creates an `istream_iterator` that reads values from the input stream `s`. When `s` reaches end of stream, this iterator will compare equal to an end-of-stream iterator created using the default constructor.

Notes

See also

ostream_iterator, Input Iterator, Output Iterator.

8.5.2 ostream_iterator

Description

An `ostream_iterator` is an Output Iterator that performs formatted output of objects of type `T` to a particular `ostream`. Note that all of the restrictions of an Output Iterator must be obeyed, including the restrictions on the ordering of `operator*` and `operator++` operations.

Example

Copy the elements of a `vector` to the standard output, one per line.

```
vector<int> V;
// ...
copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
```

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`T`	The type of object that will be written to the `ostream`. The set of value types of an `ostream_iterator` consists of a single type, `T`.	

Model of

Output Iterator.

Type requirements

`T` must be a type such that `cout << T` is a valid expression.

Public base classes

None.

Members

Member	Where defined	Description
`ostream_iterator(ostream&)`	`ostream_iterator`	See below.
`ostream_iterator(ostream&, const char* s)`	`ostream_iterator`	See below.
`ostream_iterator(const ostream_iterator&)`	Output Iterator	The copy constructor
`ostream_iterator& operator=(const ostream_iterator&)`	Output Iterator	The assignment operator
`ostream_iterator& operator=(const T&)`	Output Iterator	Used to implement the Output Iterator requirement `*i = t`.
`ostream_iterator& operator*()`	Output Iterator	Used to implement the Output Iterator requirement `*i = t`.
`ostream_iterator& operator++()`	Output Iterator	Preincrement
`ostream_iterator& operator++(int)`	Output Iterator	Postincrement
`output_iterator_tag iterator_category(const ostream_iterator&)`	iterator tags	Returns the iterator's category.

New members

These members are not defined in the Output Iterator requirements, but are specific to `ostream_iterator`.

Function	Description
`ostream_iterator(ostream& s)`	Creates an `ostream_iterator` such that assignment of `t` through it is equivalent to `s << t`.
`ostream_iterator(ostream& s, const char* delim)`	Creates an `ostream_iterator` such that assignment of `t` through it is equivalent to `s << t << delim`.

See also

istream_iterator, Output Iterator, Input Iterator.

8.5.3 front_insert_iterator

Description

`Front_insert_iterator` is an iterator adaptor that functions as an Output Iterator: assignment through a `front_insert_iterator` inserts an object before the first element of a Front Insertion Sequence.

Example

```
list<int> L;
L.push_front(3);
front_insert_iterator<list<int> > ii(L);
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 2 1 0 3
```

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`FrontInsertionSequence`	The type of Front Insertion Sequence into which values will be inserted.	

Model of

Output Iterator. A front insert iterator's set of value types (as defined in the Output Iterator requirements) consists of a single type: `FrontInsertionSequence::value_type`.

Type requirements

The template parameter `FrontInsertionSequence` must be a Front Insertion Sequence.

Public base classes

None.

Members

Member	Where defined	Description
front_insert_iterator (FrontInsertionSequence&)	front_insert_iterator	See below.
front_insert_iterator (const front_insert_iterator&)	Trivial Iterator	The copy constructor
front_insert_iterator operator=(const front_insert_iterator&)	Trivial Iterator	The assignment operator
front_insert_iterator& operator*()	Output Iterator	Used to implement the output iterator expression *i = x.
front_insert_iterator operator=(const FrontInsertionSequence:: value_type&)	Output Iterator	Used to implement the output iterator expression *i = x.
front_insert_iterator& operator++()	Output Iterator	Preincrement.
front_insert_iterator& operator++(int)	Output Iterator	Postincrement.
output_iterator_tag iterator_category(const front_insert_iterator&)	iterator tags	Returns the iterator's category. This is a global function, not a member.
template<class FrontInsertionSequence> front_insert_iterator <FrontInsertionSequence> front_inserter (FrontInsertionSequence& S)	front_insert_iterator	See below.

New members

These members are not defined in the Output Iterator requirements, but are specific to front_insert_iterator.

Member	Description
front_insert_iterator (FrontInsertionSequence& S)	Constructs a front_insert_iterator that inserts objects before the first element of S.
template<class FrontInsertionSequence> front_insert_iterator <FrontInsertionSequence> front_inserter (FrontInsertionSequence& S);	Equivalent to front_insert_iterator <FrontInsertionSequence>(S). This is a global function, not a member function.

Notes

Note the difference between assignment through a `FrontInsertionSequence::iterator` and assignment through an `front_insert_iterator<FrontInsertionSequence>`. If `i` is a valid `FrontInsertionSequence::iterator`, then it points to some particular element in the front insertion sequence; the expression `*i = t` replaces that element with `t`, and does not change the total number of elements in the sequence. If `ii` is a valid `front_insert_iterator<FrontInsertionSequence>`, however, then the expression `*ii = t` is equivalent, for some FrontInsertionSequence `seq`, to the expression `seq.push_front(t)`. That is, it does not overwrite any of `seq`'s elements and it does change `seq`'s size. Note the difference between a `front_insert_iterator` and an `insert_iterator`. It may seem that a `front_insert_iterator` is the same as an `insert_iterator` constructed with an insertion point that is the beginning of a sequence. In fact, though, there is a very important difference: **every** assignment through a `front_insert_iterator` corresponds to an insertion before the first element of the sequence. If you are inserting elements at the beginning of a sequence using an `insert_iterator`, then the elements will appear in the order in which they were inserted. If, however, you are inserting elements at the beginning of a sequence using a `front_insert_iterator`, then the elements will appear in the reverse of the order in which they were inserted. This function exists solely for the sake of convenience: since it is a non-member function, the template parameters may be inferred and the type of the `front_insert_iterator` need not be declared explicitly. One easy way to reverse a range and insert it at the beginning of a Front Insertion Sequence `S`, for example, is `copy(first, last, front_inserter(S))`.

See also

insert_iterator, back_insert_iterator, Output Iterator, Sequence, Front Insertion Sequence, Iterator overview

8.5.4 back_insert_iterator

Description

`Back_insert_iterator` is an iterator adaptor that functions as an Output Iterator: assignment through a `back_insert_iterator` inserts an object after the last element of a Back Insertion Sequence.

Example

```
list<int> L;
L.push_front(3);
back_insert_iterator<list<int> > ii(L);
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 3 0 1 2
```

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`BackInsertionSequence`	The type of Back Insertion Sequence into which values will be inserted.	

Model of

Output Iterator. An insert iterator's set of value types (as defined in the Output Iterator requirements) consists of a single type: `BackInsertionSequence::value_type`.

Type requirements

The template parameter `BackInsertionSequence` must be a Back Insertion Sequence.

Public base classes

None.

Members

Member	Where defined	Description
back_insert_iterator (BackInsertionSequence&)	back_insert_iterator	See below.
back_insert_iterator (const back_insert_iterator&)	Trivial Iterator	The copy constructor
back_insert_iterator operator=(const back_insert_iterator&)	Trivial Iterator	The assignment operator
back_insert_iterator& operator*()	Output Iterator	Used to implement the output iterator expression *i = x.
back_insert_iterator operator=(const BackInsertionSequence:: value_type&)	Output Iterator	Used to implement the output iterator expression *i = x.
back_insert_iterator& operator++()	Output Iterator	Preincrement.
back_insert_iterator& operator++(int)	Output Iterator	Postincrement.
output_iterator_tag iterator_category(const back_insert_iterator&)	iterator tags	Returns the iterator's category. This is a global function, not a member.
template<class BackInsertionSequence> back_insert_iterator <BackInsertionSequence> back_inserter (BackInsertionSequence& S)	back_insert_iterator	See below.

New members

These members are not defined in the Output Iterator requirements, but are specific to back_insert_iterator.

Member function	Description
back_insert_iterator (BackInsertionSequence& S)	Constructs a back_insert_iterator that inserts objects after the last element of S. (That is, it inserts objects just before S's past-the-end iterator.)
template<class BackInsertionSequence> back_insert_iterator <BackInsertionSequence> back_inserter (BackInsertionSequence& S);	Equivalent to back_insert_iterator <BackInsertionSequence>(S). This is a global function, not a member function.

Notes

Note the difference between assignment through a BackInsertionSequence::iterator and assignment through a `back_insert_iterator<BackInsertionSequence>`. If `i` is a valid `BackInsertionSequence::iterator`, then it points to some particular element in the back insertion sequence; the expression `*i = t` replaces that element with `t`, and does not change the total number of elements in the back insertion sequence. If `ii` is a valid `back_insert_iterator<BackInsertionSequence>`, however, then the expression `*ii = t` is equivalent, to the expression `seq.push_back(t)`. That is, it does not overwrite any of `seq`'s elements and it does change `seq`'s size. This function exists solely for the sake of convenience: since it is a non-member function, the template parameters may be inferred and the type of the `back_insert_iterator` need not be declared explicitly. One easy way to reverse a range and insert it at the end of a Back Insertion Sequence `S`, for example, is `reverse_copy(first, last, back_inserter(S))`.

See also

insert_iterator, front_insert_iterator, Output Iterator, Back Insertion Sequence, Sequence, Iterator overview

8.5.5 insert_iterator

Description

`Insert_iterator` is an iterator adaptor that functions as an Output Iterator: assignment through an `insert_iterator` inserts an object into a Container. Specifically, if `ii` is an `insert_iterator`, then `ii` keeps track of a Container `c` and an insertion point `p`; the expression `*ii = x` performs the insertion `c.insert(p, x)`. There are two different Container concepts that define this expression: Sequence, and Sorted Associative Container. Both concepts define insertion into a container by means of `c.insert(p, x)`, but the semantics of this expression is very different in the two cases. For a Sequence `S`, the expression `S.insert(p, x)` means to insert the value `x` *immediately before* the iterator `p`. That is, the two-argument version of `insert` allows you to control the location at which the new element will be inserted. For a Sorted Associative Container, however, no such control is possible: the elements in a Sorted Associative Container always appear in ascending order of keys. Sorted Associative Containers define the two-argument version of `insert` as an optimization. The first argument is only a hint: it points to the location where the search will begin. If you assign through an `insert_iterator` several times, then you will be inserting several elements into the underlying container. In the case of a Sequence, they will appear at a particular location in the underlying sequence, in the order in which they were inserted: one of the arguments to `insert_iterator`'s constructor is an iterator `p`, and the new range will be inserted immediately before `p`. In the case of a Sorted Associative Container, however, the iterator in the `insert_iterator`'s constructor is almost irrelevant. The new elements will not necessarily form a contiguous range; they will appear in the appropriate location in the container, in ascending order by key. The order in which they are inserted only affects efficiency: inserting an already-sorted range into a Sorted Associative Container is an $O(N)$ operation.

Example

Insert a range of elements into a `list`.

```
list<int> L;
L.push_front(3);
insert_iterator<list<int> > ii(L, L.begin());
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 0 1 2 3.
```

Merge two sorted lists, inserting the resulting range into a `set`. Note that a `set` never contains duplicate elements.

```
int main()
{
  const int N = 6;

  int A1[N] = {1, 3, 5, 7, 9, 11};
  int A2[N] = {1, 2, 3, 4, 5, 6};
  set<int> result;

  merge(A1, A1 + N, A2, A2 + N,
        inserter(result, result.begin()));

  copy(result.begin(), result.end(), ostream_iterator<int>(cout, " "));
  cout << endl;

  // The output is "1 2 3 4 5 6 7 9 11".
}
```

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
`Container`	The type of Container into which values will be inserted.	

Model of

Output Iterator. An insert iterator's set of value types (as defined in the Output Iterator requirements) consists of a single type: `Container::value_type`.

Type requirements

- The template parameter `Container` is a model of Container.
- `Container` is variable-sized, as described in the

  Container requirements.
- `Container` has a two-argument `insert` member function. Specifically, if `c` is an object of type `Container`, `p` is an object of type `Container::iterator` and `v` is an object of type `Container::value_type`, then `c.insert(p, v)` must be a valid expression.

Public base classes

None.

Members

Member	Where defined	Description
`insert_iterator(Container&, Container::iterator)`	`insert_iterator`	See below.
`insert_iterator(const insert_iterator&)`	Trivial Iterator	The copy constructor
`insert_iterator& operator=(const insert_iterator&)`	Trivial Iterator	The assignment operator
`insert_iterator& operator*()`	Output Iterator	Used to implement the output iterator expression `*i = x`.
`insert_iterator& operator=(const Container::value_type&)`	Output Iterator	Used to implement the output iterator expression `*i = x`.
`insert_iterator& operator++()`	Output Iterator	Preincrement.
`insert_iterator& operator++(int)`	Output Iterator	Postincrement.
`output_iterator_tag iterator_category(const insert_iterator&)`	iterator tags	Returns the iterator's category. This is a global function, not a member.
`template<class Container, class Iter) insert_iterator<Container> inserter(Container& C, Iter i);`	`insert_iterator`	See below.

New members

These members are not defined in the Output Iterator requirements, but are specific to `insert_iterator`.

Member	Description
`insert_iterator(Container& C, Container::iterator i)`	Constructs an `insert_iterator` that inserts objects in `C`. If `Container` is a Sequence, then each object will be inserted immediately before the element pointed to by `i`. If `C` is a Sorted Associative Container, then the first insertion will use `i` as a hint for beginning the search. The iterator `i` must be a dereferenceable or past-the-end iterator in `C`.
`template<class Container, class Iter) insert_iterator<Container> inserter(Container& C, Iter i);`	Equivalent to `insert_iterator<Container>(C, i)`. This is a global function, not a member function.

Notes

Note the difference between assignment through a `Container::iterator` and assignment through an `insert_iterator<Container>`. If `i` is a valid `Sequence::iterator`, then it points to some particular element in the container; the expression `*i = t` replaces that element with `t`, and does not change the total number of elements in the container. If `ii` is a valid `insert_iterator<container>`, however, then the expression `*ii = t` is equivalent, for some `container c` and some valid `container::iterator j`, to the expression `c.insert(j, t)`. That is, it does not overwrite any of `c`'s elements and it does change `c`'s size. This function exists solely for the sake of convenience: since it is a non-member function, the template parameters may be inferred and the type of the `insert_iterator` need not be declared explicitly. One easy way to reverse a range and insert it into a Sequence `S`, for example, is `reverse_copy(first, last, inserter(S, S.begin()))`.

See also

front_insert_iterator, back_insert_iterator, Output Iterator, Sequence, Iterator overview

8.5.6 reverse_iterator

Description

`Reverse_iterator` is an iterator adaptor that enables backwards traversal of a range. `Operator++` applied to an object of class `reverse_iterator<RandomAccessIterator>` means the same thing as `operator--`

applied to an object of class `RandomAccessIterator`. There are two different reverse iterator adaptors: the class `reverse_iterator` has a template argument that is a Random Access Iterator, and the class `reverse_bidirectional_iterator` has a template argument that is a Bidirectional Iterator.

Example

```
template <class T>
void forw(const vector<T>& V)
{
   vector<T>::iterator first = V.begin();
   vector<T>::iterator last = V.end();
   while (first != last)
      cout << *first++ << endl;
}

template <class T>
void rev(const vector<T>& V)
{
   typedef reverse_iterator<vector<T>::iterator,
                            T,
                            vector<T>::reference_type,
                            vector<T>::difference_type>
           reverse_iterator;
   reverse_iterator rfirst(V.end());
   reverse_iterator rlast(V.begin());

   while (rfirst != rlast)
      cout << *rfirst++ << endl;
}
```

In the function `forw`, the elements are printed in the order `*first`, `*(first+1)`, ..., `*(last-1)`. In the function `rev`, they are printed in the order `*(last - 1)`, `*(last-2)`, ..., `*first`.

Definition

Defined in the standard header iterator, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
RandomAccessIterator	The base iterator class. Incrementing an object of class reverse_iterator<Iterator> corresponds to decrementing an object of class Iterator.	
T	The reverse iterator's value type. This should always be the same as the base iterator's value type.	
Reference	The reverse iterator's reference type. This should always be the same as the base iterator's reference type.	T&
Distance	The reverse iterator's distance type. This should always be the same as the base iterator's distance type.	ptrdiff_t

Model of

Random Access Iterator

Type requirements

The base iterator type (that is, the template parameter RandomAccessIterator) must be a Random Access Iterator. The reverse_iterator's value type, reference type, and distance type (that is, the template parameters T, Reference, and Distance, respectively) must be the same as the base iterator's value type, reference type, and distance type.

Public base classes

None.

Members

Member	Where defined	Description
self	reverse_iterator	See below
reverse_iterator()	Trivial Iterator	The default constructor
reverse_iterator(const reverse_iterator& x)	Trivial Iterator	The copy constructor
reverse_iterator& operator=(const reverse_iterator& x)	Trivial Iterator	The assignment operator
reverse_iterator (RandomAccessIterator x)	reverse_iterator	See below.

Member	Where defined	Description
`RandomAccessIterator base()`	`reverse_iterator`	See below.
`Reference operator*() const`	Trivial Iterator	The dereference operator
`reverse_iterator& operator++()`	Forward Iterator	Preincrement
`reverse_iterator operator++(int)`	Forward Iterator	Postincrement
`reverse_iterator& operator--()`	Bidirectional Iterator	Predecrement
`reverse_iterator operator--(int)`	Bidirectional Iterator	Postdecrement
`reverse_iterator operator+(Distance)`	Random Access Iterator	Iterator addition
`reverse_iterator& operator+=(Distance)`	Random Access Iterator	Iterator addition
`reverse_iterator operator-(Distance)`	Random Access Iterator	Iterator subtraction
`reverse_iterator& operator-=(Distance)`	Random Access Iterator	Iterator subtraction
`Reference operator[](Distance)`	Random Access Iterator	Random access to an element.
`reverse_iterator operator+(Distance, reverse_iterator)`	Random Access Iterator	Iterator addition. This is a global function, not a member function.
`Distance operator-(const reverse_iterator&, const reverse_iterator&)`	Random Access Iterator	Finds the distance between two iterators. This is a global function, not a member function.
`bool operator==(const reverse_iterator&, const reverse_iterator&)`	Trivial Iterator	Compares two iterators for equality. This is a global function, not a member function.
`bool operator<(const reverse_iterator&, const reverse_iterator&)`	Random Access Iterator	Determines whether the first argument precedes the second. This is a global function, not a member function.
`random_access_iterator_tag iterator_category(const reverse_iterator&)`	Iterator tags	Returns the iterator's category. This is a global function, not a member function.
`T* value_type(const reverse_iterator&)`	Iterator tags	Returns the iterator's value type. This is a global function, not a member function.
`Distance* distance_type(const reverse_iterator&)`	Iterator tags	Returns the iterator's distance type. This is a global function, not a member function.

New members

These members are not defined in the Random Access Iterator requirements, but are specific to `reverse_iterator`.

Member	Description
`self`	A typedef for `reverse_iterator<RandomAccessIterator, T, Reference, Distance>`.
`RandomAccessIterator base()`	Returns the current value of the `reverse_iterator`'s base iterator. If `ri` is a reverse iterator and `i` is any iterator, the two fundamental identities of reverse iterators can be written as `reverse_iterator(i).base() == i` and `&*ri == &*(ri.base() - 1)`.
`reverse_iterator (RandomAccessIterator i)`	Constructs a `reverse_iterator` whose base iterator is `i`.

Notes

There isn't really any good reason to have two separate classes: this separation is purely because of a technical limitation in some of today's C++ compilers. If the two classes were combined into one, then there would be no way to declare the return types of the iterator tag functions `iterator_category`, `distance_type` and `value_type` correctly. The *iterator traits* class solves this problem: it addresses the same issues as the iterator tag functions, but in a cleaner and more flexible manner. Iterator traits, however, rely on *partial specialization*, and many C++ compilers do not yet implement partial specialization. Once compilers that support partial specialization become more common, these two different reverse iterator classes will be combined into a single class. The declarations for `rfirst` and `rlast` are written in this clumsy form simply as an illustration of how to declare a `reverse_iterator`. `Vector` is a Reversible Container, so it provides a typedef for the appropriate instantiation of `reverse_iterator`. The usual way of declaring these variables is much simpler:

```
vector<T>::reverse_iterator rfirst = rbegin();
vector<T>::reverse_iterator rlast = rend();
```

Note the implications of this remark. The variable `rfirst` is initialized as `reverse_iterator<...> rfirst(V.end());`. The value obtained when it is dereferenced, however, is `*(V.end() - 1)`. This is a general property: the fundamental identity of reverse iterators is `&*(reverse_iterator(i)) == &*(i - 1)`. This code sample shows why this identity is important: if `[f, l)` is a valid range, then it allows `[reverse_iterator(l), reverse_iterator(f))` to be a valid range as well. Note that the iterator `l` is not part of the range, but it is required to be dereferenceable or past-the-end. There is no requirement that any such iterator precedes `f`.

See also

Reversible Container, reverse_bidirectional_iterator, Random Access Iterator, iterator tags, Iterator Overview

8.5.7 raw_storage_iterator

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If `i` is an iterator that points to a region of uninitialized memory, then you can use `construct` to create an object in the location pointed to by `i`. `Raw_storage_iterator` is an adaptor that makes this procedure more convenient. If `r` is a `raw_storage_iterator`, then it has some underlying iterator `i`. The expression `*r = x` is equivalent to `construct(&*i, x)`.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  transform(A1, A1 + N,
            raw_storage_iterator<Int*, int>(A2),
            negate<int>());
}
```

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
OutputIterator	The type of the raw_storage_iterator's underlying iterator.	
T	The type that will be used as the argument to the constructor.	

Model of

Output Iterator

Type requirements

- ForwardIterator is a model of Forward Iterator
- ForwardIterator's value type has a constructor that takes a single argument of type T.

Public base classes

None.

Members

Member	Where defined	Description
raw_storage_iterator (ForwardIterator x)	raw_storage_iterator	See below.
raw_storage_iterator(const raw_storage_iterator&)	trivial iterator	The copy constructor
raw_storage_iterator& operator=(const raw_storage_iterator&)	trivial iterator	The assignment operator
raw_storage_iterator& operator*()	Output Iterator	Used to implement the output iterator expression *i = x.
raw_storage_iterator& operator=(const Sequence::value_type&)	Output Iterator	Used to implement the output iterator expression *i = x.
raw_storage_iterator& operator++()	Output Iterator	Preincrement.
raw_storage_iterator& operator++(int)	Output Iterator	Postincrement.
output_iterator_tag iterator_category(const raw_storage_iterator&)	iterator tags	Returns the iterator's category. This is a global function, not a member.

New members

These members are not defined in the Output Iterator requirements, but are specific to `raw_storage_iterator`.

Function	Description
`raw_storage_iterator(ForwardIterator i)`	Creates a `raw_storage_iterator` whose underlying iterator is `i`.
`raw_storage_iterator& operator=(const T& val)`	Constructs an object of `ForwardIterator`'s value type at the location pointed to by the iterator, using `val` as the constructor's argument.

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes.

See also

Allocators, `construct`, `destroy`, `uninitialized_copy` `uninitialized_fill`, `uninitialized_fill_n`,

Chapter 9

Algorithms

9.1 Non-mutating algorithms

9.1.1 for_each

Prototype

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last,
                       UnaryFunction f);
```

Description

`For_each` applies the function object `f` to each element in the range `[first, last)`; `f`'s return value, if any, is ignored. Applications are performed in forward order, *i.e.* from `first` to `last`. `For_each` returns the function object after it has been applied to each element.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator
- `UnaryFunction` is a model of Unary Function

- `UnaryFunction` does not apply any non-constant operation through its argument.
- `InputIterator`'s value type is convertible to `UnaryFunction`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `last - first` applications of `UnaryFunction`.

Example

```
template<class T> struct print : public unary_function<T, void>
{
  print(ostream& out) : os(out), count(0) {}
  void operator() (T x) { os << x << ' '; ++count; }
  ostream& os;
  int count;
};

int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  print<int> P = for_each(A, A + N, print<int>(cout));
  cout << endl << P.count << " objects printed." << endl;
}
```

Notes

This return value is sometimes useful, since a function object may have local state. It might, for example, count the number of times that it is called, or it might have a status flag to indicate whether or not a call succeeded.

See also

The function object overview, `count`, `copy`

9.1.2 find

Prototype

```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

Description

Returns the first iterator `i` in the range `[first, last)` such that `*i == value`. Returns `last` if no such iterator exists.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- EqualityComparable is a model of EqualityComparable.
- InputIterator is a model of InputIterator.
- Equality is defined between objects of type EqualityComparable and objects of InputIterator's value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear: at most `last - first` comparisons for equality.

Example

```
list<int> L;
L.push_back(3);
L.push_back(1);
L.push_back(7);

list<int>::iterator result = find(L.begin(), L.end(), 7);
assert(result == L.end() || *result == 7);
```

Notes

See also

find_if.

9.1.3 find_if

Prototype

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

Description

Returns the first iterator `i` in the range `[first, last)` such that `pred(*i)` is true. Returns `last` if no such iterator exists.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- Predicate is a model of Predicate.
- InputIterator is a model of InputIterator.
- The value type of InputIterator is convertible to the argument type of Predicate.

Preconditions

- `[first, last)` is a valid range.
- For each iterator `i` in the range `[first, last)`, *i is in the domain of Predicate.

Complexity

Linear: at most `last - first` applications of `Pred`.

Example

```
list<int> L;
L.push_back(-3);
L.push_back(0);
L.push_back(3);
L.push_back(-2);

list<int>::iterator result = find_if(L.begin(), L.end(),
                                     bind2nd(greater<int>(), 0));
assert(result == L.end() || *result > 0);
```

Notes

See also

find.

9.1.4 adjacent_find

Prototype

`Adjacent_find` is an overloaded name; there are actually two `adjacent_find` functions.

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

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

Description

The first version of `adjacent_find` returns the first iterator `i` such that `i` and `i+1` are both valid iterators in `[first, last)`, and such that `*i == *(i+1)`. It returns `last` if no such iterator exists. The second version of `adjacent_find` returns the first iterator `i` such that `i` and `i+1` are both valid iterators in `[first, last)`, and such that `binary_pred(*i, *(i+1))` is `true`. It returns `last` if no such iterator exists.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator`'s value type is Equality Comparable.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s first argument type and to its second argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. If `first == last` then no comparison are performed; otherwise, at most `(last - first) - 1` comparisons.

Example

Find the first element that is greater than its successor.

```
int A[] = {1, 2, 3, 4, 6, 5, 7, 8};
const int N = sizeof(A) / sizeof(int);

const int* p = adjacent_find(A, A + N, greater<int>());

cout << "Element " << p - A << " is out of order: "
     << *p << " > " << *(p + 1) << "." << endl;
```

Notes

See also

find, mismatch, equal, search

9.1.5 find_first_of

Prototype

find_first_of is an overloaded name; there are actually two find_first_of functions.

```
template <class InputIterator, class ForwardIterator>
InputIterator find_first_of(InputIterator first1,
                            InputIterator last1,
                            ForwardIterator first2,
                            ForwardIterator last2);

template <class InputIterator, class ForwardIterator,
          class BinaryPredicate>
InputIterator find_first_of(InputIterator first1,
                            InputIterator last1,
                            ForwardIterator first2,
                            ForwardIterator last2,
                            BinaryPredicate comp);
```

Description

Find_first_of is similar to find, in that it performs linear seach through a range of Input Iterators. The difference is that while find searches for one particular value, find_first_of searches for any of several values. Specifically, find_first_of searches for the first occurrance in the range [first1, last1) of any of the elements in [first2, last2). (Note that this behavior is reminiscent of the function strpbrk from the standard C library.) The two versions of find_first_of differ in how they compare elements for equality. The first uses operator==, and the second uses and arbitrary user-supplied function object comp. The first version returns the first iterator i in [first1, last1) such that, for some iterator j in [first2, last2), *i == *j. The second returns the first iterator i in [first1, last1) such that, for some iterator j in [first2, last2), comp(*i, *j) is true. As usual, both versions return last1 if no such iterator i exists.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator` is a model of Input Iterator.
- `ForwardIterator` is a model of Forward Iterator.
- `InputIterator`'s value type is EqualityComparable, and can be compared for equality with ForwardIterator's value type.

For the second version:

- `InputIterator` is a model of Input Iterator.
- `ForwardIterator` is a model of Forward Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `InputIterator`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

Complexity

At most `(last1 - first1) * (last2 - first2)` comparisons.

Example

Like `strpbrk`, one use for `find_first_of` is finding whitespace in a string; space, tab, and newline are all whitespace characters.

```
int main()
{
  const char* WS = "\t\n ";
  const int n_WS = strlen(WS);

  char* s1 = "This sentence contains five words.";
  char* s2 = "OneWord";


  char* end1 = find_first_of(s1, s1 + strlen(s1),
                             WS, WS + n_WS);
  char* end2 = find_first_of(s2, s2 + strlen(s2),
                             WS, WS + n_WS);

  printf("First word of s1: \%.*s\n", end1 - s1, s1);
  printf("First word of s2: \%.*s\n", end2 - s2, s2);
}
```

Notes

See also

`find`, `find_if`, `search`

9.1.6 count

Prototype

`Count` is an overloaded name: there are two `count` functions.

```
template <class InputIterator, class EqualityComparable>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const EqualityComparable& value);

template <class InputIterator, class EqualityComparable, class Size>
void count(InputIterator first, InputIterator last,
           const EqualityComparable& value,
           Size& n);
```

Description

`Count` finds the number of elements in `[first, last)` that are equal to `value`. More precisely, the first version of `count` returns the number of iterators `i` in `[first,` last) such that *i == value. The second version of count adds to n the number of iterators i in [first, last) such that *i == value. The second version of count was the one defined in the original STL, and the first version is the one defined in the draft C++ standard; the definition was changed because the older interface was clumsy and error-prone. The older interface required the use of a temporary variable, which had to be initialized to 0 before the call to count. Both interfaces are currently supported , for reasons of backward compatibility, but eventually the older version will be removed.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, which takes three arguments:

- InputIterator is a model of Input Iterator.
- EqualityComparable is a model of Equality Comparable.
- InputIterator's value type is a model of Equality Comparable.
- An object of InputIterator's value type can be compared for equality with an object of type EqualityComparable.

For the second version, which takes four arguments:

- InputIterator is a model of Input Iterator.
- EqualityComparable is a model of Equality Comparable.
- Size is an integral type that can hold values of InputIterator's distance type.
- InputIterator's value type is a model of Equality Comparable.
- An object of InputIterator's value type can be compared for equality with an object of type EqualityComparable.

Preconditions

- [first, last) is a valid range.

For the second version:

- `[first, last)` is a valid range.
- `n` plus the number of elements equal to `value` does not exceed the maximum value of type `Size`.

Complexity

Linear. Exactly `last - first` comparisons.

Example

```
int main() {
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  cout << "Number of zeros: "
       << count(A, A + N, 0)
       << endl;
}
```

Notes

The new `count` interface uses the `iterator_traits` class, which relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use the newer version of `count`, or any other STL components that involve `iterator_traits`.

See also

`count_if`, `find`, `find_if`

9.1.7 count_if

Prototype

`Count_if` is an overloaded name: there are two `count_if` functions.

```
template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last,
              Predicate pred,
              Size& n);
```

Description

`Count_if` finds the number of elements in `[first, last)` that satisfy the predicate `pred`. More precisely, the first version of `count_if` returns the number of iterators `i` in `[first, last)` such that `pred(*i)` is `true`. The second version of `count` adds to `n` the number of iterators `i` in `[first, last)` such that `pred(*i)` is `true`. The second version of `count_if` was the one defined in the original STL, and the first version is the one defined in the draft C++ standard; the definition was changed because the older interface was clumsy and error-prone. The older interface required the use of a temporary variable, which had to be initialized to 0 before the call to `count_if`. Both interfaces are currently supported , for reasons of backward compatibility, but eventually the older version will be removed.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, which takes three arguments:

- `InputIterator` is a model of Input Iterator.
- `Predicate` is a model of Predicate.
- `InputIterator`'s value type is convertible to `Predicate`'s argument type.

For the second version, which takes four arguments:

- `InputIterator` is a model of Input Iterator.
- `Predicate` is a model of Predicate.
- `Size` is an integral type that can hold values of `InputIterator`'s distance type.
- `InputIterator`'s value type is convertible to `Predicate`'s argument type.

Preconditions

For the first version:

- [first, last) is a valid range.

For the second version:

- [first, last) is a valid range.
- n plus the number of elements that satisfy pred does not exceed the maximum value of type Size.

Complexity

Linear. Exactly last - first applications of pred.

Example

```
int main() {
  int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
  const int N = sizeof(A) / sizeof(int);

  cout << "Number of even elements: "
       << count_if(A, A + N,
                   compose1(bind2nd(equal_to<int>(), 0),
                            bind2nd(modulus<int>(), 2)))
       << endl;
}
```

Notes

The new count interface uses the iterator_traits class, which relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use the newer version of count, or any other STL components that involve iterator_traits.

See also

count, find, find_if

9.1.8 mismatch

Prototype

`Mismatch` is an overloaded name; there are actually two `mismatch` functions.

```
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(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2,
         BinaryPredicate binary_pred);
```

Description

`Mismatch` finds the first position where the two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` differ. The two versions of `mismatch` use different tests for whether elements differ. The first version of `mismatch` finds the first iterator `i` in `[first1, last1)` such that `*i != *(first2 + (i - first1))`. The return value is a pair whose first element is `i` and whose second element is `*(first2 + (i - first1))`. If no such iterator `i` exists, the return value is a pair whose first element is `last1` and whose second element is `*(first2 + (last1 - first1))`. The second version of `mismatch` finds the first iterator `i` in `[first1, last1)` such that `binary_pred(*i, *(first2 + (i - first1))` is `false`. The return value is a pair whose first element is `i` and whose second element is `*(first2 + (i - first1))`. If no such iterator `i` exists, the return value is a pair whose first element is `last1` and whose second element is `*(first2 + (last1 - first1))`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.

- `InputIterator1`'s value type is a model of Equality Comparable.
- `InputIterator2`'s value type is a model of Equality Comparable.
- `InputIterator1`'s value type can be compared for equality with `InputIterator2`'s value type.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `InputIterator1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `InputIterator2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, first2 + (last2 - last1))` is a valid range.

Complexity

Linear. At most `last1 - first1` comparisons.

Example

```
int A1[] = { 3, 1, 4, 1, 5, 9, 3 };
int A2[] = { 3, 1, 4, 2, 8, 5, 7 };
const int N = sizeof(A1) / sizeof(int);

pair<int*, int*> result = mismatch(A1, A1 + N, A2);
cout << "The first mismatch is in position " << result.first - A1
     << endl;
cout << "Values are: " << *(result.first) << ", " << *(result.second)
     << endl;
```

Notes

See also

`equal`, `search`, `find`, `find_if`

9.1.9 equal

Prototype

`Equal` is an overloaded name; there are actually two `equal` functions.

```
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

`Equal` returns `true` if the two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` are identical when compared element-by-element, and otherwise returns `false`. The first version of `equal` returns `true` if and only if for every iterator `i` in `[first1, last1)`, `*i == *(first2 + (i - first1))`. The second version of `equal` returns `true` if and only if for every iterator `i` in `[first1, last1)`, `binary_pred(*i, *(first2 + (i - first1))` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `InputIterator1`'s value type is a model of Equality Comparable.
- `InputIterator2`'s value type is a model of Equality Comparable.

- `InputIterator1`'s value type can be compared for equality with `InputIterator2`'s value type.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `InputIterator1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `InputIterator2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, first2 + (last2 - last1))` is a valid range.

Complexity

Linear. At most `last1 - first1` comparisons.

Example

```
int A1[] = { 3, 1, 4, 1, 5, 9, 3 };
int A2[] = { 3, 1, 4, 2, 8, 5, 7 };
const int N = sizeof(A1) / sizeof(int);

cout << "Result of comparison: " << equal(A1, A1 + N, A2) << endl;
```

Notes

Note that this is very similar to the behavior of `mismatch`: The only real difference is that while `equal` will simply return `false` if the two ranges differ, `mismatch` returns the first location where they do differ. The expression `equal(f1, l1, f2)` is precisely equivalent to the expression `mismatch(f1, l1, f2).first == l1`, and this is in fact how `equal` could be implemented.

See also

mismatch, search, find, find_if

9.1.10 search

Prototype

Search is an overloaded name; there are actually two search functions.

```
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

Search finds a subsequence within the range [first1, last1) that is identical to [first2, last2) when compared element-by-element. It returns an iterator pointing to the beginning of that subsequence, or else last1 if no such subsequence exists. The two versions of search differ in how they determine whether two elements are the same: the first uses operator==, and the second uses the user-supplied function object binary_pred. The first version of search returns the first iterator i in the range [first1, last1 - (last2 - first2)) such that, for every iterator j in the range [first2, last2), *(i + (j - first2)) == *j. The second version returns the first iterator i in [first1, last1 - (last2 - first2)) such that, for every iterator j in [first2, last2), binary_pred(*(i + (j - first2)), *j) is true. These conditions simply mean that every element in the subrange beginning with i must be the same as the corresponding element in [first2, last2).

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator1` is a model of Forward Iterator.
- `ForwardIterator2` is a model of Forward Iterator.
- `ForwardIterator1`'s value type is a model of EqualityComparable.
- `ForwardIterator2`'s value type is a model of EqualityComparable.
- Objects of `ForwardIterator1`'s value type can be compared for equality with Objects of `ForwardIterator2`'s value type.

For the second version:

- `ForwardIterator1` is a model of Forward Iterator.
- `ForwardIterator2` is a model of Forward Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardIterator2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

Complexity

Worst case behavior is quadratic: at most `(last1 - first1) * (last2 - first2)` comparisons. This worst case, however, is rare. Average complexity is linear.

Example

```
const char S1[] = "Hello, world!";
const char S2[] = "world";
const int N1 = sizeof(S1) - 1;
const int N2 = sizeof(S2) - 1;

const char* p = search(S1, S1 + N1, S2, S2 + N2);
printf("Found subsequence \"%s\" at character %d of sequence \"%s\".\n",
       S2, p - S1, S1);
```

Notes

The reason that this range is `[first1, last1 - (last2 - first2))`, instead of simply `[first1, last1)`, is that we are looking for a subsequence that is equal to the *complete* sequence `[first2, last2)`. An iterator `i` can't be the beginning of such a subsequence unless `last1 - i` is greater than or equal to `last2 - first2`. Note the implication of this: you may call `search` with arguments such that `last1 - first1` is less than `last2 - first2`, but such a search will always fail.

See also

`find`, `find_if`, `find_end`, `search_n`, `mismatch`, `equal`

9.1.11 search_n

Prototype

`Search_n` is an overloaded name; there are actually two `search_n` functions.

```
template <class ForwardIterator, class Integer, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Integer count, const T& value);

template <class ForwardIterator, class Integer,
          class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Integer count, const T& value,
                         BinaryPredicate binary_pred);
```

Description

`Search_n` searches for a subsequence of `count` consecutive elements in the range `[first, last)`, all of which are equal to `value`. It returns an iterator pointing to the beginning of that subsequence, or else `last` if no such subsequence exists. The two versions of `search_n` differ in how they determine whether two elements are the same: the first uses `operator==`, and the second uses the user-supplied function object `binary_pred`. The first version of `search` returns the first iterator `i` in the range `[first, last - count)` such that, for every iterator `j` in the range `[i, i + count)`, `*j == value`. The second version returns the first iterator `i` in the range `[first, last - count)` such that, for every iterator `j` in the range `[i, i + count)`, `binary_pred(*j, value)` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `Integer` is an integral type.
- `T` is a model of EqualityComparable.
- `ForwardIterator`'s value type is a model of EqualityComparable.
- Objects of `ForwardIterator`'s value type can be compared for equality with Objects of type `T`.

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `Integer` is an integral type.
- `T` is a model of EqualityComparable.
- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `T` is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first, last)` is a valid range.
- `count` is non-negative .

Complexity

Linear. `Search_n` performs at most `last - first` comparisons. (The C++ standard permits the complexity to be O(`n (last - first)`), but this is unnecessarily lax. There is no reason for `search_n` to examine any element more than once.)

Example

```
bool eq_nosign(int x, int y) { return abs(x) == abs(y); }

void lookup(int* first, int* last, size_t count, int val) {
  cout << "Searching for a sequence of "
       << count
       << " '" << val << "'"
       << (count != 1 ? "s: " : ":  ");
  int* result = search_n(first, last, count, val);
  if (result == last)
    cout << "Not found" << endl;
  else
    cout << "Index = " << result - first << endl;
}

void lookup_nosign(int* first, int* last, size_t count, int val) {
  cout << "Searching for a (sign-insensitive) sequence of "
       << count
       << " '" << val << "'"
       << (count != 1 ? "s: " : ":  ");
  int* result = search_n(first, last, count, val, eq_nosign);
  if (result == last)
    cout << "Not found" << endl;
  else
    cout << "Index = " << result - first << endl;
}

int main() {
  const int N = 10;
  int A[N] = {1, 2, 1, 1, 3, -3, 1, 1, 1, 1};

  lookup(A, A+N, 1, 4);
  lookup(A, A+N, 0, 4);
  lookup(A, A+N, 1, 1);
  lookup(A, A+N, 2, 1);
  lookup(A, A+N, 3, 1);
  lookup(A, A+N, 4, 1);

  lookup(A, A+N, 1, 3);
  lookup(A, A+N, 2, 3);
  lookup_nosign(A, A+N, 1, 3);
  lookup_nosign(A, A+N, 2, 3);
}
```

The output is

```
Searching for a sequence of 1 '4':  Not found
Searching for a sequence of 0 '4's: Index = 0
Searching for a sequence of 1 '1':  Index = 0
Searching for a sequence of 2 '1's: Index = 2
Searching for a sequence of 3 '1's: Index = 6
Searching for a sequence of 4 '1's: Index = 6
Searching for a sequence of 1 '3':  Index = 4
Searching for a sequence of 2 '3's: Not found
Searching for a (sign-insensitive) sequence of 1 '3':  Index = 4
Searching for a (sign-insensitive) sequence of 2 '3's: Index = 4
```

Notes

Note that `count` is permitted to be zero: a subsequence of zero elements is well defined. If you call `search_n` with `count` equal to zero, then the search will always succeed: no matter what `value` is, every range contains a subrange of zero consecutive elements that are equal to `value`. When `search_n` is called with `count` equal to zero, the return value is always `first`. The reason that this range is `[first, last - count)`, rather than just `[first, last)`, is that we are looking for a subsequence whose length is `count`; an iterator `i` can't be the beginning of such a subsequence unless `last - count` is greater than or equal to `count`. Note the implication of this: you may call `search_n` with arguments such that `last - first` is less than `count`, but such a search will always fail.

See also

`search`, `find_end`, `find`, `find_if`

9.1.12 find_end

Prototype

`find_end` is an overloaded name; there are actually two `find_end` functions.

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate comp);
```

Description

`Find_end` is misnamed: it is much more similar to `search` than to `find`, and a more accurate name would have been `search_end`. Like `search`, `find_end` attempts to find a subsequence within the range `[first1, last1)` that is identical to `[first2, last2)`. The difference is that while `search` finds the first such subsequence, `find_end` finds the last such subsequence. `Find_end` returns an iterator pointing to the beginning of that subsequence; if no such subsequence exists, it returns `last1`. The two versions of `find_end` differ in how they determine whether two elements are the same: the first uses `operator==`, and the second uses the user-supplied function object `comp`. The first version of `find_end` returns the last iterator `i` in the range `[first1, last1 - (last2 - first2))` such that, for every iterator `j` in the range `[first2, last2)`, `*(i + (j - first2)) == *j`. The second version of `find_end` returns the last iterator `i` in `[first1, last1 - (last2 - first2))` such that, for every iterator `j` in `[first2, last2)`, `binary_pred(*(i + (j - first2)), *j)` is `true`. These conditions simply mean that every element in the subrange beginning with `i` must be the same as the corresponding element in `[first2, last2)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator1` is a model of Forward Iterator.
- `ForwardIterator2` is a model of Forward Iterator.
- `ForwardIterator1`'s value type is a model of EqualityComparable.

- `ForwardIterator2`'s value type is a model of EqualityComparable.
- Objects of `ForwardIterator1`'s value type can be compared for equality with Objects of `ForwardIterator2`'s value type.

For the second version:

- `ForwardIterator1` is a model of Forward Iterator.
- `ForwardIterator2` is a model of Forward Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardIterator2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

Complexity

The number of comparisons is proportional to `(last1 - first1) * (last2 - first2)`. If both `ForwardIterator1` and `ForwardIterator2` are models of Bidirectional Iterator, then the average complexity is linear and the worst case is at most `(last1 - first1) * (last2 - first2)` comparisons.

Example

```
int main()
{
  char* s = "executable.exe";
  char* suffix = "exe";

  const int N = strlen(s);
  const int N_suf = strlen(suffix);

  char* location = find_end(s, s + N,
                            suffix, suffix + N_suf);

  if (location != s + N) {
    cout << "Found a match for " << suffix << " within " << s << endl;
    cout << s << endl;

    int i;
    for (i = 0; i < (location - s); ++i)
      cout << ' ';
    for (i = 0; i < N_suf; ++i)
      cout << '^';
    cout << endl;
  }
  else
    cout << "No match for " << suffix << " within " << s << endl;
}
```

Notes

The reason that this range is `[first1, last1 - (last2 - first2))`, instead of simply `[first1, last1)`, is that we are looking for a subsequence that is equal to the *complete* sequence `[first2, last2)`. An iterator `i` can't be the beginning of such a subsequence unless `last1 - i` is greater than or equal to `last2 - first2`. Note the implication of this: you may call `find_end` with arguments such that `last1 - first1` is less than `last2 - first2`, but such a search will always fail.

See also

`search`

9.2 Mutating algorithms

9.2.1 copy

Prototype

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Description

`Copy` copies elements from the range `[first, last)` to the range `[result, result + (last - first))`. That is, it performs the assignments `*result = *first`, `*(result + 1) = *(first + 1)`, and so on. Generally, for every integer `n` from `0` to `last - first`, `copy` performs the assignment `*(result + n) = *(first + n)`. Assignments are performed in forward order, *i.e.* in order of increasing `n`. The return value is `result + (last - first)`

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- InputIterator is a model of Input Iterator.
- OutputIterator is a model of Output Iterator.
- InputIterator's value type is convertible to a type in OutputIterator's set of value types.

Preconditions

- `[first, last)` is a valid range.
- `result` is not an iterator within the range `[first, last)`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last - first))` is a valid range.

Complexity

Linear. Exactly `last - first` assignments are performed.

Example

```
vector<int> V(5);
iota(V.begin(), V.end(), 1);

list<int> L(V.size());
copy(V.begin(), V.end(), L.begin());
assert(equal(V.begin(), V.end(), L.begin()));
```

Notes

Note the implications of this. `Copy` cannot be used to insert elements into an empty Container: it overwrites elements, rather than inserting elements. If you want to insert elements into a Sequence, you can either use its `insert` member function explicitly, or else you can use `copy` along with an `insert_iterator` adaptor. The order of assignments matters in the case where the input and output ranges overlap: `copy` may not be used if `result` is in the range `[first, last)`. That is, it may not be used if the beginning of the output range overlaps with the input range, but it may be used if the end of the output range overlaps with the input range; `copy_backward` has opposite restrictions. If the two ranges are completely nonoverlapping, of course, then either algorithm may be used. The order of assignments also matters if `result` is an `ostream_iterator`, or some other iterator whose semantics depends on the order or assignments.

See also

`copy_backward`, `copy_n`

9.2.2 copy_n

Prototype

```
template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size count,
                      OutputIterator result);
```

Description

`Copy_n` copies elements from the range `[first, first + n)` to the range `[result, result + n)`. That is, it performs the assignments `*result = *first`, `*(result + 1) = *(first + 1)`, and so on. Generally, for every integer `i` from `0` up to (but not including) `n`, `copy_n` performs the assignment `*(result + i) = *(first + i)`. Assignments are performed in forward order, *i.e.* in order of increasing `n`. The return value is `result + n`.

Definition

Defined in the standard header algorithm

Requirements on types

- InputIterator is a model of Input Iterator.
- OutputIterator is a model of Output Iterator.
- `Size` is an integral type.
- InputIterator's value type is convertible to a type in OutputIterator's set of value types.

Preconditions

- `n >= 0`.
- `[first, first + n)` is a valid range.
- `result` is not an iterator within the range `[first, first + n)`.
- `[result, result + n)` is a valid range.

Complexity

Linear. Exactly `n` assignments are performed.

Example

```
vector<int> V(5);
iota(V.begin(), V.end(), 1);

list<int> L(V.size());
copy_n(V.begin(), V.size(), L.begin());
assert(equal(V.begin(), V.end(), L.begin()));
```

Notes

`Copy_n` is almost, but not quite, redundant. If `first` is an input iterator, as opposed to a forward iterator, then the `copy_n` operation can't be expressed in terms of `copy`.

See also

copy, copy_backward

9.2.3 copy_backward

Prototype

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);
```

Description

`Copy_backward` copies elements from the range `[first, last)` to the range `[result - (last - first), result)` . That is, it performs the assignments `*(result - 1) = *(last - 1)`, `*(result - 2) = *(last - 2)`, and so on. Generally, for every integer `n` from `0` to `last - first`, `copy_backward` performs the assignment `*(result - n - 1) = *(last - n - 1)`. Assignments are performed from the end of the input sequence to the beginning, *i.e.* in order of increasing `n`. The return value is `result - (last - first)`

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- BidirectionalIterator1 and BidirectionalIterator2 are models of BidirectionalIterator.
- BidirectionalIterator1's value type is convertible to BidirectionalIterator2's value type.

Preconditions

- `[first, last)` is a valid range.
- `result` is not an iterator within the range `[first, last)`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result - (last - first), result)` is a valid range.

Complexity

Linear. Exactly `last - first` assignments are performed.

Example

```
vector<int> V(15);
iota(V.begin(), V.end(), 1);
copy_backward(V.begin(), V.begin() + 10, V.begin() + 15);
```

Notes

`Result` is an iterator that points to the *end* of the output range. This is highly unusual: in all other STL algorithms that denote an output range by a single iterator, that iterator points to the beginning of the range. The order of assignments matters in the case where the input and output ranges overlap: `copy_backward` may not be used if `result` is in the range `[first, last)`. That is, it may not be used if the end of the output range overlaps with the input range, but it may be used if the beginning of the output range overlaps with the input range; `copy` has opposite restrictions. If the two ranges are completely nonoverlapping, of course, then either algorithm may be used.

See also

`copy`, `copy_n`

9.2.4 Swap

swap

Prototype

```
template <class Assignable>
void swap(Assignable& a, Assignable& b);
```

Description

Assigns the contents of `a` to `b` and the contents of `b` to `a`. This is used as a primitive operation by many other algorithms.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- Assignable is a model of Assignable.

Preconditions

None.

Complexity

Amortized constant time.

Example

```
int x = 1;
int y = 2;
assert(x == 1 && y == 2);
swap(x, y);
assert(x == 2 && y == 1);
```

Notes

The time required to `swap` two objects of type `T` will obviously depend on the type; "constant time" does not mean that performance will be the same for an 8-bit `char` as for a 128-bit `complex<double>`.

See also

`iter_swap`, `swap_ranges`

iter_swap

Prototype

```
template <class ForwardIterator1, class ForwardIterator2>
inline void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Description

Equivalent to `swap(*a, *b)`.

Definition

Declared in algorithm.

Requirements on types

- `ForwardIterator1` and `ForwardIterator2` are models of Forward Iterator.
- `ForwardIterator1` and `ForwardIterator2` are mutable.
- `ForwardIterator1` and `ForwardIterator2` have the same value type.

Preconditions

- `ForwardIterator1` and `ForwardIterator2` are dereferenceable.

Complexity

See `swap` for a discussion.

Example

```
int x = 1;
int y = 2;
assert(x == 1 && y == 2);
iter_swap(&x, &y);
assert(x == 2 && y == 1);
```

Notes

Strictly speaking, `iter_swap` is redundant. It exists only for technical reasons: in some circumstances, some compilers have difficulty performing the type deduction required to interpret `swap(*a, *b)`.

See also

`swap`, `swap_ranges`

swap_ranges

Prototype

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                             ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

Description

`Swap_ranges` swaps each of the elements in the range `[first1, last1)` with the corresponding element in the range `[first2, first2 + (last1 - first1))`. That is, for each integer `n` such that `0 <= n < (last1 - first1)`, it swaps `*(first1 + n)` and `*(first2 + n)`. The return value is `first2 + (last1 - first1)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

`ForwardIterator1` and `ForwardIterator2` must both be models of Forward Iterator. The value types of `ForwardIterator1` and `ForwardIterator2` must be convertible to each other.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, first2 + (last1 - first1))` is a valid range.
- The two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` do not overlap.

Complexity

Linear. Exactly `last1 - first1` swaps are performed.

Example

```
vector<int> V1, V2;
V1.push_back(1);
V1.push_back(2);
V2.push_back(3);
V2.push_back(4);

assert(V1[0] == 1 && V1[1] == 2 && V2[0] == 3 && V2[1] == 4);
swap_ranges(V1.begin(), V1.end(), V2.begin());
assert(V1[0] == 3 && V1[1] == 4 && V2[0] == 1 && V2[1] == 2);
```

Notes

See also

`swap`, `iter_swap`.

9.2.5 transform

Prototype

`Transform` is an overloaded name; there are actually two `transform` functions.

```
template <class InputIterator, class OutputIterator,
          class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction op);


template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryFunction>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryFunction binary_op);
```

Description

`Transform` performs an operation on objects; there are two versions of `transform`, one of which uses a single range of Input Iterators and one of which uses two ranges of Input Iterators. The first version of `transform` performs the operation `op(*i)` for each iterator `i` in the range `[first, last)`, and assigns the result of that operation to `*o`, where `o` is the corresponding output iterator. That is, for each `n` such that `0 <= n < last - first`, it performs the assignment `*(result + n) = op(*(first + n))`. The return value is `result + (last - first)`. The second version of `transform` is very similar, except that it uses a Binary Function instead of a Unary Function: it performs the operation `op(*i1, *i2)` for each iterator `i1` in the range `[first1, last1)` and assigns the result to `*o`, where `i2` is the corresponding iterator in the second input range and where `o` is the corresponding output iterator. That is, for each `n` such that `0 <= n < last1 - first1`, it performs the assignment `*(result + n) = op(*(first1 + n), *(first2 + n)`. The return value is `result + (last1 - first1)`. Note that `transform` may be used to modify a sequence "in place": it is permissible for the iterators `first` and `result` to be the same.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first (unary) version:

- `InputIterator` must be a model of Input Iterator.
- `OutputIterator` must be a model of Output Iterator.
- `UnaryFunction` must be a model of Unary Function.
- `InputIterator`'s value type must be convertible to `UnaryFunction`'s argument type.
- `UnaryFunction`'s result type must be convertible to a type in `OutputIterator`'s set of value types.

For the second (binary) version:

- `InputIterator1` and `InputIterator2` must be models of Input Iterator.
- `OutputIterator` must be a model of Output Iterator.
- `BinaryFunction` must be a model of Binary Function.
- `InputIterator1`'s and `InputIterator2`'s value types must be convertible, respectively, to `BinaryFunction`'s first and second argument types.
- `UnaryFunction`'s result type must be convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first (unary) version:

- `[first, last)` is a valid range.
- `result` is not an iterator within the range `[first+1, last)`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last - first))` is a valid range.

For the second (binary) version:

- `[first1, last1)` is a valid range.
- `[first2, first2 + (last1 - first1))` is a valid range.
- `result` is not an iterator within the range `[first1+1, last1)` or `[first2 + 1, first2 + (last1 - first1))`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last1 - first1))` is a valid range.

Complexity

Linear. The operation is applied exactly `last - first` times in the case of the unary version, or `last1 - first1` in the case of the binary version.

Example

Replace every number in an array with its negative.

```
const int N = 1000;
double A[N];
iota(A, A+N, 1);

transform(A, A+N, A, negate<double>());
```

Calculate the sum of two vectors, storing the result in a third vector.

```
const int N = 1000;
vector<int> V1(N);
vector<int> V2(N);
vector<int> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          plus<int>());
```

Notes

The Output Iterator `result` is not permitted to be the same as any of the Input Iterators in the range `[first, last)`, with the exception of `first` itself. That is: `transform(V.begin(), V.end(), V.begin(), fabs)` is valid, but `transform(V.begin(), V.end(), V.begin() + 1, fabs)` is not.

See also

The function object overview, `copy`, `generate`, `fill`

9.2.6 Replace

replace

Prototype

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value)
```

Description

`Replace` replaces every element in the range `[first, last)` equal to `old_value` with `new_value`. That is: for every iterator `i`, if `*i == old_value` then it performs the assignment `*i = new_value`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `T` is convertible to `ForwardIterator`'s value type.
- `T` is Assignable.
- `T` is EqualityComparable, and may be compared for equality with objects of `ForwardIterator`'s value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. `Replace` performs exactly `last - first` comparisons for equality, and at most `last - first` assignments.

Example

```
vector<int> V;
V.push_back(1);
V.push_back(2);
V.push_back(3);
V.push_back(1);

replace(V.begin(), V.end(), 1, 99);
assert(V[0] == 99 && V[3] == 99);
```

Notes

See also

`replace_if`, `replace_copy`, `replace_copy_if`

replace_if

Prototype

```
template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value)
```

Description

`Replace_if` replaces every element in the range `[first, last)` for which `pred` returns `true` with `new_value`. That is: for every iterator `i`, if `pred(*i)` is `true` then it performs the assignment `*i = new_value`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `Predicate` is a model of Predicate.
- `ForwardIterator`'s value type is convertible to `Predicate`'s argument type.
- `T` is convertible to `Forward Iterator`'s value type.
- `T` is Assignable.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. `Replace_if` performs exactly `last - first` applications of `pred`, and at most `last - first` assignments.

Example

Replace every negative number with 0.

```
vector<int> V;
V.push_back(1);
V.push_back(-3);
V.push_back(2);
V.push_back(-1);

replace_if(V.begin(), V.end(), bind2nd(less<int>(), 0), -1);
assert(V[1] == 0 && V[3] == 0);
```

Notes

See also

`replace`, `replace_copy`, `replace_copy_if`

replace_copy

Prototype

```
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

`Replace_copy` copies elements from the range `[first, last)` to the range `[result, result + (last-first))`, except that any element equal to `old_value` is not copied; `new_value` is copied instead. More precisely, for every integer `n` such that `0 <= n < last-first`, `replace_copy` performs the assignment `*(result+n) = new_value` if `*(first+n) == old_value`, and `*(result+n) = *(first+n)` otherwise.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `T` is EqualityComparable, and may be compared for equality with objects of `InputIterator`'s value type.
- `T` is Assignable.
- `T` is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, last)` is a valid range.
- There is enough space in the output range to store the copied values. That is, `[result, result + (last-first))` is a valid range.
- `result` is not an iterator within the range `[first, last)`.

Complexity

Linear. `Replace_copy` performs exactly `last - first` comparisons for equality and exactly `last - first` assignments.

Example

```
vector<int> V1;
V1.push_back(1);
V1.push_back(2);
V1.push_back(3);
V1.push_back(1);

vector<int> V2(4);

replace_copy(V1.begin(), V1.end(), V2.begin(), 1, 99);
assert(V[0] == 99 && V[1] == 2 && V[2] == 3 && V[3] == 99);
```

Notes

See also

`copy`, `replace`, `replace_if`, `replace_copy_if`

replace_copy_if

Prototype

```
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

`Replace_copy_if` copies elements from the range `[first, last)` to the range `[result, result + (last-first))`, except that any element for which `pred` is `true` is not copied; `new_value` is copied instead. More precisely, for every integer `n` such that `0 <= n < last-first`, `replace_copy_if` performs the assignment `*(result+n) = new_value` if `pred(*(first+n))`, and `*(result+n) = *(first+n)` otherwise.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `Predicate` is a model of Predicate.
- `T` is convertible to `Predicate`'s argument type.
- `T` is Assignable.
- `T` is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, last)` is a valid range.
- There is enough space in the output range to store the copied values. That is, `[result, result + (last-first))` is a valid range.
- `result` is not an iterator within the range `[first, last)`.

Complexity

Linear. `Replace_copy` performs exactly `last - first` applications of `pred` and exactly `last - first` assignments.

Example

Copy elements from one vector to another, replacing all negative numbers with `0`.

```
vector<int> V1;
V1.push_back(1);
V1.push_back(-1);
V1.push_back(-5);
V1.push_back(2);

vector<int> V2(4);

replace_copy_if(V1.begin(), V1.end(), V2.begin(),
                bind2nd(less<int>(), 0),
                0);
assert(V[0] == 1 && V[1] == 0 && V[2] == 0 && V[3] == 2);
```

Notes

See also

`copy`, `replace`, `replace_if`, `replace_copy`

9.2.7 fill

Prototype

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Description

`Fill` assigns the value `value` to every element in the range `[first, last)`. That is, for every iterator `i` in `[first, last)`, it performs the assignment `*i = value`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `T` is a model of Assignable.
- `T` is convertible to Forward Iterator's value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. `Fill` performs exactly `last - first` assignments.

Example

```
vector<double> V(4);
fill(V.begin(), V.end(), 137);
assert(V[0] == 137 && V[1] == 137 && V[2] == 137 && V[3] == 137);
```

Notes

The reason that `fill` requires its argument to be a mutable forward iterator, rather than merely an output iterator, is that it uses a range `[first, last)` of iterators. There is no sensible way to describe a range of output iterators, because it is impossible to compare two output iterators for equality. The `fill_n` algorithm does have an interface that permits use of an output iterator.

See also

`copy`, `fill_n`, `generate`, `generate_n`, `iota`

9.2.8 fill_n

Prototype

```
template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

Description

`Fill_n` assigns the value `value` to every element in the range `[first, first+n)`. That is, for every iterator `i` in `[first, first+n)`, it performs the assignment `*i = value`. The return value is `first + n`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `OutputIterator` is a model of Output Iterator.
- `Size` is an integral type (either signed or unsigned).
- `T` is a model of Assignable.
- `T` is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `n >= 0`.
- There is enough space to hold `n` values. That is, `[first, first+n)` is a valid range.

Complexity

Linear. `Fill_n` performs exactly `n` assignments.

Example

```
vector<double> V;
fill_n(back_inserter(V), 4, 137);
assert(V.size() == 4 &&
       V[0] == 42 && V[1] == 42 && V[2] == 42 && V[3] == 42);
```

Notes

See also

`copy`, `fill`, `generate`, `generate_n`, `iota`

9.2.9 generate

Prototype

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
              Generator gen);
```

Description

`Generate` assigns the result of invoking `gen`, a function object that takes no arguments, to each element in the range `[first, last)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `Generator` is a model of Generator.
- `Generator`'s result type is convertible to `ForwardIterator`'s value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `last - first` invocations of `gen`.

Example

Fill a vector with random numbers, using the standard C library function `rand`.

```
vector<int> V;
...
generate(V.begin(), V.end(), rand);
```

Notes

The function object `gen` is invoked for each iterator in the range `[first, last)`, as opposed to just being invoked a single time outside the loop. This distinction is important because a Generator need not return the same result each time it is invoked; it is permitted to read from a file, refer to and modify local state, and so on. The reason that `generate` requires its argument to be a mutable Forward Iterator, rather than just an Output Iterator, is that it uses a range `[first, last)` of iterators. There is no sensible way to describe a range of Output Iterators, because it is impossible to compare two Output Iterators for equality. The `generate_n` algorithm does have an interface that permits use of an Output Iterator.

See also

`copy`, `fill`, `fill_n`, `generate_n`, `iota`

9.2.10 generate_n

Prototype

```
template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

Description

Generate_n assigns the result of invoking gen, a function object that takes no arguments, to each element in the range [first, first+n). The return value is first + n.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- OutputIterator is a model of Output Iterator.
- Size is an integral type (either signed or unsigned).
- Generator is a model of Generator.
- Generator's result type is convertible to a type in OutputIterator's set of value types.

Preconditions

- n >= 0.
- There is enough space to hold n values. That is, [first, first+n) is a valid range.

Complexity

Linear. Exactly n invocations of gen.

Example

Print 100 random numbers, using the C standard library function rand.

```
generate_n(ostream_iterator<int>(cout, "\n"), 100, rand);
```

Notes

The function object `gen` is invoked `n` times (once for each iterator in the range `[first, first+n)`), as opposed to just being invoked a single time outside the loop. This distinction is important because a Generator need not return the same result each time it is invoked; it is permitted to read from a file, refer to and modify local state, and so on.

See also

`copy`, `fill`, `fill_n`, `generate`, `iota`

9.2.11 Remove

remove

Prototype

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

Description

`Remove` removes from the range `[first, last)` all elements that are equal to `value`. That is, `remove` returns an iterator `new_last` such that the range `[first, new_last)` contains no elements equal to `value`. The iterators in the range `[new_last, last)` are all still dereferenceable, but the elements that they point to are unspecified. `Remove` is stable, meaning that the relative order of elements that are not equal to `value` is unchanged.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `T` is a model of Equality Comparable.
- Objects of type `T` can be compared for equality with objects of `ForwardIterator`'s value type.

Preconditions

- [first, last) is a valid range.

Complexity

Linear. Remove performs exactly last - first comparisons for equality.

Example

```
vector<int> V;
V.push_back(3);
V.push_back(1);
V.push_back(4);
V.push_back(1);
V.push_back(5);
V.push_back(9);

copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
    // The output is "3 1 4 1 5 9".

vector<int>::iterator new_end = remove(V.begin(), V.end(), 1);
copy(V.begin(), new_end, ostream_iterator<int>(cout, " "));
    // The output is "3 4 5 9".
```

Notes

The meaning of "removal" is somewhat subtle. Remove does not destroy any iterators, and does not change the distance between first and last. (There's no way that it could do anything of the sort.) So, for example, if V is a vector, remove(V.begin(), V.end(), 0) does not change V.size(): V will contain just as many elements as it did before. Remove returns an iterator that points to the end of the resulting range after elements have been removed from it; it follows that the elements after that iterator are of no interest, and may be discarded. If you are removing elements from a Sequence, you may simply erase them. That is, a reasonable way of removing elements from a Sequence is S.erase(remove(S.begin(), S.end(), x), S.end()).

See also

remove_if, remove_copy, remove_copy_if, unique, unique_copy.

remove_if

Prototype

```
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

Description

`Remove_if` removes from the range `[first, last)` every element `x` such that `pred(x)` is `true`. That is, `remove_if` returns an iterator `new_last` such that the range `[first, new_last)` contains no elements for which `pred` is `true`. The iterators in the range `[new_last, last)` are all still dereferenceable, but the elements that they point to are unspecified. `Remove_if` is stable, meaning that the relative order of elements that are not removed is unchanged.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `Predicate` is a model of Predicate.
- `ForwardIterator`'s value type is convertible to `Predicate`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. `Remove_if` performs exactly `last - first` applications of `pred`.

Example

Remove all even numbers from a vector.

```
vector<int> V;
V.push_back(1);
V.push_back(4);
V.push_back(2);
V.push_back(8);
V.push_back(5);
V.push_back(7);

copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
    // The output is "1 4 2 8 5 7"

vector<int>::iterator new_end =
        remove_if(V.begin(), V.end(),
                  compose1(bind2nd(equal_to<int>(), 0),
                           bind2nd(modulus<int>(), 2)));
V.erase(new_end, V.end()); [1]

copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
    // The output is "1 5 7".
```

Notes

The meaning of "removal" is somewhat subtle. `Remove_if` does not destroy any iterators, and does not change the distance between `first` and `last`. (There's no way that it could do anything of the sort.) So, for example, if `V` is a vector, `remove_if(V.begin(), V.end(), pred)` does not change `V.size()`: `V` will contain just as many elements as it did before. `Remove_if` returns an iterator that points to the end of the resulting range after elements have been removed from it; it follows that the elements after that iterator are of no interest, and may be discarded. If you are removing elements from a Sequence, you may simply erase them. That is, a reasonable way of removing elements from a Sequence is `S.erase(remove_if(S.begin(), S.end(), pred), S.end())`.

See also

`remove`, `remove_copy`, `remove_copy_if`, `unique`, `unique_copy`.

remove_copy

Prototype

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

Description

`Remove_copy` copies elements that are not equal to `value` from the range `[first, last)` to a range beginning at `result`. The return value is the end of the resulting range. This operation is stable, meaning that the relative order of the elements that are copied is the same as in the range `[first, last)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.
- `T` is a model of Equality Comparable.
- Objects of type `T` can be compared for equality with objects of `InputIterator`'s value type.

Preconditions

- `[first, last)` is a valid range.
- There is enough space in the output range to store the copied values. That is, if there are `n` elements in `[first, last)` that are not equal to `value`, then `[result, result+n)` is a valid range.
- `result` is not an iterator in the range `[first, last)`.

Complexity

Linear. Exactly `last - first` comparisons for equality, and at most `last - first` assignments.

Example

Print all nonzero elements of a vector on the standard output.

```
vector<int> V;
V.push_back(-2);
V.push_back(0);
V.push_back(-1);
V.push_back(0);
V.push_back(1);
V.push_back(2);

remove_copy(V.begin(), V.end(),
            ostream_iterator<int>(cout, "\n"),
            0);
```

Notes

See also

`copy`, `remove`, `remove_if`, `remove_copy_if`, `unique`, `unique_copy`.

remove_copy_if

Prototype

```
template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

Description

`Remove_copy_if` copies elements from the range `[first, last)` to a range beginning at `result`, except that elements for which `pred` is `true` are not copied. The return value is the end of the resulting range. This operation is stable, meaning that the relative order of the elements that are copied is the same as in the range `[first, last)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.
- `Predicate` is a model of Predicate.
- `InputIterator`'s value type is convertible to `Predicate`'s argument type.

Preconditions

- `[first, last)` is a valid range.
- There is enough space in the output range to store the copied values. That is, if there are `n` elements in `[first, last)` that do not satisfy `pred`, then `[result, result+n)` is a valid range.
- `result` is not an iterator in the range `[first, last)`.

Complexity

Linear. Exactly `last - first` applications of `pred`, and at most `last - first` assignments.

Example

Fill a vector with the nonnegative elements of another vector.

```
vector<int> V1;
V.push_back(-2);
V.push_back(0);
V.push_back(-1);
V.push_back(0);
V.push_back(1);
V.push_back(2);

vector<int> V2;
remove_copy_if(V1.begin(), V1.end(),
               back_inserter(V2),
               bind2nd(less<int>(), 0));
```

Notes

See also

copy, remove, remove_if, remove_copy, unique, unique_copy.

9.2.12 unique

Prototype

Unique is an overloaded name; there are actually two unique functions.

```
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate binary_pred);
```

Description

Every time a consecutive group of duplicate elements appears in the range [first, last), the algorithm unique removes all but the first element. That is, unique returns an iterator new_last such that the range [first, new_last) contains no two consecutive elements that are duplicates. The iterators in the range [new_last, last) are all still dereferenceable, but the elements that they point to are unspecified. Unique is stable, meaning that the relative order of elements that are not removed is unchanged. The reason there are two different versions of unique is that there are two different definitions of what it means for a consecutive group of elements to be duplicates. In the first version, the test is simple equality: the elements in a range [f, l) are duplicates if, for every iterator i in the range, either i == f or else *i == *(i-1). In the second, the test is an arbitrary Binary Predicate binary_pred: the elements in [f, l) are duplicates if, for every iterator i in the range, either i == f or else binary_pred(*i, *(i-1)) is true.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `ForwardIterator`'s value type is Equality Comparable.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s first argument type and to `BinaryPredicate`'s second argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `(last - first) - 1` applications of `operator==` (in the case of the first version of `unique`) or of `binary_pred` (in the case of the second version).

Example

Remove duplicates from consecutive groups of equal `int`s.

```
vector<int> V;
V.push_back(1);
V.push_back(3);
V.push_back(3);
V.push_back(3);
V.push_back(2);
V.push_back(2);
V.push_back(1);

vector<int>::iterator new_end = unique(V.begin(), V.end());
copy(V.begin(), new_end, ostream_iterator<int>(cout, " "));
    // The output it "1 3 2 1".
```

Remove all duplicates from a vector of `char`s, ignoring case. First sort the vector, then remove duplicates from consecutive groups.

```
inline bool eq_nocase(char c1, char c2)
  { return tolower(c1) == tolower(c2); }
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  const char init[] = "The Standard Template Library";
  vector<char> V(init, init + sizeof(init));
  sort(V.begin(), V.end(), lt_nocase);
  copy(V.begin(), V.end(), ostream_iterator<char>(cout));
  cout << endl;
  vector<char>::iterator new_end = unique(V.begin(), V.end(),
         eq_nocase);
  copy(V.begin(), new_end, ostream_iterator<char>(cout));
  cout << endl;
}
// The output is:
//    aaaabddeeehiLlmnprrrStTtTy
//  abdehiLmnprSty
```

Notes

Note that the meaning of "removal" is somewhat subtle. `Unique`, like `remove`, does not destroy any iterators and does not change the distance between `first` and `last`. (There's no way that it could do anything of the sort.) So, for example, if `V` is a vector, `remove(V.begin(), V.end(), 0)` does not change `V.size()`: `V` will contain just as many elements as it did before. `Unique` returns an iterator that points to the end of the resulting range after elements have been removed from it; it follows that the elements after that iterator are of no interest. If you are operating on a Sequence, you may wish to use the Sequence's `erase` member function to discard those elements entirely. Strictly speaking, the first version of `unique` is redundant: you can achieve the same functionality by using an object of class `equal_to` as the Binary Predicate argument. The first version is provided strictly for the sake of convenience: testing for equality is an important special case. `BinaryPredicate` is not required to be an equivalence relation. You should be cautious, though, about using `unique` with a Binary Predicate that is not an equivalence relation: you could easily get unexpected results.

See also

`Binary Predicate`, `remove`, `remove_if`, `unique_copy`, `adjacent_find`,

9.2.13 unique_copy

Prototype

`Unique_copy` is an overloaded name; there are actually two `unique_copy` functions.

```
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

`Unique_copy` copies elements from the range `[first, last)` to a range beginning with `result`, except that in a consecutive group of duplicate elements only the first one is copied. The return value is the end of the range to which the elements are copied. This behavior is similar to the Unix filter `uniq`. The reason there are two different versions of `unique_copy` is that there are two different definitions of what it means for a consecutive group of elements to be duplicates. In the first version, the test is simple equality: the elements in a range `[f, l)` are duplicates if, for every iterator `i` in the range, either `i == f` or else `*i == *(i-1)`. In the second, the test is an arbitrary Binary Predicate `binary_pred`: the elements in `[f, l)` are duplicates if, for every iterator `i` in the range, either `i == f` or else `binary_pred(*i, *(i-1))` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator` is a model of Input Iterator.
- `InputIterator`'s value type is Equality Comparable.
- `OutputIterator` is a model of Output Iterator.

- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator` is a model of Input Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `InputIterator`'s value type is convertible to first argument type and to `BinaryPredicate`'s second argument type.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, last)` is a valid range.
- There is enough space to hold all of the elements being copied. More formally, if there are `n` elements in the range `[first, last)` after duplicates are removed from consecutive groups, then `[result, result + n)` must be a valid range.

Complexity

Linear. Exactly `last - first` applications of `operator==` (in the case of the first version of `unique`) or of `binary_pred` (in the case of the second version), and at most `last - first` assignments.

Example

Print all of the numbers in an array, but only print the first one in a consecutive group of identical numbers.

```
const int A[] = {2, 7, 7, 7, 1, 1, 8, 8, 8, 2, 8, 8};
unique_copy(A, A + sizeof(A) / sizeof(int),
            ostream_iterator<int>(cout, " "));
// The output is "2 7 1 8 2 8".
```

Notes

Strictly speaking, the first version of `unique_copy` is redundant: you can achieve the same functionality by using an object of class `equal_to` as the Binary Predicate argument. The first version is provided strictly for the sake of convenience: testing for equality is an important special case. `BinaryPredicate` is not required to be an equivalence relation. You should be cautious, though, about using `unique_copy` with a Binary Predicate that is not an equivalence relation: you could easily get unexpected results.

See also

Binary Predicate, `unique`, `remove_copy`, `remove_copy_if`, `adjacent_find`

9.2.14 reverse

Prototype

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Description

`Reverse` reverses a range. That is: for every `i` such that `0 <= i <= (last - first) / 2`, it exchanges `*(first + i)` and `*(last - (i + 1))`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- BidirectionalIterator is a model of Bidirectional Iterator.
- BidirectionalIterator is mutable.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear: `reverse(first, last)` makes `(last - first) / 2` calls to `swap`.

Example

```
vector<int> V;
V.push_back(0);
V.push_back(1);
V.push_back(2);
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
                // Output: 0 1 2
reverse(V.begin(), V.end());
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
                // Output: 2 1 0
```

Notes

See also

`reverse_copy`

9.2.15 reverse_copy

Prototype

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

Description

`Reverse_copy` copies elements from the range `[first, last)` to the range `[result, result + (last - first))` such that the copy is a reverse of the original range. Specifically: for every `i` such that `0 <= i < (last - first)`, `reverse_copy` performs the assignment `*(result + (last - first) - i) = *(first + i)`. The return value is `result + (last - first)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- BidirectionalIterator is a model of Bidirectional Iterator.
- OutputIterator is a model of Output Iterator.
- The value type of BidirectionalIterator is convertible to a type in OutputIterator's set of value types.

Preconditions

- `[first, last)` is a valid range.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last - first))` is a valid range.
- The ranges `[first, last)` and `[result, result + (last - first))` do not overlap.

Complexity

Linear: exactly `last - first` assignments.

Example

```
vector<int> V;
V.push_back(0);
V.push_back(1);
V.push_back(2);
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
                // Output: 0 1 2
list<int> L(V.size());
reverse_copy(V.begin(), V.end(), L.begin());
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
                // Output: 2 1 0
```

Notes

See also

`reverse`, `copy`

9.2.16 rotate

Prototype

```
template <class ForwardIterator>
inline ForwardIterator rotate(ForwardIterator first,
                              ForwardIterator middle,
                              ForwardIterator last);
```

Description

`Rotate` rotates the elements in a range. That is, the element pointed to by `middle` is moved to the position `first`, the element pointed to by `middle + 1` is moved to the position `first + 1`, and so on. One way to think about this operation is that it exchanges the two ranges `[first, middle)` and `[middle, last)`. Formally, for every integer `n` such that `0 <= n < last - first`, the element `*(first + n)` is assigned to `*(first + (n + (last - middle)) % (last - first))`. `Rotate` returns `first + (last - middle)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.

Preconditions

- `[first, middle)` is a valid range.
- `[middle, last)` is a valid range.

Complexity

Linear. At most `last - first` swaps are performed.

Example

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz";
rotate(alpha, alpha + 13, alpha + 26);
printf("\%s\n", alpha);
// The output is nopqrstuvwxyzabcdefghijklm
```

Notes

It follows from these two requirements that `[first, last)` is a valid range. `Rotate` uses a different algorithm depending on whether its arguments are Forward Iterators, Bidirectional Iterators, or Random Access Iterators. All three algorithms, however, are linear.

See also

`rotate_copy`

9.2.17 rotate_copy

Prototype

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first,
                           ForwardIterator middle,
                           ForwardIterator last,
                           OutputIterator result);
```

Description

`Rotate_copy` copies elements from the range `[first, last)` to the range `[result, result + (last - first))` such that `*middle` is copied to `*result`, `*(middle + 1)` is copied to `*(result + 1)`, and so on. Formally, for every integer `n` such that `0 <= n < last - first`, `rotate_copy` performs the assignment `*(result + (n + (last - middle)) % (last - first)) = *(first + n)`. `Rotate_copy` is similar to `copy` followed by `rotate`, but is more efficient. The return value is `result + (last - first)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `OutputIterator` is a model of Output Iterator.
- `ForwardIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, middle)` is a valid range.
- `[middle, last)` is a valid range.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last - first))` is a valid range.
- The ranges `[first, last)` and `[result, result + (last - first))` do not overlap.

Complexity

Linear. `Rotate_copy` performs exactly `last - first` assignments.

Example

```
const char alpha[] = "abcdefghijklmnopqrstuvwxyz";
rotate_copy(alpha, alpha + 13, alpha + 26,
            ostream_iterator<char>(cout));
// The output is nopqrstuvwxyzabcdefghijklm
```

Notes

It follows from these two requirements that `[first, last)` is a valid range.

See also

`rotate`, `copy`.

9.2.18 random_shuffle

Prototype

`Random_shuffle` is an overloaded name; there are actually two `random_shuffle` functions.

```
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

`Random_shuffle` randomly rearranges the elements in the range `[first, last)`: that is, it randomly picks one of the `N!` possible orderings, where `N` is `last - first`. There are two different versions of `random_shuffle`. The first version uses an internal random number generator, and the second uses a Random Number Generator, a special kind of function object, that is explicitly passed as an argument.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator
- `RandomNumberGenerator` is a model of Random Number Generator
- `RandomAccessIterator`'s distance type is convertible to `RandomNumberGenerator`'s argument type.

Preconditions

- `[first, last)` is a valid range.
- `last - first` is less than `rand`'s maximum value.

Complexity

Linear in `last - first`. If `last != first`, exactly `(last - first) - 1` swaps are performed.

Example

```
const int N = 8;
int A[] = {1, 2, 3, 4, 5, 6, 7, 8};
random_shuffle(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The printed result might be 7 1 6 3 2 5 4 8,
//  or any of 40,319 other possibilities.
```

Notes

This algorithm is described in section 3.4.2 of Knuth (D. E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, second edition. Addison-Wesley, 1981). Knuth credits Moses and Oakford (1963) and Durstenfeld (1964). Note that there are N! ways of arranging a sequence of N elements. `Random_shuffle` yields uniformly distributed results; that is, the probability of any particular ordering is 1/N!. The reason this comment is important is that there are a number of algorithms that seem at first sight to implement random shuffling of a sequence, but that do not in fact produce a uniform distribution over the N! possible orderings. That is, it's easy to get random shuffle wrong.

See also

`random_sample`, `random_sample_n`, `next_permutation`, `prev_permutation`, Random Number Generator

9.2.19 partition

Prototype

```
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
                          ForwardIterator last, Predicate pred)
```

Description

`Partition` reorders the elements in the range `[first, last)` based on the function object `pred`, such that the elements that satisfy `pred` precede the elements that fail to satisfy it. The postcondition is that, for some iterator `middle` in the range `[first, last)`, `pred(*i)` is `true` for every iterator `i` in the range `[first, middle)` and `false` for every iterator `i` in the range `[middle, last)`. The return value of `partition` is `middle`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `Predicate` is a model of Predicate.
- `ForwardIterator`'s value type is convertible to `Predicate`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `last - first` applications of `pred`, and at most `(last - first)/2` swaps.

Example

Reorder a sequence so that even numbers precede odd numbers.

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
partition(A, A + N,
          compose1(bind2nd(equal_to<int>(), 0),
                   bind2nd(modulus<int>(), 2)));
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is "10 2 8 4 6 5 7 3 9 1". [1]
```

Notes

The relative order of elements in these two blocks is not necessarily the same as it was in the original sequence. A different algorithm, `stable_partition`, does guarantee to preserve the relative order.

See also

`stable_partition`, Predicate, function object

9.2.20 stable_partition

Prototype

```
template <class ForwardIterator, class Predicate>
ForwardIterator stable_partition(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

Description

`Stable_partition` is much like `partition`: it reorders the elements in the range `[first, last)` based on the function object `pred`, such that all of the elements that satisfy `pred` appear before all of the elements that fail to satisfy it. The postcondition is that, for some iterator `middle` in the range `[first, last)`, `pred(*i)` is `true` for every iterator `i` in the range `[first, middle)` and `false` for every iterator `i` in the range `[middle, last)`. The return value of `stable_partition` is `middle`. `Stable_partition` differs from `partition` in that `stable_partition` is guaranteed to preserve relative order. That is, if `x` and `y` are elements in `[first, last)` such that `pred(x) == pred(y)`, and if `x` precedes `y`, then it will still be true after `stable_partition` is true that `x` precedes `y`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator
- `Predicate` is a model of Predicate
- `ForwardIterator`'s value type is convertible to `Predicate`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

`Stable_partition` is an *adaptive* algorithm: it attempts to allocate a temporary memory buffer, and its run-time complexity depends on how much memory is available. Worst-case behavior (if no auxiliary memory is available) is at most `N*log(N)` swaps, where `N` is `last - first`, and best case (if a large enough auxiliary memory buffer is available) is linear in `N`. In either case, `pred` is applied exactly `N` times.

Example

Reorder a sequence so that even numbers precede odd numbers.

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
stable_partition(A, A + N,
                 compose1(bind2nd(equal_to<int>(), 0),
                          bind2nd(modulus<int>(), 2)));
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is "2 4 6 8 10 1 3 5 7 9". [1]
```

Notes

Note that the complexity of `stable_partition` is greater than that of `partition`: the guarantee that relative order will be preserved has a significant runtime cost. If this guarantee isn't important to you, you should use `partition`.

See also

`partition`, Predicate, function object

9.3 Sorting

9.3.1 Sort

sort

Prototype

`Sort` is an overloaded name; there are actually two `sort` functions.

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Description

`Sort` sorts the elements in `[first, last)` into ascending order, meaning that if `i` and `j` are any two valid iterators in `[first, last)` such that `i` precedes `j`, then `*j` is not less than `*i`. Note: `sort` is not guaranteed to be stable. That is, suppose that `*i` and `*j` are equivalent: neither one is less than the other. It is not guaranteed that the relative order of these two elements will be preserved by `sort`. The two versions of `sort` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes two arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is LessThan Comparable.
- The ordering relation on `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version, the one that takes three arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- [first, last) is a valid range.

Complexity

O(N log(N)) comparisons (both average and worst-case), where N is last - first.

Example

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is " 1 2 4 5 7 8".
```

Notes

Stable sorting is sometimes important if you are sorting records that have multiple fields: you might, for example, want to sort a list of people by first name and then by last name. The algorithm stable_sort does guarantee to preserve the relative ordering of equivalent elements.

See also

stable_sort, partial_sort, partial_sort_copy, sort_heap, is_sorted, binary_search, lower_bound, upper_bound, less<T>, StrictWeakOrdering, LessThan Comparable

stable_sort

Prototype

Stable_sort is an overloaded name; there are actually two stable_sort functions.

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 StrictWeakOrdering comp);
```

Description

`Stable_sort` is much like `sort`: it sorts the elements in `[first, last)` into ascending order, meaning that if `i` and `j` are any two valid iterators in `[first, last)` such that `i` precedes `j`, then `*j` is not less than `*i`. `Stable_sort` differs from `sort` in two ways. First, `stable_sort` uses an algorithm that has different run-time complexity than `sort`. Second, as the name suggests, `stable_sort` is stable: it preserves the relative ordering of equivalent elements. That is, if `x` and `y` are elements in `[first, last)` such that `x` precedes `y`, and if the two elements are equivalent (neither `x < y` nor `y < x`) then a postcondition of `stable_sort` is that `x` still precedes `y`. The two versions of `stable_sort` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes two arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is LessThan Comparable.
- The ordering relation on `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version, the one that takes three arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

`Stable_sort` is an *adaptive* algorithm: it attempts to allocate a temporary memory buffer, and its run-time complexity depends on how much memory is available. Worst-case behavior (if no auxiliary memory is available) is `N (log N)^2` comparisons, where `N` is `last - first`, and best case (if a large enough auxiliary memory buffer is available) is `N (log N)`.

Example

Sort a sequence of characters, ignoring their case. Note that the relative order of characters that differ only by case is preserved.

```
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  char A[] = "fdBeACFDbEac";
  const int N = sizeof(A) - 1;
  stable_sort(A, A+N, lt_nocase);
  printf("\%s\n", A);
  // The printed result is ""AaBbCcdDeEfF".
}
```

Notes

Note that two elements may be equivalent without being equal. One standard example is sorting a sequence of names by last name: if two people have the same last name but different first names, then they are equivalent but not equal. This is why `stable_sort` is sometimes useful: if you are sorting a sequence of records that have several different fields, then you may want to sort it by one field without completely destroying the ordering that you previously obtained from sorting it by a different field. You might, for example, sort by first name and then do a stable sort by last name. `Stable_sort` uses the *merge sort* algorithm; see section 5.2.4 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1975.)

See also

`sort`, `partial_sort`, `partial_sort_copy`, `binary_search`, `lower_bound`, `upper_bound`, `less<T>`, StrictWeakOrdering, LessThan Comparable

partial_sort

Prototype

`Partial_sort` is an overloaded name; there are actually two `partial_sort` functions.

```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last,
                  StrictWeakOrdering comp);
```

Description

`Partial_sort` rearranges the elements in the range `[first, last)` so that they are partially in ascending order. Specifically, it places the smallest `middle - first` elements, sorted in ascending order, into the range `[first, middle)`. The remaining `last - middle` elements are placed, in an unspecified order, into the range `[middle, last)`. The two versions of `partial_sort` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The postcondition for the first version of `partial_sort` is as follows. If `i` and `j` are any two valid iterators in the range `[first, middle)` such that `i` precedes `j`, and if `k` is a valid iterator in the range `[middle, last)`, then `*j < *i` and `*k < *i` will both be `false`. The corresponding postcondition for the second version of `partial_sort` is that `comp(*j, *i)` and `comp(*k, *i)` are both false. Informally, this postcondition means that the first `middle - first` elements are in ascending order and that none of the elements in `[middle, last)` is less than any of the elements in `[first, middle)`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is LessThan Comparable.
- The ordering relation on `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, middle)` is a valid range.
- `[middle, last)` is a valid range.

(It follows from these two conditions that `[first, last)` is a valid range.)

Complexity

Approximately `(last - first) * log(middle - first)` comparisons.

Example

```
int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);

partial_sort(A, A + 5, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The printed result is "1 2 3 4 5 11 12 10 9 8 7 6".
```

Notes

Note that the elements in the range [first, middle) will be the same (ignoring, for the moment, equivalent elements) as if you had sorted the entire range using sort(first, last). The reason for using partial_sort in preference to sort is simply efficiency: a partial sort, in general, takes less time. partial_sort(first, last, last) has the effect of sorting the entire range [first, last), just like sort(first, last). They use different algorithms, however: sort uses the *introsort* algorithm (a variant of quicksort), and partial_sort uses *heapsort*. See section 5.2.3 of Knuth (D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1975.), and J. W. J. Williams (*CACM* **7**, 347, 1964). Both heapsort and introsort have complexity of order N log(N), but introsort is usually faster by a factor of 2 to 5.

See also

partial_sort_copy, sort, stable_sort, binary_search, lower_bound, upper_bound, less<T>, StrictWeakOrdering, LessThan Comparable

partial_sort_copy

Prototype

Partial_sort_copy is an overloaded name; there are actually two partial_sort_copy functions.

```
template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last);

template <class InputIterator, class RandomAccessIterator,
          class StrictWeakOrdering>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last, Compare comp);
```

Description

Partial_sort_copy copies the smallest N elements from the range [first, last) to the range [result_first, result_first + N), where N is the smaller of last - first and result_last - result_first. The elements in [result_first, result_first + N) will be in ascending order. The two versions of `partial_sort_copy` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The postcondition for the first version of `partial_sort_copy` is as follows. If `i` and `j` are any two valid iterators in the range `[result_first, result_first + N)` such that `i` precedes `j`, then `*j < *i` will be `false`. The corresponding postcondition for the second version is that `comp(*j, *i)` will be `false`. The return value is `result_first + N`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator` is a model of InputIterator.
- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- The value types of `InputIterator` and `RandomAccessIterator` are the same.
- `RandomAccessIterator`'s value type is LessThan Comparable.
- The ordering relation on `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `InputIterator` is a model of InputIterator.
- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- The value types of `InputIterator` and `RandomAccessIterator` are the same.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- [first, last) is a valid range.
- [result_first, result_last) is a valid range.
- [first, last) and [result_first, result_last) do not overlap.

Complexity

Approximately (last - first) * log(N) comparisons, where N is the smaller of last - first and result_last - result_first.

Example

```
int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);

vector<int> V(4);
partial_sort_copy(A, A + N, V.begin(), V.end());
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
// The printed result is "1 2 3 4".
```

Notes

See also

partial_sort, sort, stable_sort, binary_search, lower_bound, upper_bound, less<T>, StrictWeakOrdering, LessThan Comparable

is_sorted

Prototype

Is_sorted is an overloaded name; there are actually two is_sorted functions.

```
template <class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last)

template <class ForwardIterator, class StrictWeakOrdering>
bool is_sorted(ForwardIterator first, ForwardIterator last,
               StrictWeakOrdering comp)
```

Description

`Is_sorted` returns `true` if the range `[first, last)` is sorted in ascending order, and `false` otherwise. The two versions of `is_sorted` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using the function object `comp`. The first version of `is_sorted` returns `true` if and only if, for every iterator `i` in the range `[first, last - 1)`, `*(i + 1) < *i` is `false`. The second version returns `true` if and only if, for every iterator `i` in the range `[first, last - 1)`, `comp(*(i + 1), *i)` is `false`

Definition

Defined in algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `ForwardIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `ForwardIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Zero comparisons if `[first, last)` is an empty range, otherwise at most `(last - first) - 1` comparisons.

Example

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);

assert(!is_sorted(A, A + N));
sort(A, A + N);
assert(is_sorted(A, A + N));
```

Notes

See also

`sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`, `sort_heap`, `binary_search`, `lower_bound`, `upper_bound`, `less<T>`, StrictWeakOrdering, LessThan Comparable

9.3.2 nth_element

Prototype

`Nth_element` is an overloaded name; there are actually two `nth_element` functions.

```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, StrictWeakOrdering comp);
```

Description

`Nth_element` is similar to `partial_sort`, in that it partially orders a range of elements: it arranges the range `[first, last)` such that the element pointed to by the iterator `nth` is the same as the element that would be in that position if the entire range `[first, last)` had been sorted. Additionally, none of the elements in the range `[nth, last)` is less than any of the elements in the range `[first, nth)`. The two versions of `nth_element` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The postcondition for the first version of `nth_element` is as follows. There exists no iterator `i` in the range `[first, nth)` such that `*nth < *i`, and there exists no iterator `j` in the range [nth + 1, last) such that *j < *nth. The postcondition for the second version of nth_element is as follows. There exists no iterator i in the range [first, nth) such that comp(*nth, *i) is true, and there exists no iterator j in the range [nth + 1, last) such that comp(*j, *nth) is true.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes three arguments:

- RandomAccessIterator is a model of Random Access Iterator.
- RandomAccessIterator is mutable.
- RandomAccessIterator's value type is LessThan Comparable.
- The ordering relation on RandomAccessIterator's value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version, the one that takes four arguments:

- RandomAccessIterator is a model of Random Access Iterator.
- RandomAccessIterator is mutable.
- StrictWeakOrdering is a model of Strict Weak Ordering.
- RandomAccessIterator's value type is convertible to StrictWeakOrdering's argument type.

Preconditions

- [first, nth) is a valid range.
- [nth, last) is a valid range.

(It follows from these two conditions that [first, last) is a valid range.)

Complexity

On average, linear in last - first.

Example

```
int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);

nth_element(A, A + 6, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The printed result is "5 2 6 1 4 3 7 8 9 10 11 12".
```

Notes

The way in which this differs from `partial_sort` is that neither the range `[first, nth)` nor the range `[nth, last)` is be sorted: it is simply guaranteed that none of the elements in `[nth, last)` is less than any of the elements in `[first, nth)`. In that sense, `nth_element` is more similar to `partition` than to `sort`. `Nth_element` does less work than `partial_sort`, so, reasonably enough, it is faster. That's the main reason to use `nth_element` instead of `partial_sort`. Note that this is significantly less than the run-time complexity of `partial_sort`.

See also

`partial_sort`, `partition`, `sort`, StrictWeakOrdering, LessThan Comparable

9.3.3 Binary search

lower_bound

Prototype

`Lower_bound` is an overloaded name; there are actually two `lower_bound` functions.

```
template <class ForwardIterator, class LessThanComparable>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const LessThanComparable& value);

template <class ForwardIterator, class T, class StrictWeakOrdering>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, StrictWeakOrdering comp);
```

Description

`Lower_bound` is a version of binary search: it attempts to find the element `value` in an ordered range `[first, last)` . Specifically, it returns the first position where `value` could be inserted without violating the ordering. The first version of `lower_bound` uses `operator<` for comparison, and the second uses the function object `comp`. The first version of `lower_bound` returns the furthermost iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, i)`, `*j < value`. The second version of `lower_bound` returns the furthermost iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, i)`, `comp(*j, value)` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `LessThanComparable` is a model of LessThan Comparable.
- The ordering on objects of type `LessThanComparable` is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `ForwardIterator`'s value type is the same type as `LessThanComparable`.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `ForwardIterator`'s value type is the same type as `T`.
- `ForwardIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first, last)` is a valid range.
- `[first, last)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first, last)` such that `i` precedes `j`, `*j < *i` is `false`.

For the second version:

- `[first, last)` is a valid range.
- `[first, last)` is ordered in ascending order according to the function object `comp`. That is, for every pair of iterators `i` and `j` in `[first, last)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Complexity

The number of comparisons is logarithmic: at most `log(last - first) + 1`. If `ForwardIterator` is a Random Access Iterator then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to `last - first`.

Example

```
int main()
{
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    int* p = lower_bound(A, A + N, i);
    cout << "Searching for " << i << ".  ";
    cout << "Result: index = " << p - A << ", ";
    if (p != A + N)
      cout << "A[" << p - A << "] == " << *p << endl;
    else
      cout << "which is off-the-end." << endl;
  }
}
```

The output is:

```
Searching for 1.  Result: index = 0, A[0] == 1
Searching for 2.  Result: index = 1, A[1] == 2
Searching for 3.  Result: index = 2, A[2] == 3
Searching for 4.  Result: index = 5, A[5] == 5
Searching for 5.  Result: index = 5, A[5] == 5
Searching for 6.  Result: index = 6, A[6] == 8
Searching for 7.  Result: index = 6, A[6] == 8
Searching for 8.  Result: index = 6, A[6] == 8
Searching for 9.  Result: index = 7, which is off-the-end.
Searching for 10.  Result: index = 7, which is off-the-end.
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all `false`. (See the LessThan Comparable requirements for a more complete discussion.) Finding `value` in the range `[first, last)`, then, doesn't mean finding an element that is equal to `value` but rather one that is *equivalent to* `value`: one that is neither greater than nor less than `value`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same. If an element that is equivalent to `value` is already present in the range `[first, last)`, then the return value of `lower_bound` will be an iterator that points to that element. This difference between Random Access Iterators and Forward Iterators is simply because `advance` is constant time for Random Access Iterators and linear time for Forward Iterators.

See also

`upper_bound`, `equal_range`, `binary_search`

upper_bound

Prototype

`Upper_bound` is an overloaded name; there are actually two `upper_bound` functions.

```
template <class ForwardIterator, class LessThanComparable>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const LessThanComparable& value);

template <class ForwardIterator, class T, class StrictWeakOrdering>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, StrictWeakOrdering comp);
```

Description

`Upper_bound` is a version of binary search: it attempts to find the element `value` in an ordered range `[first, last)` . Specifically, it returns the last position where `value` could be inserted without violating the ordering. The first version of `upper_bound` uses `operator<` for comparison, and the second uses the function object `comp`. The first version of `upper_bound` returns the furthermost iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, i)`, `value < *j` is `false`. The second version of `upper_bound` returns the furthermost iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, i)`, `comp(value, *j)` is `false`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `LessThanComparable` is a model of LessThan Comparable.
- The ordering on objects of type `LessThanComparable` is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `ForwardIterator`'s value type is the same type as `LessThanComparable`.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `ForwardIterator`'s value type is the same type as `T`.
- `ForwardIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first, last)` is a valid range.
- `[first, last)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first, last)` such that `i` precedes `j`, `*j < *i` is `false`.

For the second version:

- `[first, last)` is a valid range.
- `[first, last)` is ordered in ascending order according to the function object `comp`. That is, for every pair of iterators `i` and `j` in `[first, last)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Complexity

The number of comparisons is logarithmic: at most `log(last - first) + 1`. If `ForwardIterator` is a Random Access Iterator then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to `last - first`.

Example

```
int main()
{
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    int* p = upper_bound(A, A + N, i);
    cout << "Searching for " << i << ".  ";
    cout << "Result: index = " << p - A << ", ";
    if (p != A + N)
      cout << "A[" << p - A << "] == " << *p << endl;
    else
      cout << "which is off-the-end." << endl;
  }
}
```

The output is:

```
Searching for 1.  Result: index = 1, A[1] == 2
Searching for 2.  Result: index = 2, A[2] == 3
Searching for 3.  Result: index = 5, A[5] == 5
Searching for 4.  Result: index = 5, A[5] == 5
Searching for 5.  Result: index = 6, A[6] == 8
Searching for 6.  Result: index = 6, A[6] == 8
Searching for 7.  Result: index = 6, A[6] == 8
Searching for 8.  Result: index = 7, which is off-the-end.
Searching for 9.  Result: index = 7, which is off-the-end.
Searching for 10.  Result: index = 7, which is off-the-end.
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all `false`. (See the LessThan Comparable requirements for a more complete discussion.) Finding `value` in the range `[first, last)`, then, doesn't mean finding an element that is equal to `value` but rather one that is *equivalent to* `value`: one that is neither greater than nor less than `value`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same. Note that even if an element that is equivalent to `value` is already present in the range `[first, last)`, the return value of `upper_bound` will not point to that element. The return value is either `last` or else an iterator `i` such that `value < *i`. If `i` is not equal to `first`, however, then `*(i - 1)` is less than or equivalent to `value`. This difference between Random Access Iterators and Forward Iterators is simply because `advance` is constant time for Random Access Iterators and linear time for Forward Iterators.

See also

`lower_bound`, `equal_range`, `binary_search`

equal_range

Prototype

`Equal_range` is an overloaded name; there are actually two `equal_range` functions.

```
template <class ForwardIterator, class LessThanComparable>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
            const LessThanComparable& value);

template <class ForwardIterator, class T, class StrictWeakOrdering>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value,
            StrictWeakOrdering comp);
```

Description

`Equal_range` is a version of binary search: it attempts to find the element `value` in an ordered range `[first, last)` . The value returned by `equal_range` is essentially a combination of the values returned by `lower_bound` and `upper_bound`: it returns a pair of iterators `i` and `j` such that `i` is the first position where `value` could be inserted without violating the ordering and `j` is the last position where `value` could be inserted without violating the ordering. It follows that every element in the range `[i, j)` is equivalent to `value`, and that `[i, j)` is the largest subrange of `[first, last)` that has this property. The first version of `equal_range` uses `operator<` for comparison, and the second uses the function object `comp`. The first version of `equal_range` returns a pair of iterators `[i, j)`. `i` is the furthermost iterator in `[first, last)` such that, for every iterator `k` in `[first, i)`, `*k < value`. `j` is the furthermost iterator in `[first, last)` such that, for every iterator `k` in `[first,` j), value < *k is false. For every iterator k in [i, j), neither value < *k nor *k < value is true. The second version of equal_range returns a pair of iterators [i, j). i is the furthermost iterator in [first, last) such that, for every iterator k in [first, i), comp(*k, value) is true. j is the furthermost iterator in [first, last) such that, for every iterator k in [first, j), comp(value, *k) is false. For every iterator k in [i, j), neither comp(value, *k) nor comp(*k, value) is true.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- ForwardIterator is a model of Forward Iterator.
- LessThanComparable is a model of LessThan Comparable.
- The ordering on objects of type LessThanComparable is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- ForwardIterator's value type is the same type as LessThanComparable.

For the second version:

- ForwardIterator is a model of Forward Iterator.
- StrictWeakOrdering is a model of Strict Weak Ordering.
- ForwardIterator's value type is the same type as T.
- ForwardIterator's value type is convertible to StrictWeakOrdering's argument type.

Preconditions

For the first version:

- [first, last) is a valid range.
- [first, last) is ordered in ascending order according to operator<. That is, for every pair of iterators i and j in [first, last) such that i precedes j, *j < *i is false.

For the second version:

- `[first, last)` is a valid range.
- `[first, last)` is ordered in ascending order according to the function object `comp`. That is, for every pair of iterators `i` and `j` in `[first, last)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Complexity

The number of comparisons is logarithmic: at most `2 * log(last - first) + 1`. If `ForwardIterator` is a Random Access Iterator then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to `last - first`.

Example

```
int main()
{
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 2; i <= 4; ++i) {
    pair<int*, int*> result = equal_range(A, A + N, i);

    cout << endl;
    cout << "Searching for " << i << endl;
    cout << "  First position where " << i << " could be inserted: "
         << result.first - A << endl;
    cout << "  Last position where " << i << " could be inserted: "
         << result.second - A << endl;
    if (result.first < A + N)
      cout << "  *result.first = " << *result.first << endl;
    if (result.second < A + N)
      cout << "  *result.second = " << *result.second << endl;
  }
}
```

The output is:

```
Searching for 2
  First position where 2 could be inserted: 1
  Last position where 2 could be inserted: 2
  *result.first = 2
  *result.second = 3

Searching for 3
  First position where 3 could be inserted: 2
  Last position where 3 could be inserted: 5
  *result.first = 3
  *result.second = 5

Searching for 4
  First position where 4 could be inserted: 5
  Last position where 4 could be inserted: 5
  *result.first = 5
  *result.second = 5
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all `false`. (See the LessThan Comparable requirements for a more complete discussion.) Finding `value` in the range `[first, last)`, then, doesn't mean finding an element that is equal to `value` but rather one that is *equivalent to* `value`: one that is neither greater than nor less than `value`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same. Note that `equal_range` may return an empty range; that is, it may return a pair both of whose elements are the same iterator. `Equal_range` returns an empty range if and only if the range `[first, last)` contains no elements equivalent to `value`. In this case it follows that there is only one position where `value` could be inserted without violating the range's ordering, so the return value is a pair both of whose elements are iterators that point to that position. This difference between Random Access Iterators and Forward Iterators is simply because `advance` is constant time for Random Access Iterators and linear time for Forward Iterators.

See also

`lower_bound`, `upper_bound`, `binary_search`

binary_search

Prototype

Binary_search is an overloaded name; there are actually two binary_search functions.

```
template <class ForwardIterator, class LessThanComparable>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const LessThanComparable& value);

template <class ForwardIterator, class T, class StrictWeakOrdering>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, StrictWeakOrdering comp);
```

Description

`Binary_search` is a version of binary search: it attempts to find the element `value` in an ordered range `[first, last)` It returns `true` if an element that is equivalent to `value` is present in `[first, last)` and `false` if no such element exists. The first version of `binary_search` uses `operator<` for comparison, and the second uses the function object `comp`. Specifically, the first version returns `true` if and only if there exists an iterator `i` in `[first, last)` such that `*i < value` and `value < *i` are both `false`. The second version returns `true` if and only if there exists an iterator `i` in `[first, last)` such that `comp(*i, value)` and `comp(value, *i)` are both `false`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `LessThanComparable` is a model of LessThan Comparable.
- The ordering on objects of type `LessThanComparable` is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `ForwardIterator`'s value type is the same type as `LessThanComparable`.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.

- StrictWeakOrdering is a model of Strict Weak Ordering.
- ForwardIterator's value type is the same type as T.
- ForwardIterator's value type is convertible to StrictWeakOrdering's argument type.

Preconditions

For the first version:

- [first, last) is a valid range.
- [first, last) is ordered in ascending order according to operator<. That is, for every pair of iterators i and j in [first, last) such that i precedes j, *j < *i is false.

For the second version:

- [first, last) is a valid range.
- [first, last) is ordered in ascending order according to the function object comp. That is, for every pair of iterators i and j in [first, last) such that i precedes j, comp(*j, *i) is false.

Complexity

The number of comparisons is logarithmic: at most log(last - first) + 2. If ForwardIterator is a Random Access Iterator then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to last - first.

Example

```
int main()
{
  int A[] = { 1, 2, 3, 3, 3, 5, 8 };
  const int N = sizeof(A) / sizeof(int);

  for (int i = 1; i <= 10; ++i) {
    cout << "Searching for " << i << ": "
         << (binary_search(A, A + N, i) ? "present" : "not present")
         << endl;
  }
}
```

The output is:

```
Searching for 1: present
Searching for 2: present
Searching for 3: present
Searching for 4: not present
Searching for 5: present
Searching for 6: not present
Searching for 7: not present
Searching for 8: present
Searching for 9: not present
Searching for 10: not present
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all `false`. (See the LessThan Comparable requirements for a more complete discussion.) Finding `value` in the range `[first, last)`, then, doesn't mean finding an element that is equal to `value` but rather one that is *equivalent to* `value`: one that is neither greater than nor less than `value`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same. Note that this is not necessarily the information you are interested in! Usually, if you're testing whether an element is present in a range, you'd like to know where it is (if it's present), or where it should be inserted (if it's not present). The functions `lower_bound`, `upper_bound`, and `equal_range` provide this information. This difference between Random Access Iterators and Forward Iterators is simply because `advance` is constant time for Random Access Iterators and linear time for Forward Iterators.

See also

`lower_bound`, `upper_bound`, `equal_range`

9.3.4 merge

Prototype

`Merge` is an overloaded name: there are actually two `merge` functions.

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class StrictWeakOrdering>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, StrictWeakOrdering comp);
```

Description

`Merge` combines two sorted ranges `[first1, last1)` and `[first2, last2)` into a single sorted range. That is, it copies elements from `[first1, last1)` and `[first2, last2)` into `[result, result + (last1 - first1) + (last2 - first2))` such that the resulting range is in ascending order. `Merge` is stable, meaning both that the relative order of elements within each input range is preserved, and that for equivalent elements in both input ranges the element from the first range precedes the element from the second. The return value is `result + (last1 - first1) + (last2 - first2)`. The two versions of `merge` differ in how elements are compared. The first version uses `operator<`. That is, the input ranges and the output range satisfy the condition that for every pair of iterators `i` and `j` such that `i` precedes `j`, `*j < *i` is `false`. The second version uses the function object `comp`. That is, the input ranges and the output range satisfy the condition that for every pair of iterators `i` and `j` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `InputIterator1`'s value type is the same type as `InputIterator2`'s value type.
- `InputIterator1`'s value type is a model of LessThan Comparable.

- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator1`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `InputIterator1`'s value type is the same type as `InputIterator2`'s value type.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator1`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first1, last1)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.
- `[first2, last2)` is a valid range.
- `[first2, last2)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.
- The ranges `[first1, last1)` and `[result, result + (last1 - first1) + (last2 - first2))` do not overlap.
- The ranges `[first2, last2)` and `[result, result + (last1 - first1) + (last2 - first2))` do not overlap.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last1 - first1) + (last2 - first2))` is a valid range.

For the second version:

- `[first1, last1)` is a valid range.
- `[first1, last1)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- `[first2, last2)` is a valid range.
- `[first2, last2)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- The ranges `[first1, last1)` and `[result, result + (last1 - first1) + (last2 - first2))` do not overlap.
- The ranges `[first2, last2)` and `[result, result + (last1 - first1) + (last2 - first2))` do not overlap.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + (last1 - first1) + (last2 - first2))` is a valid range.

Complexity

Linear. No comparisons if both `[first1, last1)` and `[first2, last2)` are empty ranges, otherwise at most `(last1 - first1) + (last2 - first2) - 1` comparisons.

Example

```
int main()
{
  int A1[] = { 1, 3, 5, 7 };
  int A2[] = { 2, 4, 6, 8 };
  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);

  merge(A1, A1 + N1, A2, A2 + N2,
        ostream_iterator<int>(cout, " "));
  // The output is "1 2 3 4 5 6 7 8"
}
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all false. (See the LessThan Comparable requirements for a more complete discussion.) Two elements `x` and `y` are *equivalent* if neither `x < y` nor `y < x`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

inplace_merge, set_union, sort

9.3.5 inplace_merge

Prototype

Inplace_merge is an overloaded name: there are actually two inplace_merge functions.

```
template <class BidirectionalIterator>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last);

template <class BidirectionalIterator, class StrictWeakOrdering>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last,
                          StrictWeakOrdering comp);
```

Description

Inplace_merge combines two consecutive sorted ranges [first, middle) and [middle, last) into a single sorted range [first, last). That is, it starts with a range [first, last) that consists of two pieces each of which is in ascending order, and rearranges it so that the entire range is in ascending order. Inplace_merge is stable, meaning both that the relative order of elements within each input range is preserved, and that for equivalent elements in both input ranges the element from the first range precedes the element from the second. The two versions of inplace_merge differ in how elements are compared. The first version uses operator<. That is, the input ranges and the output range satisfy the condition that for every pair of iterators i and j such that i precedes j, *j < *i is false. The second version uses the function object comp. That is, the input ranges and the output range satisfy the condition that for every pair of iterators i and j such that i precedes j, comp(*j, *i) is false.

Definition

Defined in algo.h.

Requirements on types

For the first version:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `BidirectionalIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `BidirectionalIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `BidirectionalIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first, middle)` is a valid range.
- `[middle, last)` is a valid range.
- `[first, middle)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first, middle)` such that `i` precedes `j`, `*j < *i` is `false`.
- `[middle, last)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[middle, last)` such that `i` precedes `j`, `*j < *i` is `false`.

For the second version:

- `[first, middle)` is a valid range.
- `[middle, last)` is a valid range.
- `[first, middle)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[first, middle)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- `[middle, last)` is in ascending order. That is, for every pair of iterators `i` and `j` in `[middle, last)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Complexity

`Inplace_merge` is an *adaptive* algorithm: it attempts to allocate a temporary memory buffer, and its run-time complexity depends on how much memory is available. `Inplace_merge` performs no comparisons if `[first, last)` is an empty range. Otherwise, worst-case behavior (if no auxiliary memory is available) is `O(N log(N))`, where `N` is `last - first`, and best case (if a large enough auxiliary memory buffer is available) is at most `(last - first) - 1` comparisons.

Example

```
int main()
{
  int A[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

  inplace_merge(A, A + 4, A + 8);
  copy(A, A + 8, ostream_iterator<int>(cout, " "));
  // The output is "1 2 3 4 5 6 7 8".
}
```

Notes

Note that you may use an ordering that is a strict weak ordering but not a total ordering; that is, there might be values `x` and `y` such that `x < y`, `x > y`, and `x == y` are all false. (See the LessThan Comparable requirements for a fuller discussion.) Two elements `x` and `y` are *equivalent* if neither `x < y` nor `y < x`. If you're using a total ordering, however (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

`merge`, `set_union`, `sort`

9.3.6 Set operations on sorted ranges

includes

Prototype

`Includes` is an overloaded name; there are actually two `includes` functions.

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
          class StrictWeakOrdering>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              StrictWeakOrdering comp);
```

Description

`Includes` tests whether one sorted range includes another sorted range. That is, it returns `true` if and only if, for every element in `[first2, last2)`, an equivalent element is also present in `[first1, last1)` . Both `[first1, last1)` and `[first2, last2)` must be sorted in ascending order. The two versions of `includes` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using the function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.

- `InputIterator1` and `InputIterator2` have the same value type.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.
- `[first1, last1)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.
- `[first2, last2)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.

For the second version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.
- `[first1, last1)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- `[first2, last2)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

Complexity

Linear. Zero comparisons if either `[first1, last1)` or `[first2, last2)` is an empty range, otherwise at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

```
int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
int A2[] = { 1, 4, 7 };
int A3[] = { 2, 7, 9 };
int A4[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
int A5[] = { 1, 2, 13, 13 };
int A6[] = { 1, 1, 3, 21 };

const int N1 = sizeof(A1) / sizeof(int);
const int N2 = sizeof(A2) / sizeof(int);
const int N3 = sizeof(A3) / sizeof(int);
const int N4 = sizeof(A4) / sizeof(int);
const int N5 = sizeof(A5) / sizeof(int);
const int N6 = sizeof(A6) / sizeof(int);

cout << "A2 contained in A1: "
     << (includes(A1, A1 + N1, A2, A2 + N2) ? "true" : "false") << endl;
cout << "A3 contained in A1: "
     << (includes(A1, A1 + N2, A3, A3 + N3) ? "true" : "false") << endl;
cout << "A5 contained in A4: "
     << (includes(A4, A4 + N4, A5, A5 + N5) ? "true" : "false") << endl;
cout << "A6 contained in A4: "
     << (includes(A4, A4 + N4, A6, A6 + N6) ? "true" : "false") << endl;
```

The output is:

```
A2 contained in A1: true
A3 contained in A1: false
A5 contained in A4: false
A6 contained in A4: true
```

Notes

This reads "an equivalent element" rather than "the same element" because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x` is true) but not equal. See the LessThan Comparable requirements for a fuller discussion.) If you're using a total ordering (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same. Note that the range `[first2, last2)` may contain a consecutive range of equivalent elements: there is no requirement that every element in the range be unique. In this case, `includes` will return `false` unless, for every element in `[first2, last2)`, a distinct equivalent element is also present in [first1, last1). That is, if a certain value appears n times in [first2, last2) and m times in [first1, last1), then includes will return false if m < n.

See also

set_union, set_intersection, set_difference, set_symmetric_difference, sort

set_union

Prototype

Set_union is an overloaded name; there are actually two set_union functions.

```
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 StrictWeakOrdering>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result,
                         StrictWeakOrdering comp);
```

Description

Set_union constructs a sorted range that is the union of the sorted ranges [first1, last1) and [first2, last2). The return value is the end of the output range. In the simplest case, set_union performs the "union" operation from set theory: the output range contains a copy of every element that is contained in [first1, last1), [first2, last2), or both. The general case is more complicated, because the input ranges may contain duplicate elements. The generalization is that if a value appears m times in [first1, last1) and n times in [first2, last2) (where m or n may be zero), then it appears max(m,n) times in the output range. Set_union is stable, meaning both that the relative order of elements within each input range is preserved, and that if an element is present in both input ranges it is copied from the first range rather than the second. The two versions of set_union differ in how they define whether one element is less than another. The first version compares objects using operator<, and the second compares objects using a function object comp.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

- `[first1, last1)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.
- `[first2, last2)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the union of the two input ranges.
- `[first1, last1)` and `[result, result + n)` do not overlap.
- `[first2, last2)` and `[result, result + n)` do not overlap.

For the second version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.
- `[first1, last1)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- `[first2, last2)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the union of the two input ranges.
- `[first1, last1)` and `[result, result + n)` do not overlap.
- `[first2, last2)` and `[result, result + n)` do not overlap.

Complexity

Linear. Zero comparisons if either `[first1, last1)` or `[first2, last2)` is empty, otherwise at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

```
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'B', 'B', 'f', 'H'};
  char A4[] = {'A', 'B', 'b', 'C', 'D', 'F', 'F', 'h', 'h'};

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Union of A1 and A2: ";
  set_union(A1, A1 + N1, A2, A2 + N2,
            ostream_iterator<int>(cout, " "));
  cout << endl
       << "Union of A3 and A4: ";
  set_union(A3, A3 + N3, A4, A4 + N4,
            ostream_iterator<char>(cout, " "),
            lt_nocase);
  cout << endl;
}
```

The output is

```
Union of A1 and A2: 1 1 2 3 5 7 8 9 11 13
Union of A3 and A4: a b B B C D f F H h
```

Notes

Even this is not a completely precise description, because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x`) but not equal. See the LessThan Comparable requirements for a more complete discussion. If the range `[first1, last1)` contains `m` elements that are equivalent to each other and the range `[first2, last2)` contains `n` elements from that equivalence class (where either `m` or `n` may be zero), then the output range contains `max(m, n)` elements from that equivalence class. Specifically, `m` of these elements will be copied from `[first1, last1)` and `max(n-m, 0)` of them will be copied from `[first2, last2)`. Note that this precision is only important if elements can be equivalent but not equal. If you're using a total ordering (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

`includes`, `set_intersection`, `set_difference`, `set_symmetric_difference`, `sort`, `merge`

set_intersection

Prototype

`Set_intersection` is an overloaded name; there are actually two `set_intersection` functions.

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
                                InputIterator1 last1,
                                InputIterator2 first2,
                                InputIterator2 last2,
                                OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class StrictWeakOrdering>
OutputIterator set_intersection(InputIterator1 first1,
                                InputIterator1 last1,
                                InputIterator2 first2,
                                InputIterator2 last2,
                                OutputIterator result,
                                StrictWeakOrdering comp);
```

Description

`Set_intersection` constructs a sorted range that is the intersection of the sorted ranges `[first1, last1)` and `[first2, last2)`. The return value is the end of the output range. In the simplest case, `set_intersection` performs the "intersection" operation from set theory: the output range contains a copy of every element that is contained in both `[first1, last1)` and `[first2, last2)`. The general case is more complicated, because the input ranges may contain duplicate elements. The generalization is that if a value appears `m` times in `[first1, last1)` and `n` times in `[first2, last2)` (where `m` or `n` may be zero), then it appears `min(m,n)` times in the output range. `Set_intersection` is stable, meaning both that elements are copied from the first range rather than the second, and that the relative order of elements in the output range is the same as in the first input range. The two versions of `set_intersection` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

- [first1, last1) is ordered in ascending order according to operator<. That is, for every pair of iterators i and j in [first1, last1) such that i precedes j, *j < *i is false.
- [first2, last2) is ordered in ascending order according to operator<. That is, for every pair of iterators i and j in [first2, last2) such that i precedes j, *j < *i is false.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that [result, result + n) is a valid range, where n is the number of elements in the intersection of the two input ranges.
- [first1, last1) and [result, result + n) do not overlap.
- [first2, last2) and [result, result + n) do not overlap.

For the second version:

- [first1, last1) is a valid range.
- [first2, last2) is a valid range.
- [first1, last1) is ordered in ascending order according to comp. That is, for every pair of iterators i and j in [first1, last1) such that i precedes j, comp(*j, *i) is false.
- [first2, last2) is ordered in ascending order according to comp. That is, for every pair of iterators i and j in [first2, last2) such that i precedes j, comp(*j, *i) is false.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that [result, result + n) is a valid range, where n is the number of elements in the intersection of the two input ranges.
- [first1, last1) and [result, result + n) do not overlap.
- [first2, last2) and [result, result + n) do not overlap.

Complexity

Linear. Zero comparisons if either [first1, last1) or [first2, last2) is empty, otherwise at most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.

Example

```
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Intersection of A1 and A2: ";
  set_intersection(A1, A1 + N1, A2, A2 + N2,
                   ostream_iterator<int>(cout, " "));
  cout << endl
       << "Intersection of A3 and A4: ";
  set_intersection(A3, A3 + N3, A4, A4 + N4,
                   ostream_iterator<char>(cout, " "),
                   lt_nocase);
  cout << endl;
}
```

The output is

```
Intersection of A1 and A2: 1 3 5
Intersection of A3 and A4: a b b f h
```

Notes

Even this is not a completely precise description, because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x`) but not equal. See the LessThan Comparable requirements for a fuller discussion. The output range consists of those elements from `[first1, last1)` for which equivalent elements exist in `[first2, last2)`. Specifically, if the range `[first1, last1)` contains `n` elements that are equivalent to each other and the range `[first1, last1)` contains `m` elements from that equivalence class (where either `m` or `n` may be zero), then the output range contains the first `min(m, n)` of these elements from `[first1, last1)`. Note that this precision is only important if elements can be equivalent but not equal. If you're using a total ordering (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

`includes`, `set_union`, `set_difference`, `set_symmetric_difference`, `sort`

set_difference

Prototype

`Set_difference` is an overloaded name; there are actually two `set_difference` functions.

```
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 StrictWeakOrdering>
OutputIterator set_difference(InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2,
                              InputIterator2 last2,
                              OutputIterator result,
                              StrictWeakOrdering comp);
```

Description

`Set_difference` constructs a sorted range that is the set difference of the sorted ranges `[first1, last1)` and `[first2, last2)`. The return value is the end of the output range. In the simplest case, `set_difference` performs the "difference" operation from set theory: the output range contains a copy of every element that is contained in `[first1, last1)` and not contained in `[first2, last2)`. The general case is more complicated, because the input ranges may contain duplicate elements. The generalization is that if a value appears `m` times in `[first1, last1)` and `n` times in `[first2, last2)` (where `m` or `n` may be zero), then it appears `max(m-n, 0)` times in the output range. `Set_difference` is stable, meaning both that elements are copied from the first range rather than the second, and that the relative order of elements in the output range is the same as in the first input range. The two versions of `set_difference` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

- `[first1, last1)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.

- `[first2, last2)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.

- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the difference of the two input ranges.

- `[first1, last1)` and `[result, result + n)` do not overlap.

- `[first2, last2)` and `[result, result + n)` do not overlap.

For the second version:

- `[first1, last1)` is a valid range.

- `[first2, last2)` is a valid range.

- `[first1, last1)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

- `[first2, last2)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the difference of the two input ranges.

- `[first1, last1)` and `[result, result + n)` do not overlap.

- `[first2, last2)` and `[result, result + n)` do not overlap.

Complexity

Linear. Zero comparisons if either `[first1, last1)` or `[first2, last2)` is empty, otherwise at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

```
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'g', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Difference of A1 and A2: ";
  set_difference(A1, A1 + N1, A2, A2 + N2,
                 ostream_iterator<int>(cout, " "));
  cout << endl
       << "Difference of A3 and A4: ";
  set_difference(A3, A3 + N3, A4, A4 + N4,
                  ostream_iterator<char>(cout, " "),
                  lt_nocase);
  cout << endl;
}
```

The output is

```
Difference of A1 and A2: 7 9 11
Difference of A3 and A4: B B g H
```

Notes

Even this is not a completely precise description, because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x`) but not equal. See the LessThan Comparable requirements for a fuller discussion. The output range consists of those elements from `[first1, last1)` for which equivalent elements do not exist in `[first2, last2)`. Specifically, if the range `[first1, last1)` contains `m` elements that are equivalent to each other and the range `[first2, last2)` contains `n` elements from that equivalence class (where either `m` or `n` may be zero), then the output range contains the *last* `max(m - n, 0)` of these elements from `[first1, last1)`. Note that this precision is only important if elements can be equivalent but not equal. If you're using a total ordering (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

includes, set_union, set_intersection, set_symmetric_difference, sort

set_symmetric_difference

Prototype

Set_symmetric_difference is an overloaded name; there are actually two set_symmetric_difference functions.

```
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 StrictWeakOrdering>
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2,
                                        OutputIterator result,
                                        StrictWeakOrdering comp);
```

Description

Set_symmetric_difference constructs a sorted range that is the set symmetric difference of the sorted ranges [first1, last1) and [first2, last2). The return value is the end of the output range. In the simplest case, set_symmetric_difference performs a set theoretic calculation: it constructs the union of the two sets A - B and B - A, where A and B are the two input ranges. That is, the output range contains a copy of every element that is contained in [first1, last1) but not [first2, last2), and a copy of every element that is contained in [first2, last2) but not [first1, last1). The general case is more complicated, because the input ranges may contain duplicate elements. The generalization is that if a value appears m times in [first1, last1) and n times in [first2, last2) (where m or n may be zero), then it appears |m-n| times in the output range. Set_symmetric_difference is stable, meaning that the relative order of elements within each input range is preserved. The two versions of set_symmetric_difference differ in how they define whether one element is less than another. The first version compares objects using operator<, and the second compares objects using a function object comp.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.

- `[first1, last1)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.

- `[first2, last2)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.

- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the symmetric difference of the two input ranges.

- `[first1, last1)` and `[result, result + n)` do not overlap.

- `[first2, last2)` and `[result, result + n)` do not overlap.

For the second version:

- `[first1, last1)` is a valid range.

- `[first2, last2)` is a valid range.

- `[first1, last1)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

- `[first2, last2)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.

- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the symmetric difference of the two input ranges.

- `[first1, last1)` and `[result, result + n)` do not overlap.

- `[first2, last2)` and `[result, result + n)` do not overlap.

Complexity

Linear. Zero comparisons if either `[first1, last1)` or `[first2, last2)` is empty, otherwise at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

```
inline bool lt_nocase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'g', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Symmetric difference of A1 and A2: ";
  set_symmetric_difference(A1, A1 + N1, A2, A2 + N2,
                           ostream_iterator<int>(cout, " "));
  cout << endl
       << "Symmetric difference of A3 and A4: ";
  set_symmetric_difference(A3, A3 + N3, A4, A4 + N4,
                           ostream_iterator<char>(cout, " "),
                           lt_nocase);
  cout << endl;
}
```

The output is

```
Symmetric difference of A1 and A2: 1 2 7 8 9 11 13
Symmetric difference of A3 and A4: B B C D F g H
```

Notes

Even this is not a completely precise description, because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x`) but not equal. See the LessThan Comparable requirements for a more complete discussion. The output range consists of those elements from `[first1, last1)` for which equivalent elements do not exist in `[first2, last2)`, and those elements from `[first2, last2)` for which equivalent elements do not exist in `[first1, last1)`. Specifically, suppose that the range `[first1, last1)` contains `m` elements that are equivalent to each other and the range `[first2, last2)` contains `n` elements from that equivalence class (where either `m` or `n` may be zero). If `m > n` then the output range contains the *last* `m - n` of these elements elements from `[first1, last1)`, and if `m < n` then the output range contains the last `n - m` of these elements elements from `[first2, last2)`.

See also

`includes`, `set_union`, `set_intersection`, `set_difference`, `sort`

9.3.7 Heap operations

push_heap

Prototype

`Push_heap` is an overloaded name; there are actually two `push_heap` functions.

```
template <class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               StrictWeakOrdering comp);
```

Description

`Push_heap` adds an element to a heap . It is assumed that `[first, last - 1)` is already a heap; the element to be added to the heap is `*(last - 1)`. The two versions of `push_heap` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The postcondition for the first version is that `is_heap(first, last)` is `true`, and the postcondition for the second version is that `is_heap(first, last, comp)` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is a model of LessThan Comparable.

- The ordering on objects of `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first, last)` is a valid range.
- `[first, last - 1)` is a valid range. That is, `[first, last)` is nonempty.
- `[first, last - 1)` is a heap. That is, `is_heap(first, last - 1)` is `true`.

For the second version:

- `[first, last)` is a valid range.
- `[first, last - 1)` is a valid range. That is, `[first, last)` is nonempty.
- `[first, last)` is a heap. That is, `is_heap(first, last - 1, comp)` is `true`.

Complexity

Logarithmic. At most `log(last - first)` comparisons.

Example

```
int main()
{
  int A[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  make_heap(A, A + 9);
  cout << "[A, A + 9)  = ";
  copy(A, A + 9, ostream_iterator<int>(cout, " "));

  push_heap(A, A + 10);
  cout << endl << "[A, A + 10) = ";
  copy(A, A + 10, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

The output is

```
[A, A + 9)  = 8 7 6 3 4 5 2 1 0
[A, A + 10) = 9 8 6 3 7 5 2 1 0 4
```

Notes

A heap is a particular way of ordering the elements in a range of random access iterators `[f, l)`. The reason heaps are useful (especially for sorting, or as priority queues) is that they satisfy two important properties. First, `*f` is the largest element in the heap. Second, it is possible to add an element to a heap (using `push_heap`), or to remove `*f`, in logarithmic time. Internally, a heap is a tree represented as a sequential range. The tree is constructed so that that each node is less than or equal to its parent node.

See also

`make_heap`, `pop_heap`, `sort_heap`, `is_heap`, `sort`

pop_heap

Prototype

`Pop_heap` is an overloaded name; there are actually two `pop_heap` functions.

```
template <class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
inline void pop_heap(RandomAccessIterator first,
                     RandomAccessIterator last,
                     StrictWeakOrdering comp);
```

Description

`Pop_heap` removes the largest element (that is, `*first`) from the heap `[first, last)`. The two versions of `pop_heap` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The postcondition for the first version of `pop_heap` is that `is_heap(first, last-1)` is `true` and that `*(last - 1)` is the element that was removed from the heap. The postcondition for the second version is that `is_heap(first, last-1, comp)` is `true` and that `*(last - 1)` is the element that was removed from the heap.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version:

- `[first, last)` is a valid range.
- `[first, last - 1)` is a valid range. That is, `[first, last)` is nonempty.
- `[first, last)` is a heap. That is, `is_heap(first, last)` is `true`.

For the second version:

- `[first, last)` is a valid range.
- `[first, last - 1)` is a valid range. That is, `[first, last)` is nonempty.
- `[first, last)` is a heap. That is, `is_heap(first, last, comp)` is `true`.

Complexity

Logarithmic. At most `2 * log(last - first)` comparisons.

Example

```
int main()
{
  int A[] = {1, 2, 3, 4, 5, 6};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  cout << "Before pop: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));

  pop_heap(A, A+N);
  cout << endl << "After pop: ";
  copy(A, A+N-1, ostream_iterator<int>(cout, " "));
  cout << endl << "A[N-1] = " << A[N-1] << endl;
}
```

The output is

```
Before pop: 6 5 3 4 2 1
After pop: 5 4 3 1 2
A[N-1] = 6
```

Notes

A heap is a particular way of ordering the elements in a range of Random Access Iterators `[f, l)`. The reason heaps are useful (especially for sorting, or as priority queues) is that they satisfy two important properties. First, `*f` is the largest element in the heap. Second, it is possible to add an element to a heap (using `push_heap`), or to remove `*f`, in logarithmic time. Internally, a heap is a tree represented as a sequential range. The tree is constructed so that that each node is less than or equal to its parent node. `Pop_heap` removes the largest element from a heap, and shrinks the heap. This means that if you call keep calling `pop_heap` until only a single element is left in the heap, you will end up with a sorted range where the heap used to be. This, in fact, is exactly how `sort_heap` is implemented.

See also

`make_heap`, `push_heap`, `sort_heap`, `is_heap`, `sort`

make_heap

Prototype

`Make_heap` is an overloaded name; there are actually two `make_heap` functions.

```
template <class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               StrictWeakOrdering comp);
```

Description

`Make_heap` turns the range `[first, last)` into a heap . The two versions of `make_heap` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. In the first version the postcondition is that `is_heap(first, last)` is `true`, and in the second version the postcondition is that `is_heap(first, last, comp)` is `true`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. At most `3*(last - first)` comparisons.

Example

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  sort_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

A heap is a particular way of ordering the elements in a range of Random Access Iterators [f, l). The reason heaps are useful (especially for sorting, or as priority queues) is that they satisfy two important properties. First, *f is the largest element in the heap. Second, it is possible to add an element to a heap (using push_heap), or to remove *f, in logarithmic time. Internally, a heap is simply a tree represented as a sequential range. The tree is constructed so that that each node is less than or equal to its parent node.

See also

push_heap, pop_heap, sort_heap, sort, is_heap

sort_heap

Prototype

Sort_heap is an overloaded name; there are actually two sort_heap functions.

```
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               StrictWeakOrdering comp);
```

Description

Sort_heap turns a heap [first, last) into a sorted range. Note that this is not a stable sort: the relative order of equivalent elements is not guaranteed to be preserved. The two versions of sort_heap differ in how they define whether one element is less than another. The first version compares objects using operator<, and the second compares objects using a function object comp.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes two arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `RandomAccessIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version, the one that takes three arguments:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

For the first version, the one that takes two arguments:

- `[first, last)` is a valid range.
- `[first, last)` is a heap. That is, `is_heap(first, last)` is `true`.

For the second version, the one that takes three arguments:

- `[first, last)` is a valid range.
- `[first, last)` is a heap. That is, `is_heap(first, last, comp)` is `true`.

Complexity

At most `N * log(N)` comparisons, where `N` is `last - first`.

Example

```
int main()
{
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  make_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  sort_heap(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

A heap is a particular way of ordering the elements in a range of Random Access Iterators `[f, l)`. The reason heaps are useful (especially for sorting, or as priority queues) is that they satisfy two important properties. First, `*f` is the largest element in the heap. Second, it is possible to add an element to a heap (using `push_heap`), or to remove `*f`, in logarithmic time. Internally, a heap is a tree represented as a sequential range. The tree is constructed so that that each node is less than or equal to its parent node.

See also

`push_heap`, `pop_heap`, `make_heap`, `is_heap`, `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`

is_heap

Prototype

`Is_heap` is an overloaded name; there are actually two is_heap functions.

```
template <class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
inline bool is_heap(RandomAccessIterator first,
                    RandomAccessIterator last,
                    StrictWeakOrdering comp)
```

Description

`Is_heap` returns `true` if the range `[first, last)` is a heap , and `false` otherwise. The two versions differ in how they define whether one element is less than another: the first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm.

Requirements on types

For the first version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `RandomAccessIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `RandomAccessIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `RandomAccessIterator` is a model of Random Access Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `RandomAccessIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Zero comparisons if `[first, last)` is an empty range, otherwise at most `(last - first) - 1` comparisons.

Example

```
int A[] = {1, 2, 3, 4, 5, 6, 7};
const int N = sizeof(A) / sizeof(int);

assert(!is_heap(A, A+N));
make_heap(A, A+N);
assert(is_heap(A, A+N));
```

Notes

A heap is a particular way of ordering the elements in a range of Random Access Iterators `[f, l)`. The reason heaps are useful (especially for sorting, or as priority queues) is that they satisfy two important properties. First, `*f` is the largest element in the heap. Second, it is possible to add an element to a heap (using `push_heap`), or to remove `*f`, in logarithmic time. Internally, a heap is a tree represented as a sequential range. The tree is constructed so that that each node is less than or equal to its parent node.

See also

`make_heap`, `push_heap`, `pop_heap`, `sort_heap`

9.3.8 Minimum and maximum

min

Prototype

`Min` is an overloaded name; there are actually two `min` functions.

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

template <class T, class BinaryPredicate>
const T& min(const T& a, const T& b, BinaryPredicate comp);
```

Description

`Min` returns the lesser of its two arguments; it returns the first argument if neither is less than the other. The two versions of `min` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using the function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `T` is a model of LessThan Comparable.

For the second version:

- `BinaryPredicate` is a model of Binary Predicate.
- `T` is convertible to `BinaryPredicate`'s first argument type and to its second argument type.

Preconditions

Complexity

Example

```
const int x = min(3, 9);
assert(x == 3);
```

Notes

See also

`max`, `min_element`, `max_element`, LessThan Comparable

max

Prototype

`Max` is an overloaded name; there are actually two `max` functions.

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

template <class T, class BinaryPredicate>
const T& max(const T& a, const T& b, BinaryPredicate comp);
```

Description

`Max` returns the greater of its two arguments; it returns the first argument if neither is greater than the other. The two versions of `max` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using the function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `T` is a model of LessThan Comparable.

For the second version:

- `BinaryPredicate` is a model of Binary Predicate.
- `T` is convertible to `BinaryPredicate`'s first argument type and to its second argument type.

Preconditions

Complexity

Example

```
const int x = max(3, 9);
assert(x == 9);
```

Notes

See also

`min`, `min_element`, `max_element`, LessThan Comparable

min_element

Prototype

`Min_element` is an overloaded name; there are actually two `min_element` functions.

```
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first,
                            ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator min_element(ForwardIterator first,
                            ForwardIterator last,
                            BinaryPredicate comp);
```

Description

`Min_element` finds the smallest element in the range `[first, last)`. It returns the first iterator `i` in `[first, last)` such that no other iterator in `[first, last)` points to a value smaller than `*i`. The return value is `last` if and only if `[first, last)` is an empty range. The two versions of `min_element` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The first version of `min_element` returns the first iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, last)`, `*j < *i` is `false`. The second version returns the first iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, last)`, `comp(*j, *i)` is `false`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator`'s value type is LessThan Comparable.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.

- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s first argument type and second argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Zero comparisons if `[first, last)` is an empty range, otherwise exactly `(last - first) - 1` comparisons.

Example

```
int main()
{
  list<int> L;
  generate_n(front_inserter(L), 1000, rand);

  list<int>::const_iterator it = min_element(L.begin(), L.end());
  cout << "The smallest element is " << *it << endl;
}
```

Notes

See also

`min`, `max`, `max_element`, LessThan Comparable, `sort`, `nth_element`

max_element

Prototype

`Max_element` is an overloaded name; there are actually two `max_element` functions.

```
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first,
                            ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator max_element(ForwardIterator first,
                            ForwardIterator last,
                            BinaryPredicate comp);
```

Description

`Max_element` finds the largest element in the range `[first, last)`. It returns the first iterator `i` in `[first, last)` such that no other iterator in `[first, last)` points to a value greater than `*i`. The return value is `last` if and only if `[first, last)` is an empty range. The two versions of `max_element` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`. The first version of `max_element` returns the first iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, last)`, `*i < *j` is `false`. The second version returns the first iterator `i` in `[first, last)` such that, for every iterator `j` in `[first, last)`, `comp(*i, *j)` is `false`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator`'s value type is LessThan Comparable.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `BinaryPredicate` is a model of Binary Predicate.
- `ForwardIterator`'s value type is convertible to `BinaryPredicate`'s first argument type and second argument type.

Preconditions

- [first, last) is a valid range.

Complexity

Linear. Zero comparisons if [first, last) is an empty range, otherwise exactly (last - first) - 1 comparisons.

Example

```
int main()
{
  list<int> L;
  generate_n(front_inserter(L), 1000, rand);

  list<int>::const_iterator it = max_element(L.begin(), L.end());
  cout << "The largest element is " << *it << endl;
}
```

Notes

See also

min, max, min_element, LessThan Comparable, sort, nth_element

9.3.9 lexicographical_compare

Prototype

Lexicographical_compare is an overloaded name; there are actually two lexicographical_compare functions.

```
template <class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1,
                             InputIterator1 last1,
                             InputIterator2 first2,
                             InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
bool lexicographical_compare(InputIterator1 first1,
                             InputIterator1 last1,
                             InputIterator2 first2,
                             InputIterator2 last2,
                             BinaryPredicate comp);
```

Description

`Lexicographical_compare` returns `true` if the range of elements `[first1, last1)` is lexicographically less than the range of elements `[first2, last2)`, and `false` otherwise. Lexicographical comparison means "dictionary" (element-by-element) ordering. That is, `[first1, last1)` is less than `[first2, last2)` if `*first1` is less than `*first2`, and greater if `*first1` is greater than `*first2`. If the two first elements are equivalent then `lexicographical_compare` compares the two second elements, and so on. As with ordinary dictionary order, the first range is considered to be less than the second if every element in the first range is equal to the corresponding element in the second but the second contains more elements. The two versions of `lexicographical_compare` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `InputIterator1`'s value type is a model of LessThan Comparable.
- `InputIterator2`'s value type is a model of LessThan Comparable.

- If v1 is an object of InputIterator1's value type and v2 is an object of InputIterator2's value type, then both v1 < v2 and v2 < v1 are defined.

For the second version:

- InputIterator1 is a model of Input Iterator.
- InputIterator2 is a model of Input Iterator.
- BinaryPredicate is a model of Binary Predicate.
- InputIterator1's value type is convertible to BinaryPredicate's first argument type and second argument type.
- InputIterator2's value type is convertible to BinaryPredicate's first argument type and second argument type.

Preconditions

- [first1, last1) is a valid range.
- [first2, last2) is a valid range.

Complexity

Linear. At most 2 * min(last1 - first1, last2 - first2) comparisons.

Example

```
int main()
{
  int A1[] = {3, 1, 4, 1, 5, 9, 3};
  int A2[] = {3, 1, 4, 2, 8, 5, 7};
  int A3[] = {1, 2, 3, 4};
  int A4[] = {1, 2, 3, 4, 5};

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3) / sizeof(int);
  const int N4 = sizeof(A4) / sizeof(int);

  bool C12 = lexicographical_compare(A1, A1 + N1, A2, A2 + N2);
  bool C34 = lexicographical_compare(A3, A3 + N3, A4, A4 + N4);

  cout << "A1[] < A2[]: " << (C12 ? "true" : "false") << endl;
  cout << "A3[] < A4[]: " << (C34 ? "true" : "false") << endl;
}
```

Notes

See also

equal, mismatch, lexicographical_compare_3way, search, LessThan Comparable, Strict Weak Ordering, sort

9.3.10 next_permutation

Prototype

Next_permutation is an overloaded name; there are actually two next_permutation functions.

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template <class BidirectionalIterator, class StrictWeakOrdering>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
                      StrictWeakOrdering comp);
```

Description

Next_permutation transforms the range of elements [first, last) into the lexicographically next greater permutation of the elements. There is a finite number of distinct permutations (at most N! , where N is last - first), so, if the permutations are ordered by lexicographical_compare, there is an unambiguous definition of which permutation is lexicographically next. If such a permutation exists, next_permutation transforms [first, last) into that permutation and returns true. Otherwise it transforms [first, last) into the lexicographically smallest permutation and returns false. The postcondition is that the new permutation of elements is lexicographically greater than the old (as determined by lexicographical_compare) if and only if the return value is true. The two versions of next_permutation differ in how they define whether one element is less than another. The first version compares objects using operator<, and the second compares objects using a function object comp.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes two arguments:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `BidirectionalIterator`'s value type is LessThan Comparable.
- The ordering relation on `BidirectionalIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version, the one that takes three arguments:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `BidirectionalIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. At most `(last - first) / 2` swaps.

Example

This example uses `next_permutation` to implement the worst known deterministic sorting algorithm. Most sorting algorithms are `O(N log(N))`, and even bubble sort is only `O(N^2)`. This algorithm is actually `O(N!)`.

```
template <class BidirectionalIterator>
void snail_sort(BidirectionalIterator first, BidirectionalIterator last)
{
  while (next_permutation(first, last)) {}
}

int main()
{
  int A[] = {8, 3, 6, 1, 2, 5, 7, 4};
  const int N = sizeof(A) / sizeof(int);

  snail_sort(A, A+N);
  copy(A, A+N, ostream_iterator<int>(cout, "\n"));
}
```

Notes

If all of the elements in `[first, last)` are distinct from each other, then there are exactly `N!` permutations. If some elements are the same as each other, though, then there are fewer. There are, for example, only three (`3!/2!`) permutations of the elements `1 1 2`. Note that the lexicographically smallest permutation is, by definition, sorted in nondecreasing order.

See also

`prev_permutation`, `lexicographical_compare`, LessThan Comparable, Strict Weak Ordering, `sort`

9.3.11 prev_permutation

Prototype

`Prev_permutation` is an overloaded name; there are actually two `prev_permutation` functions.

```
template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template <class BidirectionalIterator, class StrictWeakOrdering>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
                      StrictWeakOrdering comp);
```

Description

`Prev_permutation` transforms the range of elements `[first, last)` into the lexicographically next smaller permutation of the elements. There is a finite number of distinct permutations (at most `N!` , where `N` is `last - first`), so, if the permutations are ordered by `lexicographical_compare`, there is an unambiguous definition of which permutation is lexicographically previous. If such a permutation exists, `prev_permutation` transforms `[first, last)` into that permutation and returns `true`. Otherwise it transforms `[first, last)` into the lexicographically greatest permutation and returns `false`. The postcondition is that the new permutation of elements is lexicographically less than the old (as determined by `lexicographical_compare`) if and only if the return value is `true`. The two versions of `prev_permutation` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `BidirectionalIterator`'s value type is LessThan Comparable.
- The ordering relation on `BidirectionalIterator`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.

For the second version:

- `BidirectionalIterator` is a model of Bidirectional Iterator.
- `BidirectionalIterator` is mutable.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `BidirectionalIterator`'s value type is convertible to `StrictWeakOrdering`'s argument type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. At most `(last - first) / 2` swaps.

Example

```
int main()
{
  int A[] = {2, 3, 4, 5, 6, 1};
  const int N = sizeof(A) / sizeof(int);

  cout << "Initially:              ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  prev_permutation(A, A+N);
  cout << "After prev_permutation: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  next_permutation(A, A+N);
  cout << "After next_permutation: ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

If all of the elements in `[first, last)` are distinct from each other, then there are exactly `N!` permutations. If some elements are the same as each other, though, then there are fewer. There are, for example, only three (`3!/2!`) permutations of the elements `1 1 2`. Note that the lexicographically greatest permutation is, by definition, sorted in nonascending order.

See also

`next_permutation`, `lexicographical_compare`, LessThan Comparable, Strict Weak Ordering, `sort`

9.4 Generalized numeric algorithms

9.4.1 iota

Prototype

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

Description

`Iota` assigns sequentially increasing values to a range. That is, it assigns `value` to `*first`, `value + 1` to `*(first + 1)` and so on. In general, each iterator `i` in the range `[first, last)` is assigned `value + (i - first)`.

Definition

Defined in the standard header numeric.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `T` is Assignable.
- If `x` is an object of type `T`, then `x++` is defined.
- `T` is convertible to `ForwardIterator`'s value type.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `last - first` assignments.

Example

```
int main()
{
  vector<int> V(10);

  iota(V.begin(), V.end(), 7);
  copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

The name iota is taken from the programming language APL.

See also

fill, generate, partial_sum

9.4.2 accumulate

Prototype

Accumulate is an overloaded name; there are actually two accumulate functions.

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

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

Description

Accumulate is a generalization of summation: it computes the sum (or some other binary operation) of init and all of the elements in the range [first, last). The function object binary_op is not required to be either commutative or associative: the order of all of accumulate's operations is specified. The result is first initialized to init. Then, for each iterator i in [first, last), in order from beginning to end, it is updated by result = result + *i (in the first version) or result = binary_op(result, *i) (in the second version).

Definition

Defined in the standard header numeric, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version, the one that takes two arguments:

- InputIterator is a model of Input Iterator.
- T is a model of Assignable.

- If `x` is an object of type `T` and `y` is an object of `InputIterator`'s value type, then `x + y` is defined.
- The return type of `x + y` is convertible to `T`.

For the second version, the one that takes three arguments:

- `InputIterator` is a model of Input Iterator.
- `T` is a model of Assignable.
- `BinaryFunction` is a model of Binary Function.
- `T` is convertible to `BinaryFunction`'s first argument type.
- The value type of `InputIterator` is convertible to `BinaryFunction`'s second argument type.
- `BinaryFunction`'s return type is convertible to `T`.

Preconditions

- `[first, last)` is a valid range.

Complexity

Linear. Exactly `last - first` invocations of the binary operation.

Example

```
int main()
{
  int A[] = {1, 2, 3, 4, 5};
  const int N = sizeof(A) / sizeof(int);

  cout << "The sum of all elements in A is "
       << accumulate(A, A + N, 0)
       << endl;

  cout << "The product of all elements in A is "
       << accumulate(A, A + N, 1, multiplies<int>())
       << endl;
}
```

Notes

There are several reasons why it is important that `accumulate` starts with the value `init`. One of the most basic is that this allows `accumulate` to have a well-defined result even if `[first, last)` is an empty range: if it is empty, the return value is `init`. If you want to find the sum of all of the elements in `[first, last)`, you can just pass `0` as `init`.

See also

`inner_product`, `partial_sum`, `adjacent_difference`, `count`

9.4.3 inner_product

Prototype

`Inner_product` is an overloaded name; there are actually two `inner_product` functions.

```
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 BinaryFunction1, class BinaryFunction2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryFunction1 binary_op1,
                BinaryFunction2 binary_op2);
```

Description

`Inner_product` calculates a generalized inner product of the ranges `[first1, last1)` and `[first2, last2)`. The first version of `inner_product` returns `init` plus the inner product of the two ranges . That is, it first initializes the result to `init` and then, for each iterator `i` in `[first1, last1)`, in order from the beginning to the end of the range, updates the result by `result = result + (*i) * *(first2 + (i - first1))`. The second version of `inner_product` is identical to the first, except that it uses two user-supplied function objects instead of `operator+` and `operator*`. That is, it first initializes the result to `init` and then, for each iterator `i` in `[first1, last1)`, in order from the beginning to the end of the range, updates the result by `result = binary_op1(result, binary_op2(*i, *(first2 + (i - first1)))`.

Definition

Defined in the standard header numeric, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `T` is a model of Assignable.
- If `x` is an object of type `T`, `y` is an object of `InputIterator1`'s value type, and `z` is an object of `InputIterator2`'s value type, then `x + y * z` is defined.
- The type of `x + y * z` is convertible to `T`.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `T` is a model of Assignable.
- `BinaryFunction1` is a model of Binary Function.
- `BinaryFunction2` is a model of Binary Function.
- `InputIterator1`'s value type is convertible to `BinaryFunction2`'s first argument type.
- `InputIterator2`'s value type is convertible to `BinaryFunction2`'s second argument type.
- `T` is convertible to `BinaryFunction1`'s first argument type.
- `BinaryFunction2`'s return type is convertible to `BinaryFunction1`'s second argument type.
- `BinaryFunction1`'s return type is convertible to `T`.

Preconditions

- `[first1, last1)` is a valid range.
- `[first2, first2 + (last1 - first1))` is a valid range.

Complexity

Linear. Exactly `last1 - first1` applications of each binary operation.

Example

```
int main()
{
  int A1[] = {1, 2, 3};
  int A2[] = {4, 1, -2};
  const int N1 = sizeof(A1) / sizeof(int);

  cout << "The inner product of A1 and A2 is "
       << inner_product(A1, A1 + N1, A2, 0)
       << endl;
}
```

Notes

There are several reasons why it is important that `inner_product` starts with the value `init`. One of the most basic is that this allows `inner_product` to have a well-defined result even if `[first1, last1)` is an empty range: if it is empty, the return value is `init`. The ordinary inner product corresponds to setting `init` to 0. Neither binary operation is required to be either associative or commutative: the order of all operations is specified.

See also

`accumulate`, `partial_sum`, `adjacent_difference`, `count`

9.4.4 partial_sum

Prototype

`Partial_sum` is an overloaded name; there are actually two `partial_sum` functions.

```
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

`Partial_sum` calculates a generalized partial sum: `*first` is assigned to `*result`, the sum of `*first` and `*(first + 1)` is assigned to `*(result + 1)`, and so on. More precisely, a running sum is first initialized to `*first` and assigned to `*result`. For each iterator `i` in `[first + 1, last)`, in order from beginning to end, the sum is updated by `sum = sum + *i` (in the first version) or `sum = binary_op(sum, *i)` (in the second version) and is assigned to `*(result + (i - first))`.

Definition

Defined in the standard header numeric, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- If `x` and `y` are objects of `InputIterator`'s value type, then `x + y` is defined.
- The return type of `x + y` is convertible to `InputIterator`'s value type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `BinaryFunction` is a model of BinaryFunction.
- `InputIterator`'s value type is convertible to `BinaryFunction`'s first argument type and second argument type.
- `BinaryFunction`'s result type is convertible to `InputIterator`'s value type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, last)` is a valid range.
- `[result, result + (last - first))` is a valid range.

Complexity

Linear. Zero applications of the binary operation if `[first, last)` is a empty range, otherwise exactly `(last - first) - 1` applications.

Example

```
int main()
{
  const int N = 10;
  int A[N];

  fill(A, A+N, 1);
  cout << "A:                ";
  copy(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Partial sums of A: ";
  partial_sum(A, A+N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

Note that `result` is permitted to be the same iterator as `first`. This is useful for computing partial sums "in place". The binary operation is not required to be either associative or commutative: the order of all operations is specified.

See also

`adjacent_difference`, `accumulate`, `inner_product`, `count`

9.4.5 adjacent_difference

Prototype

`Adjacent_difference` is an overloaded name; there are actually two `adjacent_difference` functions.

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first,
                                   InputIterator last,
                                   OutputIterator result);

template <class InputIterator, class OutputIterator,
          class BinaryFunction>
OutputIterator adjacent_difference(InputIterator first,
                                   InputIterator last,
                                   OutputIterator result,
                                   BinaryFunction binary_op);
```

Description

`Adjacent_difference` calculates the differences of adjacent elements in the range `[first, last)`. This is, `*first` is assigned to `*result` , and, for each iterator `i` in the range `[first + 1, last)`, the difference of `*i` and `*(i - 1)` is assigned to `*(result + (i - first))`. The first version of `adjacent_difference` uses `operator-` to calculate differences, and the second version uses a user-supplied binary function. In the first version, for each iterator `i` in the range `[first + 1, last)`, `*i - *(i - 1)` is assigned to `*(result + (i - first))`. In the second version, the value that is assigned to `*(result + 1)` is instead `binary_op(*i, *(i - 1))`.

Definition

Defined in the standard header numeric, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `ForwardIterator` is a model of Forward Iterator.
- `OutputIterator` is a model of Output Iterator.
- If `x` and `y` are objects of `ForwardIterator`'s value type, then `x - y` is defined.
- `InputIterator`s value type is convertible to a type in `OutputIterator`'s set of value types.
- The return type of `x - y` is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `ForwardIterator` is a model of Forward Iterator.
- `OutputIterator` is a model of Output Iterator.
- `BinaryFunction` is a model of Binary Function.
- `InputIterator`'s value type is convertible to a `BinaryFunction`'s first argument type and second argument type.
- `InputIterators` value type is convertible to a type in `OutputIterator`'s set of value types.
- `BinaryFunction`'s result type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

- `[first, last)` is a valid range.
- `[result, result + (last - first))` is a valid range.

Complexity

Linear. Zero applications of the binary operation if `[first, last)` is an empty range, otherwise exactly `(last - first) - 1` applications.

Example

```
int main()
{
  int A[] = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100};
  const int N = sizeof(A) / sizeof(int);
  int B[N];

  cout << "A[]:         ";
  copy(A, A + N, ostream_iterator<int>(cout, " "));
  cout << endl;

  adjacent_difference(A, A + N, B);
  cout << "Differences: ";
  copy(B, B + N, ostream_iterator<int>(cout, " "));
  cout << endl;

  cout << "Reconstruct: ";
  partial_sum(B, B + N, ostream_iterator<int>(cout, " "));
  cout << endl;
}
```

Notes

The reason it is useful to store the value of the first element, as well as simply storing the differences, is that this provides enough information to reconstruct the input range. In particular, if addition and subtraction have the usual arithmetic definitions, then `adjacent_difference` and `partial_sum` are inverses of each other. Note that `result` is permitted to be the same iterator as `first`. This is useful for computing differences "in place".

See also

`partial_sum`, `accumulate`, `inner_product`, `count`

Chapter 10

Function Objects

10.1 Introduction

Summary

A *Function Object*, or *Functor* (the two terms are synonymous) is simply any object that can be called as if it is a function. An ordinary function is a function object, and so is a function pointer; more generally, so is an object of a class that defines `operator()`.

Description

The basic function object concepts are Generator, Unary Function, and Binary Function: these describe, respectively, objects that can be called as `f()`, `f(x)`, and `f(x,y)`. (This list could obviously be extended to *ternary function* and beyond, but, in practice, no STL algorithms require function objects of more than two arguments.) All other function object concepts defined by the STL are refinements of these three. Function objects that return `bool` are an important special case. A Unary Function whose return type is `bool` is called a Predicate, and a Binary Function whose return type is `bool` is called a Binary Predicate. There is an important distinction, but a somewhat subtle one, between function objects and *adaptable function objects*. In general, a function object has restrictions on the type of its argument. The type restrictions need not be simple, though: `operator()` may be overloaded, or may be a member template, or both. Similarly, there need be no way for a program to determine what those restrictions are. An adaptable function object, however, does specify what the argument and return types are, and provides nested `typedef`s so that those types can be named and used in programs. If a type `F0` is a model of Adaptable Generator, then it must define `F0::result_type`. Similarly, if `F1` is a model of Adaptable Unary Function then it must define `F1::argument_type` and `F1::result_type`, and if `F2` is a model of Adaptable Binary Function then it must define `F2::first_argument_type`, `F2::second_argument_type`, and `F2::result_type`. The STL provides base classes `unary_function` and `binary_function` to simplify the definition of Adaptable Unary Functions and Adaptable Binary Functions. Adaptable function objects are important because they can be used by *function object adaptors*: function objects that transform or manipulate other function objects. The STL provides many different function object adaptors, including `unary_negate` (which returns the logical complement of the value returned by a particular AdaptablePredicate), and `unary_compose` and `binary_compose`, which perform composition of function object. Finally, the STL includes many different predefined function objects, including arithmetic operations (`plus`, `minus`, `multiplies`, `divides`, `modulus`, and `negate`), comparisons (`equal_to`, `not_equal_to` `greater`, `less`, `greater_equal`, and `less_equal`), and logical operations (`logical_and`, `logical_or`, and `logical_not`). It is possible to perform very sophisticated operations without actually writing a new function object, simply by combining predefined function objects and function object adaptors.

Examples

Fill a `vector` with random numbers. In this example, the function object is simply a function pointer.

```
vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

Sort a `vector` of `double` by magnitude, *i.e.* ignoring the elements' signs. In this example, the function object is an object of a user-defined class.

```
struct less_mag : public binary_function<double, double, bool> {
   bool operator()(double x, double y) { return fabs(x) < fabs(y); }
};

vector<double> V;
...
sort(V.begin(), V.end(), less_mag());
```

Find the sum of elements in a `vector`. In this example, the function object is of a user-defined class that has local state.

```
struct adder : public unary_function<double, void>
{
   adder() : sum(0) {}
   double sum;
   void operator()(double x) { sum += x; }
};

vector<double> V;
...
adder result = for_each(V.begin(), V.end(), adder()); [3]
cout << "The sum is " << result.sum << endl;
```

Remove all elements from a `list` that are greater than 100 and less than 1000.

```
list<int> L;
...
list<int>::iterator new_end =
   remove_if(L.begin(), L.end(),
             compose2(logical_and<bool>(),
                      bind2nd(greater<int>(), 100),
                      bind2nd(less<int>(), 1000)));
L.erase(new_end, L.end());
```

Concepts

- Generator
- Unary Function
- Binary Function

- Predicate
- Binary Predicate

- Adaptable Generator
- Adaptable Unary Function
- Adaptable Binary Function
- Adaptable Predicate
- Adaptable Binary Predicate

Types

- `plus`
- `minus`
- `multiplies` (formerly called `times`)
- `divides`
- `modulus`,
- `negate`
- `equal_to`
- `not_equal_to`
- `greater`
- `less`
- `greater_equal`
- `less_equal`,
- `logical_and`
- `logical_or`
- `logical_not`
- `subtractive_rng`

- `identity`
- `project1st`
- `project2nd`
- `select1st`
- `select2nd`

- `unary_function`
- `binary_function`

- `unary_compose`
- `binary_compose`
- `unary_negate`

- `binary_negate`
- `binder1st`
- `binder2nd`
- `pointer_to_unary_function`
- `pointer_to_binary_function`

Functions

- `compose1`
- `compose2`
- `not1`
- `not2`
- `bind1st`
- `bind2nd`
- `ptr_fun`

Notes

The reason for the name "adaptable function object" is that adaptable function objects may be used by function object adaptors. The `unary_function` and `binary_function` bases are similar to the `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, and `random_access_iterator` bases: they are completely empty, and serve only to provide type information. This is an example of how to use function objects; it is not the recommended way of calculating the sum of elements in a vector. The `accumulate` algorithm is a better way of calculating a sum.

See also

10.2 Concepts

10.2.1 Generator

Description

A Generator is a kind of function object: an object that is called as if it were an ordinary C++ function. A Generator is called with no arguments.

Refinement of

Assignable

Associated types

Result type	The type returned when the Generator is called

Notation

`F`	A type that is a model of Generator
`Result`	The result type of `F`
`f`	Object of type `F`

Definitions

The *range* of a Generator is the set of all possible value that it may return.

Valid expressions

Name	Expression	Type reqs	Return type
Function call	`f()`		`Result`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f()`		Returns some value of type `Result`	The return value is in `f`'s range.

Complexity guarantees

Invariants

Models

- `Result (*)()`

Notes

Two different invocations of `f` may return different results: a Generator may refer to local state, perform I/O, and so on. The expression `f()` is permitted to change `f`'s state; `f` might, for example, represent a pseudo-random number generator.

See also

Function Object overview, Unary Function, Binary Function, Adaptable Generator

10.2.2 Unary Function

Description

A Unary Function is a kind of function object: an object that is called as if it were an ordinary C++ function. A Unary Function is called with a single argument.

Refinement of

Assignable

Associated types

Argument type	The type of the Unary Function's argument.
Result type	The type returned when the Unary Function is called

Notation

`F`	A type that is a model of Unary Function
`X`	The argument type of `F`
`Result`	The result type of `F`
`f`	Object of type `F`
`x`	Object of type `X`

Definitions

The *domain* of a Unary Function is the set of all permissible values for its argument. The *range* of a Unary Function is the set of all possible values that it may return.

Valid expressions

Name	Expression	Type reqs	Return type
Function call	`f(x)`		`Result`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(x)`	`x` is in `f`'s domain	Calls `f` with `x` as an argument, and returns a value of type `Result`	The return value is in `f`'s range

Complexity guarantees

Invariants

Models

- `Result (*)(X)`

Notes

Two different invocations of `f` may return different results, even if `f` is called with the same arguments both times. A Unary Function may refer to local state, perform I/O, and so on. The expression `f(x)` is permitted to change `f`'s state.

See also

Function Object overview, Generator, Binary Function Adaptable Unary Function

10.2.3 Binary Function

Description

A Binary Function is a kind of function object: an object that is called as if it were an ordinary C++ function. A Binary Function is called with two arguments.

Refinement of

Assignable

Associated types

First argument type	The type of the Binary Function's first argument.
Second argument type	The type of the Binary Function's second argument.
Result type	The type returned when the Binary Function is called

Notation

`F`	A type that is a model of BinaryFunction
`X`	The first argument type of `F`
`Y`	The second argument type of `F`
`Result`	The result type of `F`
`f`	Object of type `F`
`x`	Object of type `X`
`y`	Object of type `Y`

Definitions

The *domain* of a Binary Function is the set of all ordered pairs `(x, y)` that are permissible values for its arguments. The *range* of a Binary Function is the set of all possible value that it may return.

Valid expressions

Name	Expression	Type reqs	Return type
Function call	`f(x,y)`		`Result`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(x,y)`	The ordered pair `(x,y)` is in `f`'s domain	Calls `f` with `x` and `y` as arguments, and returns a value of type `Result`	The return value is in `f`'s range

Complexity guarantees

Invariants

Models

- `Result (*)(X,Y)`

Notes

Two different invocations of `f` may return different results, even if `f` is called with the same arguments both times. A Binary Function may refer to local state, perform I/O, and so on. The expression `f(x,y)` is permitted to change `f`'s state.

See also

Function Object overview, Generator, Unary Function Adaptable Binary Function

10.2.4 Adaptable Generator

Description

An Adaptable Generator is a Generator with a nested `typedef` that defines its result type. This nested `typedef` makes it possible to use function object adaptors.

Refinement of

Generator

Associated types

Result type	`F::result_type`	The type returned when the Generator is called

Notation

`F`	A type that is a model of Adaptable Generator

Definitions

Valid expressions

None, except for those defined by Generator

Expression semantics

Complexity guarantees

Invariants

Models

The STL does not include any types that are models of Adaptable Generator. An example of a user-defined Adaptable Generator is as follows.

```
struct counter
{
  typedef int result_type;

  counter() : n(0) {}
  result_type operator()() { return n++; }

  result_type n;
};
```

Notes

Note the implication of this: a function pointer `T (*f)()` is a Generator, but not an Adaptable Generator: the expression `f::result_type` is nonsensical.

See also

Generator, Adaptable Unary Function, Adaptable Binary Function

10.2.5 Adaptable Unary Function

Description

An Adaptable Unary Function is a Unary Function with nested `typedef`s that define its argument type and result type. These nested `typedef` make it possible to use function object adaptors.

Refinement of

Unary Function

Associated types

Argument type	`F::argument_type`	The type of `F`'s argument
Result type	`F::result_type`	The type returned when the Unary Function is called

Notation

`F`	A type that is a model of Unary Function

Definitions

Valid expressions

None, except for those defined by Unary Function

Expression semantics

Complexity guarantees

Invariants

Models

- `negate`
- `identity`
- `pointer_to_unary_function`

Notes

Note the implication of this: a function pointer `T (*f)(X)` is a Unary Function, but not an Adaptable Unary Function: the expressions `f::argument_type` and `f::result_type` are nonsensical. When you define a class that is a model of Adaptable Unary Function, you must provide these `typedef`s. The easiest way to do this is to derive the class from the base class `unary_function`. This is an empty class, with no member functions or member variables; the only reason it exists is to make defining Adaptable Unary Functions more convenient. `Unary_function` is very similar to the base classes used by the iterator tag functions.

See also

Unary Function, Adaptable Generator, Adaptable Binary Function

10.2.6 Adaptable Binary Function

Description

An Adaptable Binary Function is a Binary Function with nested `typedef`s that define its argument types and result type. These nested `typedef`s make it possible to use function object adaptors.

Refinement of

Binary Function

Associated types

First argument type	`F::first_argument_type`	The type of `F`'s first argument
Second argument type	`F::second_argument_type`	The type of `F`'s second argument
Result type	`F::result_type`	The type returned when the Binary Function is called

Notation

`F`	A type that is a model of Binary Function

Definitions

Valid expressions

None, except for those defined by Binary Function

Expression semantics

Complexity guarantees

Invariants

Models

- `plus`
- `project1st`
- `pointer_to_binary_function`

Notes

Note the implication of this: a function pointer `T (*f)(X,Y)` is a Binary Function, but not an Adaptable Binary Function: the expressions `f::first_argument_type`, `f::second_argument_type`, and `f::result_type` are nonsensical. When you define a class that is a model of Adaptable Binary Function, you must provide these `typedef`s. The easiest way to do this is to derive the class from the base class `binary_function`. This is an empty class, with no member functions or member variables; the only reason it exists is to make defining Adaptable Binary Functions more convenient. `Binary_function` is very similar to the base classes used by the iterator tag functions.

See also

Binary Function, Adaptable Generator, Adaptable Unary Function

10.2.7 Predicates

Predicate

Description

A Predicate is a Unary Function whose result represents the truth or falsehood of some condition. A Predicate might, for example, be a function that takes an argument of type `int` and returns `true` if the argument is positive.

Refinement of

Unary Function

Associated types

Result type	The type returned when the Predicate is called. The result type must be convertible to `bool`.

Notation

`F`	A type that is a model of Predicate
`X`	The argument type of `F`
`f`	Object of type `F`
`x`	Object of type `X`

Valid expressions

Name	Expression	Type reqs	Return type
Function call	`f(x)`		Convertible to `bool`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(x)`	`x` is in the domain of `f`.	Returns `true` if the condition is satisfied, `false` if it is not.	The result is either `true` or `false`.

Complexity guarantees

Invariants

Models

- `bool (*)(int)`

Notes

See also

Adaptable Predicate, Binary Predicate, Adaptable Binary Predicate

Binary Predicate

Description

A Binary Predicate is a Binary Function whose result represents the truth or falsehood of some condition. A Binary Predicate might, for example, be a function that takes two arguments and tests whether they are equal.

Refinement of

Binary Function

Associated types

Result type	The type returned when the Binary Predicate is called. The result type must be convertible to `bool`.

Notation

`F`	A type that is a model of Binary Predicate
`X`	The first argument type of `F`
`Y`	The second argument type of `F`
`f`	Object of type `F`
`x`	Object of type `X`
`y`	Object of type `Y`

Valid expressions

Name	Expression	Type reqs	Return type
Function call	`f(x,y)`		Convertible to `bool`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(x,y)`	The ordered pair `(x,y)` is in the domain of `f`.	Returns `true` if the condition is satisfied, `false` if it is not.	The result is either `true` or `false`.

Complexity guarantees

Invariants

Models

- `bool (*)(int,int)`
- `equal_to`

Notes

See also

Predicate, Adaptable Predicate, Adaptable Binary Predicate

Adaptable Predicate

Description

An Adaptable Predicate is a Predicate that is also an Adaptable Unary Function. That is, it is a Unary Function whose return type is `bool`, and that includes nested `typedef`s that define its argument type and return type.

Refinement of

Predicate, Adaptable Unary Function

Associated types

None, except for those associated with Predicate and Adaptable Unary Function.

Notation

Definitions

Valid expressions

None, except for those defined by the Predicate and Adaptable Unary Function requirements.

Expression semantics

Complexity guarantees

Invariants

Models

- `logical_not`
- `unary_negate`

Notes

See also

Predicate, Binary Predicate, Adaptable Binary Predicate

Adaptable Binary Predicate

Description

An Adaptable Binary Predicate is a Binary Predicate that is also an Adaptable Binary Function. That is, it is a Binary Function whose return type is `bool`, and that includes nested `typedef`s that define its argument types and return type.

Refinement of

Predicate, Adaptable Binary Function

Associated types

None, except for those associated with Predicate and Adaptable Binary Function.

Notation

Definitions

Valid expressions

None, except for those defined by the Predicate and Adaptable Binary Function requirements.

Expression semantics

Complexity guarantees

Invariants

Models

- `less`
- `equal_to`
- `logical_and`
- `logical_or`
- `binary_negate`

Notes

See also

Binary Predicate, Predicate, Adaptable Predicate

StrictWeakOrdering

Description

A Strict Weak Ordering is a Binary Predicate that compares two objects, returning `true` if the first precedes the second. This predicate must satisfy the standard mathematical definition of a *strict weak ordering*. The precise requirements are stated below, but what they roughly mean is that a Strict Weak Ordering has to behave the way that "less than" behaves: if `a` is less than `b` then `b` is not less than `a`, if `a` is less than `b` and `b` is less than `c` then `a` is less than `c`, and so on.

Refinement of

Binary Predicate

Associated types

First argument type	The type of the Strict Weak Ordering's first argument.
Second argument type	The type of the Strict Weak Ordering's second argument. The first argument type and second argument type must be the same.
Result type	The type returned when the Strict Weak Ordering is called. The result type must be convertible to `bool`.

Notation

`F`	A type that is a model of Strict Weak Ordering
`X`	The type of Strict Weak Ordering's arguments.
`f`	Object of type `F`
`x`, `y`, `z`	Object of type `X`

Definitions

- Two objects `x` and `y` are *equivalent* if both `f(x, y)` and `f(y, x)` are false. Note that an object is always (by the irreflexivity invariant) equivalent to itself.

Valid expressions

None, except for those defined in the Binary Predicate requirements.

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(x, y)`	The ordered pair `(x,y)` is in the domain of `f`	Returns `true` if `x` precedes `y`, and `false` otherwise	The result is either `true` or `false`

Complexity guarantees

Invariants

Irreflexivity	`f(x, x)` must be `false`.
Antisymmetry	`f(x, y)` implies `!f(y, x)`
Transitivity	`f(x, y)` and `f(y, z)` imply `f(x, z)`.
Transitivity of equivalence	Equivalence (as defined above) is transitive: if `x` is equivalent to `y` and `y` is equivalent to `z`, then `x` is equivalent to `z`. (This implies that equivalence does in fact satisfy the mathematical definition of an equivalence relation.)

Models

- `less<int>`
- `less<double>`
- `greater<int>`
- `greater<double>`

Notes

The first three axioms, irreflexivity, antisymmetry, and transitivity, are the definition of a *partial ordering*; transitivity of equivalence is required by the definition of a *strict weak ordering*. A *total ordering* is one that satisfies an even stronger condition: equivalence must be the same as equality.

See also

LessThan Comparable, `less`, Binary Predicate, function objects

10.2.8 Random Number Generator

Description

A Random Number Generator is a function object that can be used to generate a random sequence of integers. That is: if `f` is a Random Number Generator and `N` is a positive integer, then `f(N)` will return an integer less than `N` and greater than or equal to `0`. If `f` is called many times with the same value of `N`, it will yield a sequence of numbers that is uniformly distributed in the range `[0, N)`.

Refinement of

Unary Function

Associated types

Argument type	The type of the Random Number Generator's argument. This must be an integral type.
Result type	The type returned when the Random Number Generator is called. It must be the same as the argument type.

Notation

`F`	A type that is a model of Random Number Generator.
`Integer`	The argument type of `F`.
`f`	Object of type `F`.
`N`	Object of type `Integer`

Definitions

The *domain* of a Random Number Generator (*i.e.* the set of permissible values for its argument) is the set of numbers that are greater than zero and less than some maximum value. The *range* of a Random Number Generator is the set of nonnegative integers that are less than the Random Number Generator's argument.

Valid expressions

None, except for those defined by Unary Function.

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Function call	`f(N)`	`N` is positive.	Returns a pseudo-random number of type `Integer`.	The return value is less than `N`, and greater than or equal to 0.

Complexity guarantees

Invariants

Uniformity	In the limit as `f` is called many times with the same argument `N`, every integer in the range `[0, N)` will appear an equal number of times.

Models

Notes

Uniform distribution means that all of the numbers in the range `[0, N)` appear with equal frequency. Or, to put it differently, the probability for obtaining any particular value is `1/N`. Random number generators are a very subtle subject: a good random number generator must satisfy many statistical properties beyond uniform distribution. See section 3.4 of Knuth for a discussion of what it means for a sequence to be random, and section 3.2 for several algorithms that may be used to write random number generators. (D. E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, third edition. Addison-Wesley, 1998.)

See also

10.3 Predefined function objects

10.3.1 Arithmetic operations

plus

Description

`Plus<T>` is a function object. Specifically, it is an Adaptable Binary Function. If `f` is an object of class `plus<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `x+y`.

Example

Each element in `V3` will be the sum of the corresponding elements in `V1` and `V2`

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);
vector<double> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          plus<double>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Binary Function, Default Constructible

Type requirements

T must be a numeric type; if x and y are objects of type T, then x+y must be defined and must have a return type that is convertible to T. T must be Assignable.

Public base classes

`binary_function<T, T, T>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `T`
`T operator()(const T& x, const T& y)`	Adaptable Binary Function	Function call operator. The return value is `x + y`.
`plus()`	Default Constructible	The default constructor.

New members

All of `plus`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Plus` does not introduce any new members.

Notes

See also

The Function Object overview, Adaptable Binary Function, `binary_function`, `minus`, `multiplies`, `divides`, `modulus`, `negate`

minus

Description

`Minus<T>` is a function object. Specifically, it is an Adaptable Binary Function. If `f` is an object of class `minus<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `x-y`.

Example

Each element in `V3` will be the difference of the corresponding elements in `V1` and `V2`

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);
vector<double> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          minus<double>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Binary Function, Default Constructible

Type requirements

`T` must be a numeric type; if `x` and `y` are objects of type `T`, then `x-y` must be defined and must have a return type that is convertible to `T`. `T` must be Assignable.

Public base classes

`binary_function<T, T, T>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `T`
`T operator()(const T& x, const T& y)`	Adaptable Binary Function	Function call operator. The return value is `x - y`.
`minus()`	Default Constructible	The default constructor.

New members

All of `minus`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Minus` does not introduce any new members.

Notes

See also

The Function Object overview, Adaptable Binary Function, `binary_function`, `plus`, `multiplies`, `divides`, `modulus`, `negate`

multiplies

Description

`Multiplies<T>` is a function object. Specifically, it is an Adaptable Binary Function. If `f` is an object of class `multiplies<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `x*y`.

Example

Each element in `V3` will be the product of the corresponding elements in `V1` and `V2`

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);
vector<double> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          multiplies<double>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Binary Function, Default Constructible

Type requirements

`T` must be a numeric type; if `x` and `y` are objects of type `T`, then `x*y` must be defined and must have a return type that is convertible to `T`. `T` must be Assignable.

Public base classes

`binary_function<T, T, T>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `T`
`T operator()(const T& x, const T& y)`	Adaptable Binary Function	Function call operator. The return value is `x * y`.
`multiplies()`	Default Constructible	The default constructor.

New members

All of `multiplies`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Multiplies` does not introduce any new members.

Notes

Warning: the name of this function object has been changed from `times` to `multiplies`. The name was changed for two reasons. First, it is called `multiplies` in the C++ standard. Second, the name `times` conflicts with a function in the Unix header `<sys/times.h>`.

See also

The Function Object overview, Adaptable Binary Function, `binary_function`, `plus`, `minus`, `divides`, `modulus`, `negate`

divides

Description

`Divides<T>` is a function object. Specifically, it is an Adaptable Binary Function. If `f` is an object of class `divides<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `x/y`.

Example

Each element in `V3` will be the quotient of the corresponding elements in `V1` and `V2`

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);
vector<double> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          divides<double>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Binary Function, Default Constructible

Type requirements

`T` must be a numeric type; if `x` and `y` are objects of type `T`, then `x/y` must be defined and must have a return type that is convertible to `T`. `T` must be Assignable.

Public base classes

`binary_function<T, T, T>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `T`
`T operator()(const T& x, const T& y)`	Adaptable Binary Function	Function call operator. The return value is `x / y`.
`divides()`	Default Constructible	The default constructor.

New members

All of `divides`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Divides` does not introduce any new members.

Notes

See also

The Function Object overview, Adaptable Binary Function, `binary_function`, `plus`, `minus`, `multiplies`, `modulus`, `negate`

modulus

Description

`Modulus<T>` is a function object. Specifically, it is an Adaptable Binary Function. If `f` is an object of class `modulus<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `x%y`.

Example

Each element in `V3` will be the modulus of the corresponding elements in `V1` and `V2`

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);
vector<double> V3(N);

iota(V1.begin(), V1.end(), 1);
fill(V2.begin(), V2.end(), 75);

assert(V2.size() >= V1.size() && V3.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(), V3.begin(),
          modulus<int>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Binary Function, Default Constructible

Type requirements

`T` must be an integral type; if `x` and `y` are objects of type `T`, then `x%y` must be defined and must have a return type that is convertible to `T`. `T` must be Assignable.

Public base classes

`binary_function<T, T, T>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `T`
`T operator()(const T& x, const T& y)`	Adaptable Binary Function	Function call operator. The return value is `x % y`.
`modulus()`	Default Constructible	The default constructor.

New members

All of `modulus`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Modulus` does not introduce any new members.

Notes

See also

The Function Object overview, Adaptable Binary Function, `binary_function`, `plus`, `minus`, `multiplies`, `divides`, `negate`

negate

Description

`Negate<T>` is a function object. Specifically, it is an Adaptable Unary Function. If `f` is an object of class `negate<T>` and `x` is an object of class `T`, then `f(x)` returns `-x`.

Example

Each element in `V2` will be the negative (additive inverse) of the corresponding element in `V1`.

```
const int N = 1000;
vector<double> V1(N);
vector<double> V2(N);

iota(V1.begin(), V1.end(), 1);

assert(V2.size() >= V1.size());
transform(V1.begin(), V1.end(), V2.begin(),
          negate<int>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The function object's argument type and result type.	

Model of

Adaptable Unary Function, Default Constructible

Type requirements

`T` must be a numeric type; if `x` is an object of type `T`, then `-x` must be defined and must have a return type that is convertible to `T`. `T` must be Assignable.

Public base classes

`unary_function<T, T>`

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the second argument: `T`
`result_type`	Adaptable Unary Function	The type of the result: `T`
`T operator()(const T& x)`	Adaptable Unary Function	Function call operator. The return value is `-x`.
`negate()`	Default Constructible	The default constructor.

New members

All of `negate`'s members are defined in the Adaptable Unary Function and Default Constructible requirements. `Negate` does not introduce any new members.

Notes

See also

The Function Object overview, Adaptable Unary Function, `unary_function`, `plus`, `minus`, `multiplies`, `divides`, `modulus`

10.3.2 Comparisons

equal_to

Description

`Equal_to<T>` is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `equal_to<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `true` if `x == y` and `false` otherwise.

Example

Rearrange a vector such that all of the elements that are equal to zero precede all nonzero elements.

```
vector<int> V;
...
partition(V.begin(), V.end(), bind2nd(equal_to<int>(), 0));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `equal_to`'s arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` is EqualityComparable.

Public base classes

`binary_function<T, T, bool>`.

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Predicate	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Predicate	The type of the second argument: `T`
`result_type`	Adaptable Binary Predicate	The type of the result: `bool`
`equal_to()`	Default Constructible	The default constructor.
`bool operator()(const T& x, const T& y)`	Binary Function	Function call operator. The return value is `x == y`.

New members

All of `equal_to`'s members are defined in the Adaptable Binary Predicate and DefaultConstructible requirements. `Equal_to` does not introduce any new members.

Notes

See also

The function object overview, Adaptable Binary Predicate, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`

not_equal_to

Description

`Not_equal_to<T>` is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `not_equal_to<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `true` if `x != y` and `false` otherwise.

Example

Finds the first nonzero element in a list.

```
list<int> L;
...
list<int>::iterator first_nonzero =
       find_if(L.begin(), L.end(), bind2nd(not_equal_to<int>(), 0));
assert(first_nonzero == L.end() || *first_nonzero != 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `not_equal_to`'s arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` is EqualityComparable.

Public base classes

`binary_function<T, T, bool>`.

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Predicate	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Predicate	The type of the second argument: `T`
`result_type`	Adaptable Binary Predicate	The type of the result: `bool`
`not_equal_to()`	Default Constructible	The default constructor.
`bool operator()(const T& x, const T& y)`	Binary Function	Function call operator. The return value is `x != y`.

New members

All of `not_equal_to`'s members are defined in the Adaptable Binary Predicate and DefaultConstructible requirements. `Not_equal_to` does not introduce any new members.

Notes

See also

The function object overview, Adaptable Binary Predicate, `equal_to`, `greater`, `less`, `greater_equal`, `less_equal`

less

Description

`Less<T>` is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `less<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `true` if `x < y` and `false` otherwise.

Example

Finds the first negative element in a list.

```
list<int> L;
...
list<int>::iterator first_negative =
       find_if(L.begin(), L.end(), bind2nd(less<int>(), 0));
assert(first_negative == L.end() || *first_negative < 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `less`'s arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` is LessThan Comparable.

Public base classes

`binary_function<T, T, bool>`.

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Predicate	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Predicate	The type of the second argument: `T`
`result_type`	Adaptable Binary Predicate	The type of the result: `bool`
`less()`	Default Constructible	The default constructor.
`bool operator()(const T& x, const T& y)`	Binary Function	Function call operator. The return value is `x < y`.

New members

All of `less`'s members are defined in the Adaptable Binary Predicate and Default-Constructible requirements. `less` does not introduce any new members.

Notes

See also

The function object overview, Strict Weak Ordering, Adaptable Binary Predicate, LessThan Comparable, `equal_to`, `not_equal_to`, `greater`, `greater_equal`, `less_equal`

greater

Description

`Greater<T>` is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `greater<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `true` if `x > y` and `false` otherwise.

Example

Sort a vector in descending order, rather than the default ascending order.

```
vector<int> V;
...
sort(V.begin(), V.end(), greater<int>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `greater`'s arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` is LessThan Comparable.

Public base classes

`binary_function<T, T, bool>`.

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Predicate	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Predicate	The type of the second argument: `T`
`result_type`	Adaptable Binary Predicate	The type of the result: `bool`
`greater()`	Default Constructible	The default constructor.
`bool operator()(const T& x, const T& y)`	Binary Function	Function call operator. The return value is `x > y`.

New members

All of `greater`'s members are defined in the Adaptable Binary Predicate and DefaultConstructible requirements. `Greater` does not introduce any new members.

Notes

See also

The function object overview, Adaptable Binary Predicate, LessThan Comparable, `equal_to`, `not_equal_to`, `less`, `greater_equal`, `less_equal`

less_equal

Description

`Less_equal<T>` is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `less_equal<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns `true` if `x <= y` and `false` otherwise.

Example

Finds the first non-positive element in a list.

```
list<int> L;
...
list<int>::iterator first_nonpositive =
       find_if(L.begin(), L.end(), bind2nd(less_equal<int>(), 0));
assert(first_nonpositive == L.end() || *first_nonpositive <= 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `less_equal`'s arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

T is LessThan Comparable.

Public base classes

binary_function<T, T, bool>.

Members

Member	Where defined	Description
first_argument_type	Adaptable Binary Predicate	The type of the first argument: T
second_argument_type	Adaptable Binary Predicate	The type of the second argument: T
result_type	Adaptable Binary Predicate	The type of the result: bool
less_equal()	Default Constructible	The default constructor.
bool operator()(const T& x, const T& y)	Binary Function	Function call operator. The return value is x <= y.

New members

All of less_equal's members are defined in the Adaptable Binary Predicate and DefaultConstructible requirements. Less_equal does not introduce any new members.

Notes

See also

The function object overview, Adaptable Binary Predicate, equal_to, not_equal_to, greater, less, greater_equal,

greater_equal

Description

Greater_equal<T> is a function object. Specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If f is an object of class greater_equal<T> and x and y are objects of class T, then f(x,y) returns true if x >= y and false otherwise.

Example

Find the first nonnegative element in a list.

```
list<int> L;
...
list<int>::iterator first_nonnegative =
    find_if(L.begin(), L.end(), bind2nd(greater_equal<int>(), 0));
assert(first_nonnegative == L.end() || *first_nonnegative >= 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
T	The type of greater_equal's arguments.	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

T is LessThan Comparable.

Public base classes

binary_function<T, T, bool>.

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Predicate	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Predicate	The type of the second argument: `T`
`result_type`	Adaptable Binary Predicate	The type of the result: `bool`
`greater_equal()`	Default Constructible	The default constructor.
`bool operator()(const T& x, const T& y)`	Binary Function	Function call operator. The return value is `x >= y`.

New members

All of `greater_equal`'s members are defined in the Adaptable Binary Predicate and DefaultConstructible requirements. `Greater_equal` does not introduce any new members.

Notes

See also

The function object overview, Adaptable Binary Predicate, `equal_to`, `not_equal_to`, `greater` `less`, `less_equal`

10.3.3 Logical operations

logical_and

Description

`Logical_and<T>` is a function object; specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `logical_and<T>` and `x` and `y` are objects of class `T` (where `T` is convertible to `bool`) then `f(x,y)` returns true if and only if both `x` and `y` are `true`.

Example

Finds the first element in a list that lies in the range from 1 to 10.

```
list<int> L;
...
list<int>::iterator in_range =
     find_if(L.begin(), L.end(),
             compose2(logical_and<bool>(),
                      bind2nd(greater_equal<int>(), 1),
                      bind2nd(less_equal<int>(), 10)));
assert(in_range == L.end() || (*in_range >= 1 && *in_range <= 10));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `logical_and`'s arguments	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` must be convertible to `bool`.

Public base classes

`binary_function<T, T, bool>`

Members

Member	Where defined	Description
first_argument_type	Adaptable Binary Function	The type of the first argument: T
second_argument_type	Adaptable Binary Function	The type of the second argument: T
result_type	Adaptable Binary Function	The type of the result: bool
bool operator()(const T& x, const T& y) const	Binary Function	Function call operator. The return value is x && y.
logical_and()	Default Constructible	The default constructor.

New members

All of logical_and's members are defined in the Adaptable Binary Function and Default Constructible requirements. Logical_and does not introduce any new members.

Notes

Logical_and and logical_or are not very useful by themselves. They are mainly useful because, when combined with the function object adaptor binary_compose, they perform logical operations on other function objects.

See also

The function object overview, logical_or, logical_not.

logical_or

Description

Logical_or<T> is a function object; specifically, it is an Adaptable Binary Predicate, which means it is a function object that tests the truth or falsehood of some condition. If f is an object of class logical_and<T> and x and y are objects of class T (where T is convertible to bool) then f(x,y) returns true if and only if either x or y is true.

Example

Finds the first instance of either ' ' or '\n' in a string.

```
char str[MAXLEN];
...
const char* wptr = find_if(str, str + MAXLEN,
                           compose2(logical_or<bool>(),
                                    bind2nd(equal_to<char>(), ' '),
                                    bind2nd(equal_to<char>(), '\n')));
assert(wptr == str + MAXLEN || *wptr == ' ' || *wptr == '\n');
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `logical_or`'s arguments	

Model of

Adaptable Binary Predicate, DefaultConstructible

Type requirements

`T` must be convertible to `bool`.

Public base classes

`binary_function<T, T, bool>`

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `T`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `T`
`result_type`	Adaptable Binary Function	The type of the result: `bool`
`bool operator()(const T& x, const T& y) const`	Binary Function	Function call operator. The return value is `x || y`.
`logical_or()`	Default Constructible	The default constructor.

New members

All of `logical_or`'s members are defined in the Adaptable Binary Function and Default Constructible requirements. `Logical_or` does not introduce any new members.

Notes

`Logical_and` and `logical_or` are not very useful by themselves. They are mainly useful because, when combined with the function object adaptor `binary_compose`, they perform logical operations on other function objects.

See also

The function object overview, `logical_and`, `logical_not`.

logical_not

Description

`Logical_not<T>` is a function object; specifically, it is an Adaptable Predicate, which means it is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `logical_not<T>` and `x` is an object of class `T` (where `T` is convertible to `bool`) then `f(x)` returns true if and only if `x` is `false`.

Example

Transforms a vector of `bool` into its logical complement.

```
vector<bool> V;
...
transform(V.begin(), V.end(), V.begin(), logical_not<bool>());
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`T`	The type of `logical_not`'s argument	

Model of

Adaptable Predicate, DefaultConstructible

Type requirements

`T` must be convertible to `bool`.

Public base classes

`unary_function<T, bool>`

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the second argument: `T`
`result_type`	Adaptable Unary Function	The type of the result: `bool`
`bool operator()(const T& x) const`	Unary Function	Function call operator. The return value is `!x`.
`logical_not()`	Default Constructible	The default constructor.

Notes

See also

The function object overview, `logical_or`, `logical_and`.

10.4 Function object adaptors

10.4.1 binder1st

Description

`Binder1st` is a function object adaptor: it is used to transform an adaptable binary function into an adaptable unary function. Specifically, if `f` is an object of class `binder1st<AdaptableBinaryFunction>`, then `f(x)` returns `F(c, x)`, where `F` is an object of class `AdaptableBinaryFunction` and where `c` is a constant. Both `F` and `c` are passed as arguments to `binder1st`'s constructor. The easiest way to create a `binder1st` is not to call the constructor explicitly, but instead to use the helper function `bind1st`.

Example

Finds the first nonzero element in a list.

```
list<int> L;
...
list<int>::iterator first_nonzero =
       find_if(L.begin(), L.end(), bind1st(not_equal_to<int>(), 0));
assert(first_nonzero == L.end() || *first_nonzero != 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`AdaptableBinaryFunction`	The type of the binary function whose first argument is being bound to a constant.	

Model of

Adaptable Unary Function

Type requirements

`AdaptableBinaryFunction` must be a model of Adaptable Binary Function.

Public base classes

```
unary_function<AdaptableBinaryFunction::second_argument_type,
               AdaptableBinaryFunction::result_type>
```

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the function object's argument, which is `AdaptableBinaryFunction::second_argument_type`
`result_type`	Adaptable Unary Function	The type of the result: `AdaptableBinaryFunction::result_type`
`result_type operator()(const argument_type& x) const`	Adaptable Unary Function	Function call. Returns `F(c, x)`, where `F` and `c` are the arguments with which this `binder1st` was constructed.
`binder1st(const AdaptableBinaryFunction& F, AdaptableBinaryFunction::first_argument_type c)`	`binder1st`	See below
`template <class AdaptableBinaryFunction, class T> binder1st <AdaptableBinaryFunction> bind1st(const AdaptableBinaryFunction& F, const T& c);`	`binder1st`	See below

New members

These members are not defined in the Adaptable Unary Function requirements, but are specific to `binder1st`.

Member	Description
`binder1st(const AdaptableBinaryFunction& F, AdaptableBinaryFunction::first_argument_type c)`	The constructor. Creates a `binder1st` such that calling it with the argument `x` (where `x` is of type `AdaptableBinaryFunction::second_argument_type`) corresponds to the call `F(c, x)`.
`template <class AdaptableBinaryFunction, class T> binder1st <AdaptableBinaryFunction> bind1st(const AdaptableBinaryFunction& F, const T& c);`	If `F` is an object of type `AdaptableBinaryFunction`, then `bind1st(F, c)` is equivalent to `binder1st<AdaptableBinaryFunction>(F, c)`, but is more convenient. The type `T` must be convertible to `AdaptableBinaryFunction::first_argument_type`. This is a global function, not a member function.

Notes

Intuitively, you can think of this operation as "binding" the first argument of a binary function to a constant, thus yielding a unary function. This is a special case of a closure.

See also

The function object overview, `binder2nd`, Adaptable Unary Function, Adaptable Binary Function

10.4.2 binder2nd

Description

`Binder2nd` is a function object adaptor: it is used to transform an adaptable binary function into an adaptable unary function. Specifically, if `f` is an object of class `binder2nd<AdaptableBinaryFunction>`, then `f(x)` returns `F(x, c)`, where `F` is an object of class `AdaptableBinaryFunction` and where `c` is a constant. Both `F` and `c` are passed as arguments to `binder2nd`'s constructor. The easiest way to create a `binder2nd` is not to call the constructor explicitly, but instead to use the helper function `bind2nd`.

Example

Finds the first positive number in a list.

```
list<int> L;
...
list<int>::iterator first_positive =
       find_if(L.begin(), L.end(), bind2nd(greater<int>(), 0));
assert(first_positive == L.end() || *first_positive > 0);
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`AdaptableBinaryFunction`	The type of the binary function whose second argument is being bound to a constant.	

Model of

Adaptable Unary Function

Type requirements

`AdaptableBinaryFunction` must be a model of Adaptable Binary Function.

Public base classes

```
unary_function<AdaptableBinaryFunction::first_argument_type,
               AdaptableBinaryFunction::result_type>
```

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the function object's argument, which is `AdaptableBinaryFunction:: first_argument_type`
`result_type`	Adaptable Unary Function	The type of the result: `AdaptableBinaryFunction:: result_type`
`result_type operator()(const argument_type& x) const`	Adaptable Unary Function	Function call. Returns `F(x, c)`, where `F` and `c` are the arguments with which this `binder1st` was constructed.
`binder2nd(const AdaptableBinaryFunction& F, AdaptableBinaryFunction:: second_argument_type c)`	`binder2nd`	See below
`template <class AdaptableBinaryFunction, class T> binder2nd <AdaptableBinaryFunction> bind2nd(const AdaptableBinaryFunction& F, const T& c);`	`binder2nd`	See below

New members

These members are not defined in the Adaptable Unary Function requirements, but are specific to `binder2nd`.

Member	Description
`binder2nd(const AdaptableBinaryFunction& F, AdaptableBinaryFunction:: second_argument_type c)`	The constructor. Creates a `binder2nd` such that calling it with the argument `x` (where `x` is of type `AdaptableBinaryFunction::first_argument_type`) corresponds to the call `F(x, c)`.
`template <class AdaptableBinaryFunction, class T> binder2nd <AdaptableBinaryFunction> bind2nd(const AdaptableBinaryFunction& F, const T& c);`	If `F` is an object of type `AdaptableBinaryFunction`, then `bind2nd(F, c)` is equivalent to `binder2nd<AdaptableBinaryFunction>(F, c)`, but is more convenient. The type `T` must be convertible to `AdaptableBinaryFunction:: second_argument_type`. This is a global function, not a member function.

Intuitively, you can think of this operation as "binding" the second argument of a binary function to a constant, thus yielding a unary function. This is a special case of a closure.

See also

The function object overview, binder1st, Adaptable Unary Function, Adaptable Binary Function

10.4.3 ptr_fun

Prototype

```
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*x)(Arg));

template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*x)(Arg1, Arg2));
```

Description

Ptr_fun takes a function pointer as its argument and returns a function pointer adaptor, a type of function object. It is actually two different functions, not one (that is, the name ptr_fun is overloaded). If its argument is of type Result (*)(Arg) then ptr_fun creates a pointer_to_unary_function, and if its argument is of type Result (*)(Arg1, Arg2) then ptr_fun creates a pointer_to_binary_function.

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Requirements on types

The argument must be a pointer to a function that takes either one or two arguments. The argument type(s) and the return type of the function are arbitrary, with the restriction that the function must return a value; it may not be a void function.

Preconditions

Complexity

Example

See the examples in the discussions of pointer_to_unary_function and pointer_to_binary_function.

Notes

See also

Function Objects, `pointer_to_unary_function`, `pointer_to_binary_function`, Adaptable Unary Function, Adaptable Binary Function

10.4.4 pointer_to_unary_function

Description

`Pointer_to_unary_function` is a function object adaptor that allows a function pointer `Result (*f)(Arg)` to be treated as an Adaptable Unary Function. That is: if `F` is a `pointer_to_unary_function<Arg, Result>` that was initialized with an underlying function pointer `f` of type `Result (*)(Arg)`, then `F(x)` calls the function `f(x)`. The difference between `f` and `F` is that `pointer_to_unary_function` is an Adaptable Unary Function, *i.e.* it defines the nested `typedef`s `argument_type` and `result_type`. Note that a function pointer of type `Result (*)(Arg)` is a perfectly good Unary Function object, and may be passed to an STL algorithm that expects an argument that is a Unary Function. The only reason for using the `pointer_to_unary_function` object is if you need to use an ordinary function in a context that requires an Adaptable Unary Function, *e.g.* as the argument of a function object adaptor. Most of the time, you need not declare an object of type `pointer_to_unary_function` directly. It is almost always easier to construct one using the `ptr_fun` function.

Example

The following code fragment replaces all of the numbers in a range with their absolute values, using the standard library function `fabs`. There is no need to use a `pointer_to_unary_function` adaptor in this case.

```
transform(first, last, first, fabs);
```

The following code fragment replaces all of the numbers in a range with the negative of their absolute values. In this case we are composing `fabs` and `negate`. This requires that `fabs` be treated as an adaptable unary function, so we do need to use a `pointer_to_unary_function` adaptor.

```
transform(first, last, first,
          compose1(negate<double>, ptr_fun(fabs)));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Arg`	The function object's argument type	
`Result`	The function object's result type	

Model of

Adaptable Unary Function

Type requirements

- `Arg` is Assignable.
- `Result` is Assignable.

Public base classes

`unary_function<Arg, Result>`

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the function object's argument: `Arg`.
`result_type`	Adaptable Unary Function	The type of the result: `Result`
`result_type operator()(argument_type x)`	Unary Function	Function call operator.
`pointer_to_unary_function (Result (*f)(Arg))`	`pointer_to_unary_function`	See below.
`pointer_to_unary_function ()`	`pointer_to_unary_function`	See below.
`template <class Arg, class Result> pointer_to_unary_function <Arg, Result> ptr_fun(Result (*x)(Arg));`	`pointer_to_unary_function`	See below.

New members

These members are not defined in the Adaptable Unary Function requirements, but are specific to `pointer_to_unary_function`.

Member	Description
`pointer_to_unary_function (Result (*f)(Arg))`	The constructor. Creates a `pointer_to_unary_function` whose underlying function is `f`.
`pointer_to_unary_function()`	The default constructor. This creates a `pointer_to_unary_function` that does not have an underlying C function, and that therefore cannot actually be called.
`template <class Arg, class Result> pointer_to_unary_function<Arg, Result> ptr_fun(Result (*x)(Arg));`	If `f` is of type `Result (*)(Arg)` then `ptr_fun(f)` is equivalent to `pointer_to_unary_function<Arg,Result>(f)`, but more convenient. This is a global function, not a member.

Notes

See also

`pointer_to_binary_function`, `ptr_fun`, Adaptable Unary Function

10.4.5 pointer_to_binary_function

Description

`Pointer_to_binary_function` is a function object adaptor that allows a function pointer `Result (*f)(Arg1, Arg2)` to be treated as an Adaptable Binary Function. That is: if `F` is a `pointer_to_binary_function<Arg1, Arg2, Result>` that was initialized with an underlying function pointer `f` of type `Result (*)(Arg1, Arg2)`, then `F(x, y)` calls the function `f(x, y)`. The difference between `f` and `F` is that `pointer_to_binary_function` is an Adaptable Binary Function, *i.e.* it defines the nested `typedefs` `first_argument_type`, `second_argument_type`, and `result_type`. Note that a function pointer of type `Result (*)(Arg1, Arg2)` is a perfectly good Binary Function object, and may be passed to an STL algorithm that expects an argument that is a Binary Function. The only reason for using the `pointer_to_binary_function` class is if you need to use an ordinary function in a context that requires an Adaptable Binary Function, *e.g.* as the argument of a function object adaptor. Most of the time, you need not declare an object of type `pointer_to_binary_function` directly. It is almost always easier to construct one using the `ptr_fun` function.

Example

The following code fragment finds the first string in a list that is equal to "OK". It uses the standard library function `strcmp` as an argument to a function object adaptor, so it must first use a `pointer_to_binary_function` adaptor to give `strcmp` the Adaptable Binary Function interface.

```
list<char*> L;
...
list<char*>::iterator item =
              find_if(L.begin(), L.end(),
                      not1(binder2nd(ptr_fun(strcmp), "OK")));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Arg1`	The function object's first argument type	
`Arg2`	The function object's second argument type	
`Result`	The function object's result type	

Model of

Adaptable Binary Function

Type requirements

- `Arg1` is Assignable.
- `Arg2` is Assignable.
- `Result` is Assignable.

Public base classes

`binary_function<Arg1, Arg2, Result>`

Members

Member	Where defined	Description
first_argument_type	Adaptable Binary Function	The type of the first argument: Arg1.
second_argument_type	Adaptable Binary Function	The type of the second argument: Arg2
result_type	Adaptable Binary Function	The type of the result: Result
Result operator()(Arg1 x, Arg2 y)	Binary Function	Function call operator.
pointer_to_binary_function (Result (*f)(Arg1, Arg2))	pointer_to_binary_function	See below.
pointer_to_binary_function()	pointer_to_binary_function	See below.
template <class Arg1, class Arg2, class Result> pointer_to_unary_function<Arg1, Arg2, Result> ptr_fun(Result (*x)(Arg1, Arg2));	pointer_to_binary_function	See below.

New members

These members are not defined in the Adaptable Binary Function requirements, but are specific to pointer_to_binary_function.

Member	Description
pointer_to_binary_function (Result (*f)(Arg1, Arg2))	The constructor. Creates a pointer_to_binary_function whose underlying function is f.
pointer_to_binary_function()	The default constructor. This creates a pointer_to_binary_function that does not have an underlying function, and that therefore cannot actually be called.
template <class Arg1, class Arg2, class Result> pointer_to_unary_function <Arg1, Arg2, Result> ptr_fun(Result (*x)(Arg1, Arg2));	If f is of type Result (*)(Arg1, Arg2) then ptr_fun(f) is equivalent to pointer_to_binary_function <Arg1,Arg2,Result>(f), but more convenient. This is a global function, not a member function.

Notes

See also

pointer_to_unary_function, ptr_fun, Adaptable Binary Function

10.4.6 unary_negate

Description

`Unary_negate` is a function object adaptor: it is an Adaptable Predicate that represents the logical negation of some other Adaptable Predicate. That is: if `f` is an object of class `unary_negate<AdaptablePredicate>`, then there exists an object `pred` of class `AdaptablePredicate` such that `f(x)` always returns the same value as `!pred(x)`. There is rarely any reason to construct a `unary_negate` directly; it is almost always easier to use the helper function `not1`.

Example

Finds the first element in a list that does not lie in the range from 1 to 10.

```
list<int> L;
...
list<int>::iterator in_range =
     find_if(L.begin(), L.end(),
             not1(compose2(logical_and<bool>(),
                           bind2nd(greater_equal<int>(), 1),
                           bind2nd(less_equal<int>(), 10))));
assert(in_range == L.end() || !(*in_range >= 1 && *in_range <= 10));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`AdaptablePredicate`	The type of the function object that this `unary_negate` is the logical negation of.	

Model of

Adaptable Predicate

Type requirements

`AdaptablePredicate` must be a model of Adaptable Predicate.

Public base classes

unary_function<AdaptablePredicate::argument_type, bool>

Members

Member	Where defined	Description
argument_type	Adaptable Unary Function	The type of the argument: AdaptablePredicate::argument_type
result_type	Adaptable Unary Function	The type of the result: bool
bool operator()(argument_type)	Unary Function	Function call operator.
unary_negate(const AdaptablePredicate& pred)	unary_negate	See below.
template <class AdaptablePredicate> unary_negate <AdaptablePredicate> not1(const AdaptablePredicate& pred);	unary_negate	See below.

New members

These members are not defined in the Adaptable Predicate requirements, but are specific to unary_negate.

Member	Description
unary_negate(const AdaptablePredicate& pred)	The constructor. Creates a unary_negate<AdaptablePredicate> whose underlying predicate is pred.
template <class AdaptablePredicate> unary_negate <AdaptablePredicate> not1(const AdaptablePredicate& pred);	If p is of type AdaptablePredicate then not1(p) is equivalent to unary_negate<AdaptablePredicate>(p), but more convenient. This is a global function, not a member function.

Notes

Strictly speaking, unary_negate is redundant. It can be constructed using the function object logical_not and the adaptor unary_compose.

See also

The function object overview, Adaptable Predicate, Predicate, `binary_negate`, `unary_compose`, `binary_compose`

10.4.7 binary_negate

Description

`Binary_negate` is a function object adaptor: it is an Adaptable Binary Predicate that represents the logical negation of some other Adaptable Binary Predicate. That is: if `f` is an object of class binary_negate<AdaptableBinaryPredicate>, then there exists an object `pred` of class `AdaptableBinaryPredicate` such that `f(x,y)` always returns the same value as `!pred(x,y)`. There is rarely any reason to construct a `binary_negate` directly; it is almost always easier to use the helper function `not2`.

Example

Finds the first character in a string that is neither `' '` nor `'\n'`.

```
char str[MAXLEN];
...
const char* wptr = find_if(str, str + MAXLEN,
                           compose2(not2(logical_or<bool>()),
                                    bind2nd(equal_to<char>(), ' '),
                                    bind2nd(equal_to<char>(), '\n')));
assert(wptr == str + MAXLEN || !(*wptr == ' ' || *wptr == '\n'));
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`AdaptableBinaryPredicate`	The type of the function object that this `binary_negate` is the logical negation of.	

Model of

Adaptable Binary Predicate

Type requirements

`AdaptableBinaryPredicate` must be a model of Adaptable Binary Predicate.

Public base classes

```
binary_function<AdaptableBinaryPredicate::first_argument_type,
                AdaptableBinaryPredicate::second_argument_type,
                bool>
```

Members

Member	Where defined	Description
`first_argument_type`	Adaptable Binary Function	The type of the first argument: `AdaptableBinaryPredicate::first_argument_type`
`second_argument_type`	Adaptable Binary Function	The type of the second argument: `AdaptableBinaryPredicate::second_argument_type`
`result_type`	Adaptable Binary Function	The type of the result: `bool`
`binary_negate(const AdaptableBinaryPredicate& pred)`	`binary_negate`	See below.
`template <class AdaptableBinaryPredicate> binary_negate <AdaptableBinaryPredicate> not2(const AdaptableBinaryPredicate& pred);`	`binary_negate`	See below.

New members

These members are not defined in the Adaptable Binary Predicate requirements, but are specific to `binary_negate`.

Member	Description
`binary_negate(const AdaptableBinaryPredicate& pred)`	The constructor. Creates a `binary_negate<AdaptableBinaryPredicate>` whose underlying predicate is `pred`.
`template <class AdaptableBinaryPredicate> binary_negate <AdaptableBinaryPredicate> not2(const AdaptableBinaryPredicate& pred);`	If `p` is of type `AdaptableBinaryPredicate` then `not2(p)` is equivalent to `binary_negate<AdaptableBinaryPredicate>(p)`, but more convenient. This is a global function, not a member function.

Notes

See also

The function object overview, AdaptablePredicate, Predicate, `unary_negate`, `unary_compose`, `binary_compose`

10.4.8 Member function adaptors

mem_fun

Description

`Mem_fun_t` is an adaptor for member functions. If `X` is some class with a member function `Result X::f()` (that is, a member function that takes no arguments and that returns a value of type `Result`), then a `mem_fun_t<Result, X>` is a function object adaptor that makes it possible to call `f()` as if it were an ordinary function instead of a member function. `Mem_fun_t<Result, X>`'s constructor takes a pointer to one of `X`'s member functions. Then, like all function objects, `mem_fun_t` has an `operator()` that allows the `mem_fun_t` to be invoked with ordinary function call syntax. In this case, `mem_fun_t`'s `operator()` takes an argument of type `X*`. If `F` is a `mem_fun_t` that was constructed to use the member function `X::f`, and if `x` is a pointer of type `X*`, then the expression `F(x)` is equivalent to the expression `x->f()`. The difference is simply that `F` can be passed to STL algorithms whose arguments must be function objects. `Mem_fun_t` is one of a family of member function adaptors. These adaptors are useful if you want to combine generic programming with inheritance and polymorphism, since, in C++, polymorphism involves calling member functions through pointers or references. As with many other adaptors, it is usually inconvenient to use `mem_fun_t`'s constructor directly. It is usually better to use the helper function `mem_fun` instead.

Example

```
struct B {
  virtual void print() = 0;
};

struct D1 : public B {
  void print() { cout << "I'm a D1" << endl; }
};

struct D2 : public B {
  void print() { cout << "I'm a D2" << endl; }
};

int main()
{
  vector<B*> V;

  V.push_back(new D1);
  V.push_back(new D2);
  V.push_back(new D2);
  V.push_back(new D1);

  for_each(V.begin(), V.end(), mem_fun(&B::print));
}
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Result`	The member function's return type.	
`X`	The class whose member function the `mem_fun_t` invokes.	

Model of

Adaptable Unary Function

Type requirements

- `X` has at least one member function that takes no arguments and that returns a value of type `Result`.

Public base classes

`unary_function<X*, Result>`

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the argument: `X*`
`result_type`	Adaptable Unary Function	The type of the result: `Result`
`Result operator()(X* x) const`	Unary Function	Function call operator. Invokes `x->f()`, where `f` is the member function that was passed to the constructor.
`explicit mem_fun_t(Result (X::*f)())`	`mem_fun_t`	See below.
`template <class Result, class X> mem_fun_t<Result, X> mem_fun(Result (X::*f)());`	`mem_fun_t`	See below.

New members

These members are not defined in the Adaptable Unary Function requirements, but are specific to `mem_fun_t`.

Member	Description
`explicit mem_fun_t(Result (X::*f)())`	The constructor. Creates a `mem_fun_t` that calls the member function `f`.
`template <class Result, class X> mem_fun_t<Result, X> mem_fun(Result (X::*f)());`	If `f` if of type `Result (X::*)` then `mem_fun(f)` is the same as `mem_fun_t<Result, X>(f)`, but is more convenient. This is a global function, not a member function.

Notes

See also

`mem_fun_ref_t`, `mem_fun1_t`, `mem_fun1_ref_t`

mem_fun_ref

Description

Mem_fun_ref_t is an adaptor for member functions. If X is some class with a member function Result X::f() (that is, a member function that takes no arguments and that returns a value of type Result), then a mem_fun_ref_t<Result, X> is a function object adaptor that makes it possible to call f() as if it were an ordinary function instead of a member function. mem_fun_ref_t<Result, X>'s constructor takes a pointer to one of X's member functions. Then, like all function objects, mem_fun_ref_t has an operator() that allows the mem_fun_ref_t to be invoked with ordinary function call syntax. In this case, mem_fun_ref_t's operator() takes an argument of type X&. If F is a mem_fun_ref_t that was constructed to use the member function X::f, and if x is of type X, then the expression F(x) is equivalent to the expression x.f(). The difference is simply that F can be passed to STL algorithms whose arguments must be function objects. Mem_fun_ref_t is one of a family of member function adaptors. These adaptors are useful if you want to combine generic programming with inheritance and polymorphism, since, in C++, polymorphism involves calling member functions through pointers or references. In fact, though, mem_fun_ref_t is usually not as useful as mem_fun_t. The difference between the two is that mem_fun_t's argument is a pointer to an object while mem_fun_ref_t's argument is a reference to an object. References, unlike pointers, can't be stored in STL containers: pointers are objects in their own right, but references are merely aliases. As with many other adaptors, it is usually inconvenient to use mem_fun_ref_t's constructor directly. It is usually better to use the helper function mem_fun_ref instead.

Example

```
struct B {
  virtual void print() = 0;
};

struct D1 : public B {
  void print() { cout << "I'm a D1" << endl; }
};

struct D2 : public B {
  void print() { cout << "I'm a D2" << endl; }
};

int main()
{
  vector<D1> V;

  V.push_back(D1());
  V.push_back(D1());

  for_each(V.begin(), V.end(), mem_fun_ref(B::print));
}
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Result`	The member function's return type.	
`X`	The class whose member function the `mem_fun_ref_t` invokes.	

Model of

Adaptable Unary Function

Type requirements

- `X` has at least one member function that takes no arguments and that returns a value of type `Result`.

Public base classes

`unary_function<X, Result>`

Members

Member	Where defined	Description
`argument_type`	Adaptable Unary Function	The type of the argument: `X`
`result_type`	Adaptable Unary Function	The type of the result: `Result`
`Result operator()(X& x) const`	Unary Function	Function call operator. Invokes `x.f()`, where `f` is the member function that was passed to the constructor.
`explicit mem_fun_ref_t(Result (X::*f)())`	`mem_fun_ref_t`	See below.
`template <class Result, class X> mem_fun_ref_t<Result, X> mem_fun_ref(Result (X::*f)());`	`mem_fun_ref_t`	See below.

New members

These members are not defined in the Adaptable Unary Function requirements, but are specific to `mem_fun_ref_t`.

Member	Description
`explicit mem_fun_ref_t(Result (X::*f)())`	The constructor. Creates a `mem_fun_ref_t` that calls the member function `f`.
`template <class Result, class X> mem_fun_ref_t<Result, X> mem_fun_ref(Result (X::*f)());`	If `f` is of type `Result (X::*)()` then `mem_fun_ref(f)` is the same as `mem_fun_ref_t<Result, X>(f)`, but is more convenient. This is a global function, not a member function.

Notes

See also

`mem_fun_t`, `mem_fun1_t`, `mem_fun1_ref_t`

mem_fun1

Description

Mem_fun1_t is an adaptor for member functions. If X is some class with a member function Result X::f(Arg) (that is, a member function that takes one argument of type Arg and that returns a value of type Result), then a mem_fun1_t<Result, X, Arg> is a function object adaptor that makes it possible to call f as if it were an ordinary function instead of a member function. Mem_fun1_t<Result, X, Arg>'s constructor takes a pointer to one of X's member functions. Then, like all function objects, mem_fun1_t has an operator() that allows the mem_fun1_t to be invoked with ordinary function call syntax. In this case, mem_fun1_t's operator() takes two arguments; the first is of type X* and the second is of type Arg. If F is a mem_fun1_t that was constructed to use the member function X::f, and if x is a pointer of type X* and a is a value of type Arg, then the expression F(x, a) is equivalent to the expression x->f(a). The difference is simply that F can be passed to STL algorithms whose arguments must be function objects. Mem_fun1_t is one of a family of member function adaptors. These adaptors are useful if you want to combine generic programming with inheritance and polymorphism, since, in C++, polymorphism involves calling member functions through pointers or references. As with many other adaptors, it is usually inconvenient to use mem_fun1_t's constructor directly. It is usually better to use the helper function mem_fun instead.

Example

```
struct Operation {
  virtual double eval(double) = 0;
};

struct Square : public Operation {
  double eval(double x) { return x * x; }
};

struct Negate : public Operation {
  double eval(double x) { return -x; }
};

int main() {
  vector<Operation*> operations;
  vector<double> operands;

  operations.push_back(new Square);
  operations.push_back(new Square);
  operations.push_back(new Negate);
  operations.push_back(new Negate);
  operations.push_back(new Square);

  operands.push_back(1);
  operands.push_back(2);
  operands.push_back(3);
  operands.push_back(4);
  operands.push_back(5);

  transform(operations.begin(), operations.end(),
            operands.begin(),
            ostream_iterator<double>(cout, "\n"),
            mem_fun(Operation::eval));
}
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Result`	The member function's return type.	
`X`	The class whose member function the `mem_fun1_t` invokes.	
`Arg`	The member function's argument type.	

Model of

Adaptable Binary Function

Type requirements

- X has at least one member function that takes a single argument of type Arg and that returns a value of type Result.

Public base classes

binary_function<X*, Arg, Result>

Members

Member	Where defined	Description
first_argument_type	Adaptable Binary Function	The type of the first argument: X*
second_argument_type	Adaptable Binary Function	The type of the second argument: Arg
result_type	Adaptable Binary Function	The type of the result: Result
Result operator()(X* x, Arg a) const	Binary Function	Function call operator. Invokes x->f(a), where f is the member function that was passed to the constructor.
explicit mem_fun1_t(Result (X::*f)(Arg))	mem_fun1_t	See below.
template <class Result, class X, class Arg> mem_fun1_t<Result, X, Arg> mem_fun(Result (X::*f)(Arg));	mem_fun1_t	See below.

New members

These members are not defined in the Adaptable Binary Function requirements, but are specific to mem_fun1_t.

Member	Description
explicit mem_fun1_t(Result (X::*f)(Arg))	The constructor. Creates a mem_fun1_t that calls the member function f.
template <class Result, class X, class Arg> mem_fun1_t<Result, X, Arg> mem_fun(Result (X::*f)(Arg));	If f is of type Result (X::*)(Arg) then mem_fun(f) is the same as mem_fun1_t<Result, X, Arg>(f), but is more convenient. This is a global function, not a member function.

Notes

See also

mem_fun_t, mem_fun_ref_t, mem_fun1_ref_t

mem_fun1_ref

Description

Mem_fun1_ref_t is an adaptor for member functions. If X is some class with a member function Result X::f(Arg) (that is, a member function that takes one argument of type Arg and that returns a value of type Result), then a mem_fun1_ref_t<Result, X, Arg> is a function object adaptor that makes it possible to call f as if it were an ordinary function instead of a member function. Mem_fun1_ref_t<Result, X, Arg>'s constructor takes a pointer to one of X's member functions. Then, like all function objects, mem_fun1_ref_t has an operator() that allows the mem_fun1_ref_t to be invoked with ordinary function call syntax. In this case, mem_fun1_ref_t's operator() takes two arguments; the first is of type X and the second is of type Arg. If F is a mem_fun1_ref_t that was constructed to use the member function X::f, and if x is an object of type X and a is a value of type Arg, then the expression F(x, a) is equivalent to the expression x.f(a). The difference is simply that F can be passed to STL algorithms whose arguments must be function objects. Mem_fun1_ref_t is one of a family of member function adaptors. These adaptors are useful if you want to combine generic programming with inheritance and polymorphism, since, in C++, polymorphism involves calling member functions through pointers or references. In fact, though, mem_fun1_ref_t is usually not as useful as mem_fun1_t. The difference between the two is that mem_fun1_t's first argument is a pointer to an object while mem_fun1_ref_t's argument is a reference to an object. References, unlike pointers, can't be stored in STL containers: pointers are objects in their own right, but references are merely aliases. As with many other adaptors, it is usually inconvenient to use mem_fun1_ref_t's constructor directly. It is usually better to use the helper function mem_fun_ref instead.

Example

Given a vector of vectors, extract one element from each vector.

```
int main() {
  int A1[5] = {1, 2, 3, 4, 5};
  int A2[5] = {1, 1, 2, 3, 5};
  int A3[5] = {1, 4, 1, 5, 9};

  vector<vector<int> > V;
  V.push_back(vector<int>(A1, A1 + 5));
  V.push_back(vector<int>(A2, A2 + 5));
  V.push_back(vector<int>(A3, A3 + 5));

  int indices[3] = {0, 2, 4};

  int& (vector<int>::*extract)(vector<int>::size_type);
  extract = vector<int>::operator[];
  transform(V.begin(), V.end(), indices,
            ostream_iterator<int>(cout, "\n"),
            mem_fun_ref(extract));
}
```

Definition

Defined in the standard header functional, and in the nonstandard backward-compatibility header function.h.

Template parameters

Parameter	Description	Default
`Result`	The member function's return type.	
`X`	The class whose member function the `mem_fun1_ref_t` invokes.	
`Arg`	The member function's argument type.	

Model of

Adaptable Binary Function

Type requirements

- `X` has at least one member function that takes a single argument of type `Arg` and that returns a value of type `Result`.

Public base classes

`binary_function<X, Arg, Result>`

Members

Member	Where defined	Description
first_argument_type	Adaptable Binary Function	The type of the first argument: X
second_argument_type	Adaptable Binary Function	The type of the second argument: Arg
result_type	Adaptable Binary Function	The type of the result: Result
Result operator()(X& x, Arg a) const	Binary Function	Function call operator. Invokes x.f(a), where f is the member function that was passed to the constructor.
explicit mem_fun1_ref_t(Result (X::*f)(Arg))	mem_fun1_ref_t	See below.
template <class Result, class X, class Arg> mem_fun1_ref_t<Result, X, Arg> mem_fun_ref(Result (X::*f)(Arg));	mem_fun1_ref_t	See below.

New members

These members are not defined in the Adaptable Binary Function requirements, but are specific to mem_fun1_ref_t.

Member	Description
explicit mem_fun1_ref_t(Result (X::*f)(Arg))	The constructor. Creates a mem_fun1_ref_t that calls the member function f.
template <class Result, class X, class Arg> mem_fun1_ref_t<Result, X, Arg> mem_fun1_ref(Result (X::*f)(Arg));	If f is of type Result (X::*)(Arg) then mem_fun_ref(f) is the same as mem_fun1_ref_t<Result, X, Arg>(f), but is more convenient. This is a global function, not a member function.

Notes

See also

mem_fun_t, mem_fun_ref_t, mem_fun1_t

Chapter 11

Utilities

11.1 Concepts

11.1.1 Assignable

Description

A type is Assignable if it is possible to copy objects of that type and to assign values to variables.

Refinement of

Associated types

Notation

`X`	A type that is a model of Assignable
`x`, `y`	Object of type `X`

Definitions

Valid expressions

Name	Expression	Type requirements	Return type
Copy constructor	`X(x)`		`X`
Copy constructor	`X x(y);` `X x = y;`		
Assignment	`x = y`		`X&`
Swap	`swap(x,y)`		`void`

Expression semantics

Name	Expression	Pre-condition	Semantics	Postcondition
Copy constructor	`X(x)`			`X(x)` is a copy of `x`
Copy constructor	`X(x)`			`X(x)` is a copy of `x`
Copy constructor	`X x(y);` `X x = y;`			`x` is a copy of `y`
Assignment	`x = y`			`x` is a copy of `y`

Swap	`swap(x,y)`		Equivalent to
			{
			`X tmp = x;`
			`x = y;`
			`y = tmp;`
			}	

Complexity guarantees

Invariants

Models

- int

Notes

One implication of this requirement is that a `const` type is not Assignable. For example, `const int` is not Assignable: if `x` is declared to be of type `const int`, then `x = 7` is illegal. Similarly, the type `pair<const int, int>` is not Assignable. The reason this says "`x` is a copy of `y`", rather than "`x == y`", is that `operator==` is not necessarily defined: equality is not a requirement of Assignable. If the type `X` is EqualityComparable as well as Assignable, then a copy of `x` should compare equal to `x`.

See also

DefaultConstructible

11.1.2 Default Constructible

Description

A type is DefaultConstructible if it has a default constructor, that is, if it is possible to construct an object of that type without initializing the object to any particular value.

Refinement of

Associated types

Notation

`X`	A type that is a model of DefaultConstructible
`x`	An object of type `X`

Definitions

Valid expressions

Name	Expression	Type reqs	Return type
Default constructor	`X()`		`X`
Default constructor	`X x;`		

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default constructor	`X()`			
Default constructor	`X x;`			

Complexity guarantees

Models

- int
- `vector<double>`

Notes

The form `X x = X()` is not guaranteed to be a valid expression, because it uses a copy constructor. A type that is DefaultConstructible is not necessarily Assignable

See also

Assignable

11.1.3 Equality Comparable

Description

A type is EqualityComparable if objects of that type can be compared for equality using `operator==`, and if `operator==` is an equivalence relation.

Refinement of

Associated types

Notation

`X`	A type that is a model of EqualityComparable
`x`, `y`, `z`	Object of type `X`

Definitions

Valid expressions

Name	Expression	Type reqs	Return type
Equality	`x == y`		Convertible to `bool`
Inequality	`x != y`		Convertible to `bool`

Expression semantics

Name	Expression	Precondition	Semantics	Post-condition
Equality	`x == y`	`x` and `y` are in the domain of `==`		
Inequality	`x != y`	`x` and `y` are in the domain of `==`	Equivalent to `!(x == y)`	

Complexity guarantees

Invariants

Identity	`&x == &y` implies `x == y`
Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

Models

- int
- `vector<int>`

Notes

See also

LessThanComparable.

11.1.4 LessThan Comparable

Description

A type is LessThanComparable if it is ordered: it must be possible to compare two objects of that type using `operator<`, and `operator<` must be a partial ordering.

Refinement of

Associated types

Notation

`X`	A type that is a model of LessThanComparable
`x`, `y`, `z`	Object of type `X`

Definitions

Consider the relation `!(x < y) && !(y < x)`. If this relation is transitive (that is, if `!(x < y) && !(y < x) && !(y < z) && !(z < y)` implies `!(x < z) && !(z < x)`), then it satisfies the mathematical definition of an equivalence relation. In this case, `operator<` is a *strict weak ordering*. If `operator<` is a strict weak ordering, and if each equivalence class has only a single element, then `operator<` is a *total ordering*.

Valid expressions

Name	Expression	Type reqs	Return type
Less	`x < y`		Convertible to `bool`
Greater	`x > y`		Convertible to `bool`
Less or equal	`x <= y`		Convertible to `bool`
Greater or equal	`x >= y`		Convertible to `bool`

Expression semantics

Name	Expression	Precondition	Semantics	Post-condition
Less	`x < y`	`x` and `y` are in the domain of `<`		
Greater	`x > y`	`x` and `y` are in the domain of `<`	Equivalent to `y < x`	
Less or equal	`x <= y`	`x` and `y` are in the domain of `<`	Equivalent to `!(y < x)`	
Greater or equal	`x >= y`	`x` and `y` are in the domain of `<`	Equivalent to `!(x < y)`	

Complexity guarantees

Invariants

Irreflexivity	`x < x` must be false.
Antisymmetry	`x < y` implies !(y ¡ x)
Transitivity	`x < y` and `y < z` implies `x < z`

Models

- int

Notes

Only `operator<` is fundamental; the other inequality operators are essentially syntactic sugar. Antisymmetry is a theorem, not an axiom: it follows from irreflexivity and transitivity. Because of irreflexivity and transitivity, `operator<` always satisfies the definition of a *partial ordering*. The definition of a *strict weak ordering* is stricter, and the definition of a *total ordering* is stricter still.

See also

EqualityComparable, StrictWeakOrdering

11.2 Functions

11.2.1 Relational Operators

Prototype

```
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

The Equality Comparable requirements specify that it must be possible to compare objects using `operator!=` as well as `operator==`; similarly, the LessThan Comparable requirements include `operator>`, `operator<=` and `operator>=` as well as `operator<`. Logically, however, most of these operators are redundant: all of them can be defined in terms of `operator==` and `operator<`. These four templates use `operator==` and `operator<` to define the other four relational operators. They exist purely for the sake of convenience: they make it possible to write algorithms in terms of the operators `!=`, `>`, `<=`, and `>=`, without requiring that those operators be explicitly defined for every type. As specified in the Equality Comparable requirements, `x != y` is equivalent to `!(x == y)`. As specified in the LessThan Comparable requirements, `x > y` is equivalent to `y < x`, `x >= y` is equivalent to `!(x < y)`, and `x <= y` is equivalent to `!(y < x)`.

Definition

Defined in the standard header utility, and in the nonstandard backward-compatibility header function.h.

Requirements on types

The requirement for `operator!=` is that `x == y` is a valid expression for objects `x` and `y` of type `T`. The requirement for `operator>` is that `y < x` is a valid expression for objects `x` and `y` of type `T`. The requirement for `operator<=` is that `y < x` is a valid expression for objects `x` and `y` of type `T`. The requirement for `operator>=` is that `x < y` is a valid expression for objects `x` and `y` of type `T`.

Preconditions

The precondition for `operator!=` is that `x` and `y` are in the domain of `operator==`. The precondition for `operator>`, `operator<=`, and `operator>=` is that `x` and `y` are in the domain of `operator<`.

Complexity

Example

```
template <class T> void relations(T x, T y)
{
  if (x == y) assert(!(x != y));
  else assert(x != y);

  if (x < y) {
    assert(x <= y);
    assert(y > x);
    assert(y >= x);
  }
  else if (y < x) {
    assert(y <= x);
    assert(x < y);
    assert(x <= y);
  }
  else {
    assert(x <= y);
    assert(x >= y);
  }
}
```

Notes

See also

Equality Comparable, LessThan Comparable

11.3 Classes

11.3.1 pair

Description

`Pair<T1,T2>` is a heterogeneous pair: it holds one object of type `T1` and one of type `T2`. A pair is much like a Container, in that it "owns" its elements. It is not actually a model of Container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a Container. Functions that need to return two values often return a `pair`.

Example

```
pair<bool, double> result = do_a_calculation();
if (result.first)
  do_something_more(result.second);
else
  report_error();
```

Definition

Defined in the standard header utility, and in the nonstandard backward-compatibility header pair.h.

Template parameters

Parameter	Description	Default
`T1`	The type of the first element stored in the `pair`	
`T2`	The type of the second element stored in the `pair`	

Model of

Assignable

Type requirements

`T1` and `T2` must both be models of Assignable. Additional operations have additional requirements. `Pair`'s default constructor may only be used if both `T1` and `T2` are DefaultConstructible, `operator==` may only be used if both `T1` and `T2` are EqualityComparable, and `operator<` may only be used if both `T1` and `T2` are LessThanComparable.

Public base classes

None.

Members

Member	Where defined	Description
first_type	pair	See below.
second_type	pair	See below.
pair()	pair	The default constructor. See below.
pair(const first_type&, const second_type&)	pair	The pair constructor. See below.
pair(const pair&)	Assignable	The copy constructor
pair& operator=(const pair&)	Assignable	The assignment operator
first	pair	See below.
second	pair	See below.
bool operator==(const pair&, const pair&)	pair	See below.
bool operator<(const pair&, const pair&)	pair	See below.
template <class T1, class T2> pair<T1, T2> make_pair(const T1&, const T2&)	pair	See below.

New members

These members are not defined in the Assignable requirements, but are specific to pair.

Member	Description
first_type	The type of the pair's first component. This is a typedef for the template parameter T1
second_type	The type of the pair's second component. This is a typedef for the template parameter T2
pair()	The default constructor. It uses constructs objects of types T1 and T2 using their default constructors. This constructor may only be used if both T1 and T2 are DefaultConstructible.
pair(const first_type& x, const second_type& y)	The pair constructor. Constructs a pair such that first is constructed from x and second is constructed from y.
first	Public member variable of type first_type: the first object stored in the pair.
second	Public member variable of type second_type: The second object stored in the pair.
template <class T1, class T2> bool operator==(const pair<T1,T2>& x, const pair<T1,T2>& y);	The equality operator. The return value is true if and only the first elements of x and y are equal, and the second elements of x and y are equal. This operator may only be used if both T1 and T2 are EqualityComparable. This is a global function, not a member function.
template <class T1, class T2> bool operator<(const pair<T1,T2>& x, const pair<T1,T2>& y);	The comparison operator. It uses lexicographic comparison: the return value is true if the first element of x is less than the first element of y, and false if the first element of y is less than the first element of x. If neither of these is the case, then operator< returns the result of comparing the second elements of x and y. This operator may only be used if both T1 and T2 are LessThanComparable. This is a global function, not a member function.
template <class T1, class T2> pair<T1, T2> make_pair(const T1& x, const T2& x)	Equivalent to pair<T1, T2>(x, y). This is a global function, not a member function. It exists only for the sake of convenience.

Notes

See also

Assignable, Default Constructible, LessThan Comparable

Chapter 12

Memory Allocation

12.1 Classes

12.1.1 Allocators

Summary

Allocators encapsulate allocation and deallocation of memory. They provide a low-level interface that permits efficient allocation of many small objects; different allocator types represent different schemes for memory management. Note that allocators simply allocate and deallocate memory, as opposed to creating and destroying objects. The STL also includes several low-level algorithms for manipulating uninitialized memory. Note also that allocators do not attempt to encapsulate multiple memory models. The C++ language only defines a single memory model (the difference of two pointers, for example, is always `ptrdiff_t`), and this memory model is the only one that allocators support. *This is a major change from the definition of allocators in the original STL.*

Description

The details of the allocator interface are still subject to change, and we do not guarantee that specific member functions will remain in future versions. You should think of an allocator as a "black box". That is, you may select a container's memory allocation strategy by instantiating the container template with a particular allocator , but you should not make any assumptions about how the container actually uses the allocator. The available allocators are as follows. In most cases you shouldn't have to worry about the distinction: the default allocator, `alloc`, is usually the best choice.

`alloc`	The default allocator. It is thread-safe, and usually has the best performance characteristics.
`pthread_alloc`	A thread-safe allocator that uses a different memory pool for each thread; you can only use `pthread_alloc` if your operating system provides pthreads. `Pthread_alloc` is usually faster than `alloc`, especially on multiprocessor systems. It can, however, cause resource fragmentation: memory deallocated in one thread is not available for use by other threads.
`single_client_alloc`	A fast but thread-unsafe allocator. In programs that only have one thread, this allocator might be faster than `alloc`.
`malloc_alloc`	An allocator that simply uses the standard library function `malloc`. It is thread-safe but slow; the main reason why you might sometimes want to use it is to get more useful information from bounds-checking or leak-detection tools while you are debugging.

Examples

```
vector<double> V(100, 5.0);     // Uses the default allocator.
vector<double, single_client_alloc> local(V.begin(), V.end());
```

Concepts

- Allocator

Types

- `alloc`
- `pthread_alloc`
- `single_client_alloc`
- `malloc_alloc`
- `raw_storage_iterator`

Functions

- `construct`
- `destroy`
- `uninitialized_copy`

- `uninitialized_fill`
- `uninitialized_fill_n`
- `get_temporary_buffer`
- `return_temporary_buffer`

Notes

Different containers may use different allocators. You might, for example, have some containers that use the default allocator `alloc` and others that use `pthread_alloc`. Note, however, that `vector<int>` and `vector<int, pthread_alloc>` are distinct types.

See also

12.1.2 raw_storage_iterator

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If `i` is an iterator that points to a region of uninitialized memory, then you can use `construct` to create an object in the location pointed to by `i`. `Raw_storage_iterator` is an adaptor that makes this procedure more convenient. If `r` is a `raw_storage_iterator`, then it has some underlying iterator `i`. The expression `*r = x` is equivalent to `construct(&*i, x)`.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  transform(A1, A1 + N,
            raw_storage_iterator<Int*, int>(A2),
            negate<int>());
}
```

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header iterator.h.

Template parameters

Parameter	Description	Default
OutputIterator	The type of the `raw_storage_iterator`'s underlying iterator.	
`T`	The type that will be used as the argument to the constructor.	

Model of

Output Iterator

Type requirements

- `ForwardIterator` is a model of Forward Iterator
- `ForwardIterator`'s value type has a constructor that takes a single argument of type `T`.

Public base classes

None.

Members

Member	Where defined	Description
`raw_storage_iterator (ForwardIterator x)`	`raw_storage_iterator`	See below.
`raw_storage_iterator(const raw_storage_iterator&)`	trivial iterator	The copy constructor
`raw_storage_iterator& operator=(const raw_storage_iterator&)`	trivial iterator	The assignment operator
`raw_storage_iterator& operator*()`	Output Iterator	Used to implement the output iterator expression `*i = x`.
`raw_storage_iterator& operator=(const Sequence::value_type&)`	Output Iterator	Used to implement the output iterator expression `*i = x`.
`raw_storage_iterator& operator++()`	Output Iterator	Preincrement.
`raw_storage_iterator& operator++(int)`	Output Iterator	Postincrement.
`output_iterator_tag iterator_category(const raw_storage_iterator&)`	iterator tags	Returns the iterator's category. This is a global function, not a member.

New members

These members are not defined in the Output Iterator requirements, but are specific to `raw_storage_iterator`.

Function	Description
`raw_storage_iterator (ForwardIterator i)`	Creates a `raw_storage_iterator` whose underlying iterator is `i`.
`raw_storage_iterator& operator=(const T& val)`	Constructs an object of `ForwardIterator`'s value type at the location pointed to by the iterator, using `val` as the constructor's argument.

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes.

See also

Allocators, `construct`, `destroy`, `uninitialized_copy` `uninitialized_fill`, `uninitialized_fill_n`,

12.2 Functions

12.2.1 uninitialized_copy

Prototype

```
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first,
                                   InputIterator last,
                                   ForwardIterator result);
```

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If each iterator in the range `[result, result + (last - first))` points to uninitialized memory, then `uninitialized_copy` creates a copy of `[first, last)` in that range. That is, for each iterator `i` in the input range, `uninitialized_copy` creates a copy of `*i` in the location pointed to by the corresponding iterator in the output range by calling `construct(&*(result + (i - first)), *i)`.

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `ForwardIterator`'s value type has a constructor that takes a single argument whose type is `InputIterator`'s value type.

Preconditions

- `[first, last)` is a valid range.
- `[result, result + (last - first))` is a valid range.
- Each iterator in `[result, result + (last - first))` points to a region of uninitialized memory that is large enough to store a value of `ForwardIterator`'s value type.

Complexity

Linear. Exactly `last - first` constructor calls.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  uninitialized_copy(A1, A1 + N, A2);
}
```

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes.

See also

Allocators, `construct`, `destroy`, `uninitialized_fill`, `uninitialized_fill_n`, `raw_storage_iterator`

12.2.2 uninitialized_copy_n

Prototype

```
template <class InputIterator, class Size, class ForwardIterator>
ForwardIterator uninitialized_copy_n(InputIterator first, Size count,
                                     ForwardIterator result);
```

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If each iterator in the range `[result, result + n)` points to uninitialized memory, then `uninitialized_copy_n` creates a copy of `[first, first + n)` in that range. That is, for each iterator `i` in the input range, `uninitialized_copy_n` creates a copy of `*i` in the location pointed to by the corresponding iterator in the output range by calling `construct(&*(result + (i - first)), *i)`.

Definition

Defined in the standard header memory.

Requirements on types

- `InputIterator` is a model of Input Iterator.
- `Size` is an integral type.
- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `ForwardIterator`'s value type has a constructor that takes a single argument whose type is `InputIterator`'s value type.

Preconditions

- `n >= 0`
- `[first, first + n)` is a valid range.
- `[result, result + n)` is a valid range.
- Each iterator in `[result, result + n)` points to a region of uninitialized memory that is large enough to store a value of `ForwardIterator`'s value type.

Complexity

Linear. Exactly `n` constructor calls.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  int A1[] = {1, 2, 3, 4, 5, 6, 7};
  const int N = sizeof(A1) / sizeof(int);

  Int* A2 = (Int*) malloc(N * sizeof(Int));
  uninitialized_copy_n(A1, N, A2);
}
```

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes. `Uninitialized_copy_n` is almost, but not quite, redundant. If `first` is an input iterator, as opposed to a forward iterator, then the `uninitialized_copy_n` operation can't be expressed in terms of `uninitialized_copy`.

See also

Allocators, `construct`, `destroy`, `uninitialized_copy`, `uninitialized_fill`, `uninitialized_fill_n`, `raw_storage_iterator`

12.2.3 uninitialized_fill

Prototype

```
template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                        const T& x);
```

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If each iterator in the range `[first, last)` points to uninitialized memory, then `uninitialized_fill` creates copies of `x` in that range. That is, for each iterator `i` in the range `[first, last)`, `uninitialized_copy` creates a copy of `x` in the location pointed to `i` by calling `construct(&*i, x)`.

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `ForwardIterator`'s value type has a constructor that takes a single argument of type `T`.

Preconditions

- `[first, last)` is a valid range.
- Each iterator in `[first, last)` points to a region of uninitialized memory that is large enough to store a value of `ForwardIterator`'s value type.

Complexity

Linear. Exactly `last - first` constructor calls.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  const int N = 137;

  Int val(46);
  Int* A = (Int*) malloc(N * sizeof(Int));
  uninitialized_fill(A, A + N, val);
}
```

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes.

See also

Allocators, `construct`, `destroy`, `uninitialized_copy`, `uninitialized_fill_n`, `raw_storage_iterator`

12.2.4 uninitialized_fill_n

Prototype

```
template <class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n,
                                     const T& x);
```

Description

In C++, the operator `new` allocates memory for an object and then creates an object at that location by calling a constructor. Occasionally, however, it is useful to separate those two operations. If each iterator in the range `[first, first + n)` points to uninitialized memory, then `uninitialized_fill_n` creates copies of `x` in that range. That is, for each iterator `i` in the range `[first, first + n)`, `uninitialized_fill_n` creates a copy of `x` in the location pointed to `i` by calling `construct(&*i, x)`.

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

- `ForwardIterator` is a model of Forward Iterator.
- `ForwardIterator` is mutable.
- `Size` is an integral type that is convertible to `ForwardIterator`'s distance type.
- `ForwardIterator`'s value type has a constructor that takes a single argument of type `T`.

Preconditions

- `n` is nonnegative.
- `[first, first + n)` is a valid range.
- Each iterator in `[first, first + n)` points to a region of uninitialized memory that is large enough to store a value of `ForwardIterator`'s value type.

Complexity

Linear. Exactly `n` constructor calls.

Example

```
class Int {
public:
  Int(int x) : val(x) {}
  int get() { return val; }
private:
  int val;
};

int main()
{
  const int N = 137;

  Int val(46);
  Int* A = (Int*) malloc(N * sizeof(Int));
  uninitialized_fill_n(A, N, val);
}
```

Notes

In particular, this sort of low-level memory management is used in the implementation of some container classes.

See also

Allocators, `construct`, `destroy`, `uninitialized_copy`, `uninitialized_fill`, `raw_storage_iterator`

12.2.5 get_temporary_buffer

Prototype

```
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t len, T*);
```

Description

Some algorithms, such as `stable_sort` and `inplace_merge`, are *adaptive*: they attempt to use extra temporary memory to store intermediate results, and their run-time complexity is better if that extra memory is available. The first argument to `get_temporary_buffer` specifies the requested size of the temporary buffer, and the second specifies the type of object that will be stored in the buffer. That is, `get_temporary_buffer(len, (T*) 0)` requests a buffer that is aligned for objects of type `T` and that is large enough to hold `len` objects of type `T`. The return value of `get_temporary_buffer` is a pair `P` whose first component is a pointer to the temporary buffer and whose second argument indicates how large the buffer is: the buffer pointed to by `P.first` is large enough to hold `P.second` objects of type `T`. `P.second` is greater than or equal to `0` , and less than or equal to `len`. Note that `P.first` is a pointer to uninitialized memory, rather than to actual objects of type `T`; this memory can be initialized using `uninitialized_copy`, `uninitialized_fill`, or `uninitialized_fill_n`. As the name suggests, `get_temporary_buffer` should only be used to obtain temporary memory. If a function allocates memory using `get_temporary_buffer`, then it must deallocate that memory, using `return_temporary_buffer` , before it returns. **Note:** `get_temporary_buffer` and `return_temporary_buffer` are only provided for backward compatibility. If you are writing new code, you should instead use the `temporary_buffer` class.

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

Preconditions

- `len` is greater than `0`.

Complexity

Example

```
int main()
{
  pair<int*, ptrdiff_t> P = get_temporary_buffer(10000, (int*) 0);
  int* buf = P.first;
  ptrdiff_t N = P.second;
  uninitialized_fill_n(buf, N, 42);
  int* result = find_if(buf, buf + N, bind2nd(not_equal_to<int>(), 42));
  assert(result == buf + N);
  return_temporary_buffer(buf);
}
```

Notes

If `P.second` is 0, this means that `get_temporary_buffer` was unable to allocate a temporary buffer at all. In that case, `P.first` is a null pointer. It is unspecified whether `get_temporary_buffer` is implemented using `malloc`, or `::operator new`, or some other method. The only portable way to return memory that was allocated using `get_temporary_buffer` is to use `return_temporary_buffer`.

See also

`temporary_buffer`, `return_temporary_buffer`, Allocators

12.2.6 return_temporary_buffer

Prototype

```
template <class T> void return_temporary_buffer(T* p);
```

Description

Return_temporary_buffer is used to deallocate memory that was allocated using get_temporary_buffer. **Note:** get_temporary_buffer and return_temporary_buffer are only provided for backward compatibility. If you are writing new code, you should instead use the temporary_buffer class.

Definition

Defined in the standard header memory, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

Preconditions

The argument p is a pointer to a block of memory that was allocated using get_temporary_buffer(ptrdiff_t, T*).

Complexity

Example

```
int main()
{
  pair<int*, ptrdiff_t> P = get_temporary_buffer(10000, (int*) 0);
  int* buf = P.first;
  ptrdiff_t N = P.second;
  uninitialized_fill_n(buf, N, 42);
  int* result = find_if(buf, buf + N, bind2nd(not_equal_to<int>(), 42));
  assert(result == buf + N);
  return_temporary_buffer(buf);
}
```

Notes

As is always true, memory that was allocated using a particular allocation function must be deallocated using the corresponding deallocation function. Memory obtained using get_temporary_buffer must be deallocated using return_temporary_buffer, rather than using free or ::operator delete.

See also

temporary_buffer, get_temporary_buffer, Allocators