Rogue Wave Standard C++ Library Class Reference

Rogue Wave Software
Corvallis, Oregon USA

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

for

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

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

User's Guide and Tutorial Author: Timothy A. Budd

Class Reference Authors: Wendi Minne, Tom Pearson, and Randy Smithey

Product Team:

Development:	Anna Dahan, Philippe Le Mouel, Randy Smithey
Quality Engineering:	Kevin Djang, Randall Robinson
Manuals:	Elaine Cull, Wendi Minne, Julie Prince, Randy Smithey
Support:	North Krimsley

Significant contributions by: Joe Delaney

Part # RW81-01-100096

Printing Date: October, 1996

Rogue Wave Software, Inc., 850 SW 35th St., Corvallis, Oregon, 97333 USA

Product Information:	**(541) 754-3010**
	(800) 487-3217
Technical Support:	**(541) 754-2311**
FAX:	**(541) 757-6650**
World Wide Web:	**http://www.roguewave.com**

Class Reference

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

The tables on the next few pages list the contents organized by category.

For each class, the entry begins with a brief summary of the class; a synopsis, which indicates the header file(s); and the signature of a class object. The entry continues with a text description of the class followed by the C++ code that describes the class interface. Next, all methods associated with a class, including constructors, operators, member functions, etc., are grouped in categories according to their general use and described. The categories are not a part of the C++ language, but do provide a way of organizing the methods. Following the member function descriptions, many of the classes include examples. Finally, any warnings associated with using the class are described.

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

Standards Conformance

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

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

	replace_copy replace_copy_if replace_if reverse reverse_copy rotate rotate_copy search search_n set_difference set_intersection set_symmetric_difference set_union sort sort_heap stable_partition stable_sort swap swap_ranges transform unique unique_copy upper_bound

Complex Number Library `#include <complex>`	complex

Containers
`#include <bitset>`
`#include <deque>`
`#include <list>`
`#include <map>` for map and multimap
`#include <queue>` for queue and priority_queue
`#include <set>` for set and multiset
`#include <stack>`
`#include <vector>`
	bitset
	deque
	list
	map
	multimap
	multiset
	priority_queue
	queue
	set
	stack
	vector

Function Adaptors `#include <functional>`	bind1st bind2nd not1 not2 ptr_fun

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

Generalized Numeric Operations `#include <numeric>`	accumulate adjacent_difference accumulate inner_product partial_sum

Insert Iterators `#include <iterator>`	back_insert_iterator back_inserter front_insert_iterator front_inserter insert_iterator inserter

Iterators #include <iterator>	bidirectional iterator forward iterator input iterator output iterator random access iterator reverse_bidirectional_iterator reverse_iterator

Iterator operations #include <iterator>	advance distance

Memory Handling Primitives #include <memory>	get_temporary_buffer return_temporary_buffer

Memory Management #include <memory>	allocator auto_ptr raw_storage_iterator uninitialized_copy uninitialized_fill uninitialized_fill_n

Numeric Limits Library #include <limits>	numeric limits

String Library #include <string>	basic_string string wstring

Utility Classes #include <utility>	pair

Utility Operators #include <utility>	operator!= operator> operator<= operator>=

accumulate

Generalized Numeric Operation

Summary

Accumulate all elements within a range into a single value.

Synopsis

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

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

Description

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

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

Complexity

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

Example

```
//
// accum.cpp
//
 #include <numeric>   //for accumulate
 #include <vector>    //for vector
 #include <functional> //for times
 #include <iostream.h>

 int main()
 {
   //
   //Typedef for vector iterators
   //
   typedef vector<int>::iterator iterator;
   //
   //Initialize a vector using an array of ints
   //
```

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

    cout << " where N = 10." << endl;
    cout << "The sum = (N*N + N)/2 = " << sum << endl;
    cout << "The product = N! = " << prod << endl;
    return 0;
  }
 Output :
 For the series: 1 2 3 4 5 6 7 8 9 10  where N = 10.
 The sum = (N*N + N)/2 = 55
 The product = N! = 3628800
```

Warnings

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

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

instead of:

```
vector<int>
```

Summary

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

Synopsis

```
#include <numeric>

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

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

Description

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

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

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

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

or

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

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

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

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

Complexity

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

Example

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

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

Warning

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

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

instead of:

```
vector<int>
```

adjacent_find

Algorithm

Summary

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

Synopsis

```
#include <algorithm>

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

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

Description

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

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

or

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

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

Complexity

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

Example

```
//
// find.cpp
//
#include <vector>
#include <algorithm>
#include <iostream.h>

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

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

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

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

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

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

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

    return 0;
  }

Output :
3 3 2 2
```

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

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

instead of:

```
vector<int>
```

See Also ***find***

advance

Iterator Operation

Summary

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

Synopsis

```
#include <iterator>

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

Description

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

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

Example

```
//
// advance.cpp
//
 #include<iterator>
 #include<list>
 #include<iostream.h>

 int main()
 {

   //
   //Initialize a list using an array
   //
   int arr[6] = {3,4,5,6,7,8};
   list<int> l(arr,arr+6);
   //
   //Declare a list iterator, s.b. a ForwardIterator
   //
   list<int>::iterator itr = l.begin();
   //
   //Output the original list
   //
   cout << "For the list: ";
   copy(l.begin(),l.end(),
        ostream_iterator<int,char>(cout," "));
```

```
    cout << endl << endl;
    cout << "When the iterator is initialized to l.begin(),"
         << endl << "it points to " << *itr << endl << endl;
    //
    // operator+ is not available for a ForwardIterator,
    // so use advance.
    //

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

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

Warnings

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

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

instead of:

```
vector<int>
```

See Also

sequence, random_iterator, distance

Algorithms

Summary

Generic algorithms for performing various operations on containers and sequences.

Synopsis

```
#include <algorithm>
```

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

Description

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

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

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

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

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

Algorithms by Mutating/Non-mutating Function

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

Non-mutating operations

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

Mutating operations

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

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

Algorithms by Operation

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

Initializing operations

`fill`	`generate`
`fill_n`	`generate_n`

Search operations

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

Binary search operations (Elements must be sorted)

`binary_search`	`lower_bound`
`equal_range`	`upper_bound`

Compare operations

`equal`	`mismatch`
`lexicographical_compare`	

Copy operations

`copy`	`copy_backward`

Transforming operations

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

Swap operations

`swap`	`swap_ranges`

Scanning operations

`accumulate`	`for_each`

Remove operations

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

Sorting operations

`nth_element`	`sort`
`partial_sort`	`stable_sort`
`partial_sort_copy`	

Merge operations (Elements must be sorted)

`inplace_merge`	`merge`

Set operations (Elements must be sorted)

`includes`	`set_symmetric_difference`
`set_difference`	`set_union`
`set_intersection`	

Heap operations

`make_heap`	`push_heap`
`pop_heap`	`sort_heap`

Minimum and maximum

`max`	`min`
`max_element`	`min_element`

Permutation generators

`next_permutation`	`prev_permutation`

Algorithms by Category

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

Algorithms that use no iterators:

`max`	`min`	`swap`

Algorithms that require only input iterators:

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

Algorithms that require only output iterators:

fill_n	generate_n

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

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

Algorithms that require forward iterators:

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

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

`rotate_copy`

Algorithms that require bidirectional iterators

copy_backward	partition
inplace_merge	prev_permutation
next_permutation	reverse
	stable_permutation

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

`reverse_copy`

Algorithms that require random access iterators:

make_heap	pop_heap	sort
nth_element	push_heap	sort_heap
partial_sort	random_shuffle	stable_sort

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

`partial_sort_copy`

Complexity

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

See Also

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

Summary

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

Synopsis

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

Description

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

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

The Alternate Allocator

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

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

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

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

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

The Standard allocator interface solves the problem with member templates. The precise type you are going to construct is not specified when you create an allocator, but when you actually go to allocate space or construct an object on existing space. This clever solution is well ahead of nearly all existing compiler implementations.

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

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

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

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

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

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

Standard Interface

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

  template <class U> struct rebind;
  allocator () throw();
  template <class U> allocator(const allocator<U>&) throw();
  template <class U>
    allocator& operator=(const allocator<U>&) throw();
  ~allocator () throw();
  pointer  address (reference) const;
  const_pointer address (const_reference) const;
  pointer allocate (size_type,
     typename allocator<void> const_pointer = 0);
  void deallocate(pointer);
  size_type max_size () const;
  void construct (pointer, const T&);
  void destroy (pointer);
};


// specialize for void:
  template <> class allocator<void> {
  public:
    typedef size_t      size_type;
    typedef ptrdiff_t   difference_type;
    typedef void*       pointer;
    typedef const void* const_pointer;
    //  reference-to-void members are impossible.
    typedef void  value_type;
    template <class U>
```

```
      struct rebind { typedef allocator<U> other; };

    allocator() throw();
    template <class U>
      allocator(const allocator<U>&) throw();
    template <class U>
      allocator operator=(const allocator<U>&) throw();
   ~allocator() throw();

    pointer allocate(size_type, const void* hint);
    void deallocate(pointer p);
    size_type max_size() const throw();
  };

// globals
template <class T>
  void* operator new(size_t N, allocator<T>& a);
template <class T, class U>
  bool operator==(const allocator<T>&,
                  const allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const allocator<T>&,
                  const allocator<U>&) throw();
```

Types

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

`difference_type`
Type used to hold values representing distances between storage addresses.

`pointer`
Type of pointer returned by allocator.

`const_pointer`
`Const` version of `pointer`.

`reference`
Type of reference to allocated objects.

`const_reference`
`Const` version of `reference`.

`value_type`
Type of allocated object.

`template <class U> struct rebind;`
Provides a way to convert an allocator templated on one type to an allocator templated on another type. This struct contains a single type member: `typedef allocator<U> other`.

Operations

```
allocator()
```
Default constructor.

```
template <class U>
allocator(const allocator<U>&)
```
Copy constructor.

```
template <class U>
allocator& operator=(const allocator<U>&) throw()>&)
```
Assignment operator.

```
~allocator()
```
Destructor.

```
pointer address(reference x) const;
```
Returns the address of the reference `x` as a pointer.

```
const_pointer address(const_reference x) const;
```
Returns the address of the reference `x` as a `const_pointer`.

```
pointer allocate(size_type n,
    typename allocator<void>::const_pointer p = 0)
```
Allocates storage. Returns a pointer to the first element in a block of storage `n*sizeof(T)` bytes in size. The block will be aligned appropriately for objects of type `T`. Throws the exception `bad_alloc` if the storage is unavailable. This function uses operator `new(size_t)`. The second parameter `p` can be used by an allocator to localize memory allocation, but the default allocator does not use it.

```
void deallocate(pointer p)
```
Deallocates the storage indicated by `p`. The storage must have been obtained by a call to `allocate`.

```
size_type max_size () const;
```
Returns the largest size for which a call to `allocate` might succeed.

```
void construct (pointer p, const T& val);
```
Constructs an object of type `T2` with the initial value of `val` at the location specified by `p`. This function calls the `placement new` operator.

```
void destroy (pointer p)
```
Calls the destructor on the object pointed to by `p`, but does not delete.

Alternate Interface

```
class allocator
{
public:
typedef size_t                size_type ;
typedef ptrdiff_t             difference_type ;
 allocator ();
```

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

protected:
   allocator_type*     alloc_;

public:
   allocator_interface ();
   allocator_interface (Allocator*);
   void alloc (Allocator*);
   pointer address (T& x);
   size_type max_size () const;
   pointer allocate (size_type, pointer = 0);
   void deallocate (pointer);
   void construct (pointer, const T&);
   void destroy (T*);
};
//
// Specialization
//
class allocator_interface <allocator,void>
 {
 typedef void*                 pointer ;
 typedef const void*           const_pointer ;
 };
```

Alternate Allocator Description

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

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

See Also

container

associative containers

Summary

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

See Also

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

Summary

A simple, smart pointer class.

Synopsis

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

Description

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

Interface

```
template <class X> class auto_ptr {

   public:

     // constructor/copy/destroy

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

     // members

     X& operator* () const throw();
     X* operator-> () const throw();
     X* get () const throw();
     X* release () throw();
 };
```

Constructors and Destructors

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

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

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

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

`~auto_ptr ();`

Deletes the underlying pointer.

Operators

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

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

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

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

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

Returns the underlying pointer.

Member Functions

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

Returns the underlying pointer.

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

Releases ownership of the underlying pointer. Returns that pointer.

Example

```
//
// auto_ptr.cpp
//
#include <iostream.h>
#include <memory>

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

int main ()
{
   //
   // b will hold a pointer to an X.
   //
   auto_ptr<X> b(new X(12345));
```

```
    //
    // a will now be the owner of the underlying pointer.
    //
    auto_ptr<X> a = b;
    //
    // Output the value contained by the underlying pointer.
    //
    cout << a->get() << endl;
    //
    // The pointer will be deleted when a is destroyed on
    // leaving scope.
    //
    return 0;
  }

Output :
12345
```

Summary

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

Synopsis

```
#include <iterator>

template <class Container>
class back_insert_iterator : public output_iterator;
```

Description

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

Interface

```
template <class Container>
 class back_insert_iterator : public output_iterator {

protected:
   Container& container;
public:
   back_insert_iterator (Container&);
   back_insert_iterator<Container>&
    operator= (const Container::value_type&);
   back_insert_iterator<Container>& operator* ();
   back_insert_iterator<Container>& operator++ ();
   back_insert_iterator<Container> operator++ (int);
};

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

Constructor

`back_insert_iterator (Container& x);`

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

Operators

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

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

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

Returns `*this`.

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

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

Increments the input iterator and returns `*this`.

Helper Function

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

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

Example

```
//
// ins_itr.cpp
//
 #include <iterator>
 #include <deque>
 #include <iostream.h>

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

```
    cout << "Use a front_inserter: " << endl << "     ";
    copy(d.begin(), d.end(),
         ostream_iterator<int,char>(cout," "));
    //
    // Insert d2 at back of d.
    //
    copy(d2.begin(), d2.end(), back_inserter(d));
    //
    // Output the new deque.
    //
    cout << endl << endl;
    cout << "Use a back_inserter: " << endl << "     ";
    copy(d.begin(), d.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl;

    return 0;
  }

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

Warning

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

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

instead of:

```
vector<int>
```

See Also

insert iterators

Summary

A templated class for handling sequences of character-like entities. ***string*** and ***wstring*** are specialized versions of ***basic_string*** for `char`s and `wchar_t`s, respectively.

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


Synopsis

```
#include <string>

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >

class basic_string;
```

Description

basic_string<charT, traits, Allocator> is a homogeneous collection of character-like entities. It provides general string functionality such as compare, append, assign, insert, remove, and replace , along with various searches. ***basic_string*** also functions as an STL sequence container, providing random access iterators. This allows some of the generic algorithms to apply to strings.

Any underlying character-like type may be used as long as an appropriate `string_char_traits` class is provided or the default `traits` class is applicable.

Interface

```
template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
class basic_string {

public:

// Types

typedef traits                              traits_type;
typedef typename traits::char_type          value_type;
typedef Allocator                           allocator_type;

typename size_type;
typename difference_type;
typename reference;
typename const_reference;
typename pointer;
typename const_pointer;
typename iterator;
```

```
typename const_iterator;
typename const_reverse_iterator;
typename reverse_iterator;

static const size_type npos = -1;

// Constructors/Destructors

explicit basic_string(const Allocator& = Allocator());
basic_string (const basic_string<charT, traits, Allocator>&);
basic_string(const basic_string&, size_type, size_type = npos);
basic_string(const charT*, size_type,
             const Allocator& = Allocator());
basic_string(const charT*, Allocator& = Allocator());
basic_string(size_type, charT,
             const Allocator& = Allocator());
template <class InputIterator>
basic_string(InputIterator, InputIterator,
             const Allocator& = Allocator());
~basic_string();

// Assignment operators
 basic_string& operator=(const basic_string&);
 basic_string& operator=(const charT*);
 basic_string& operator=(charT);

// Iterators

 iterator       begin();
 const_iterator begin() const;
 iterator       end();
 const_iterator end() const;

 reverse_iterator       rbegin();
 const_reverse_iterator rbegin() const;
 reverse_iterator       rend();
 const_reverse_iterator rend() const;

// Capacity

   size_type       size() const;
   size_type       length() const;
   size_type       max_size() const;
   void            resize(size_type, charT);
   void            resize(size_type);
   size_type       capacity() const;
   void            reserve(size_type);
   bool            empty() const;

// Element access

   const_reference operator[](size_type) const;
   reference       operator[](size_type);
   const_reference at(size_type) const;
   reference       at(size_type);

// Modifiers
```

```
basic_string& operator+=(const basic_string&);
basic_string& operator+=(const charT*);
basic_string& operator+=(charT);

basic_string& append(const basic_string&);
basic_string& append(const basic_string&,
                     size_type, size_type);
basic_string& append(const charT*, size_type);
basic_string& append(const charT*);
basic_string& append(size_type, charT);
template<class InputIterator>
 basic_string& append(InputIterator, InputIterator);

basic_string& assign(const basic_string&);
basic_string& assign(const basic_string&,
                     size_type, size_type);
basic_string& assign(const charT*, size_type);
basic_string& assign(const charT*);
basic_string& assign(size_type, charT);
template<class InputIterator>
 basic_string& assign(InputIterator, InputIterator);

basic_string& insert(size_type, const basic_string&);
basic_string& insert(size_type, const basic_string&,
                     size_type, size_type);
basic_string& insert(size_type, const charT*, size_type);
basic_string& insert(size_type, const charT*);
basic_string& insert(size_type, size_type, charT);
iterator insert(iterator, charT = charT());
void insert(iterator, size_type, charT);
template<class InputIterator>
 void insert(iterator, InputIterator,
            InputIterator);

basic_string& erase(size_type = 0, size_type= npos);
iterator erase(iterator);
iterator erase(iterator, iterator);

basic_string& replace(size_type, size_type,
                      const basic_string&);
basic_string& replace(size_type, size_type,
                      const basic_string&,
                      size_type, size_type);
basic_string& replace(size_type, size_type,
                      const charT*, size_type);
basic_string& replace(size_type, size_type,
                      const charT*);
basic_string& replace(size_type, size_type,
                      size_type, charT);
basic_string& replace(iterator, iterator,
                      const basic_string&);
basic_string& replace(iterator, iterator,
                      const charT*, size_type);
basic_string& replace(iterator, iterator,
                      const charT*);
basic_string& replace(iterator, iterator,
                      size_type, charT);
template<class InputIterator>
```

```
 basic_string& replace(iterator, iterator,
                       InputIterator, InputIterator);

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

// String operations

 const charT* c_str() const;
 const charT* data() const;
 const allocator_type& get_allocator() const;

 size_type find(const basic_string&,
                size_type = 0) const;
 size_type find(const charT*,
                size_type, size_type) const;
 size_type find(const charT*, size_type = 0) const;
 size_type find(charT, size_type = 0) const;
 size_type rfind(const basic_string&,
                 size_type = npos) const;
 size_type rfind(const charT*,
                 size_type, size_type) const;
 size_type rfind(const charT*,
                 size_type = npos) const;
 size_type rfind(charT, size_type = npos) const;

 size_type find_first_of(const basic_string&,
                         size_type = 0) const;
 size_type find_first_of(const charT*,
                         size_type, size_type) const;
 size_type find_first_of(const charT*,
                         size_type = 0) const;
 size_type find_first_of(charT, size_type = 0) const;

 size_type find_last_of(const basic_string&,
                        size_type = npos) const;
 size_type find_last_of(const charT*,
                        size_type, size_type) const;
 size_type find_last_of(const charT*, size_type = npos) const;
 size_type find_last_of(charT, size_type = npos) const;

 size_type find_first_not_of(const basic_string&,
                             size_type = 0) const;
 size_type find_first_not_of(const charT*,
                             size_type, size_type) const;
 size_type find_first_not_of(const charT*, size_type = 0) const;
 size_type find_first_not_of(charT, size_type = 0) const;

 size_type find_last_not_of(const basic_string&,
                            size_type = npos) const;
 size_type find_last_not_of(const charT*,
                            size_type, size_type) const;
 size_type find_last_not_of(const charT*,
                            size_type = npos) const;
 size_type find_last_not_of(charT, size_type = npos) const;

 basic_string substr(size_type = 0, size_type = npos) const;
 int compare(const basic_string&) const;
```

```
   int compare(size_type, size_type, const basic_string&) const;
   int compare(size_type, size_type, const basic_string&,
               size_type, size_type) const;
   int compare(size_type, size_type, charT*) const;
   int compare(charT*) const;
   int compare(size_type, size_type, const charT*, size_type)
const;
};

// Non-member Operators

template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&,
                        const basic_string&);
template <class charT, class traits, class Allocator>
 basic_string operator+ (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 basic_string operator+ (charT, const basic_string&);
template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&, const charT*);
template <class charT, class traits, class Allocator>
 basic_string operator+ (const basic_string&, charT);

template <class charT, class traits, class Allocator>
 bool operator== (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator== (const charT*, const basic_string&);
template <class charT, class traits , class Allocator>
 bool operator== (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator< (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator< (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator< (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator!= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator!= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator!= (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator> (const basic_&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator> (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator> (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
 bool operator<= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator<= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator<= (const basic_string&, const charT*);
```

```
template <class charT, class traits, class Allocator>
 bool operator>= (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator>= (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
 bool operator>= (const basic_string&, const charT*);

template <class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& a,
          basic_string<charT,traits,Allocator>& b);

template<class charT, class traits, class Allocator>
 istream& operator>> (istream&, basic_string&);
template <class charT, class traits, class Allocator>
 ostream& operator<< (ostream&, const basic_string&);
template <class Stream, class charT,
          class traits, class Allocator>
 Stream& getline (Stream&, basic_string&, charT);
```

Constructors and Destructors

In all cases, the `Allocator` parameter will be used for storage management.

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

The default constructor. Creates a ***basic_string*** with the following effects:

`data()`	a non-null pointer that is copyable and can have 0 added to it
`size()`	0
`capacity()`	an unspecified value

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

Copy constructor. Creates a string that is a copy of `str`.

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

Creates a string if `pos<=size()` and determines length `rlen` of initial string value as the smaller of `n` and `str.size() - pos`. This has the following effects:

`data()`	points at the first element of an allocated copy of `rlen` elements of the string controlled by `str` beginning at position `pos`
`size()`	`rlen`
`capacity()`	a value at least as large as `size()`
`get_allocator()`	`str.get_allocator()`

An `out_of_range` exception will be thrown if `pos>str.size()`.

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

Creates a string that contains the first `n` characters of `s`. `s` must not be a `NULL` pointer. The effects of this constructor are:

`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

An `out_of_range` exception will be thrown if `n == npos`.

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

Constructs a string containing all characters in `s` up to, but not including, a `traits::eos()` character. `s` must not be a null pointer. The effects of this constructor are:

`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`traits::length(s)`
`capacity()`	a value at least as large as `size()`

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

Constructs a string containing `n` repetitions of `c`. A `length_error` exception is thrown if `n == npos`. The effects of this constructor are:

`data()`	points at the first element of an allocated array of `n` elements, each storing the initial value `c`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

```
template <class InputIterator>
basic_string  (InputIterator first, InputIterator last,
               const Allocator& a = Allocator());
```

Creates a ***basic_string*** of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The effects of this constructor are:

`data()`	points at the first element of an allocated copy of the elements in the range `[first,last)`
`size()`	distance between `first` and `last`
`capacity()`	a value at least as large as `size()`

```
~basic_string ();
```

Releases any allocated memory for this ***basic_string***.

Operators

```
basic_string&
operator= (const basic_string& str);
```

Assignment operator. Sets the contents of this string to be the same as `str`. The effects of `operator=` are:

`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `str.size()`
`size()`	`str.size()`
`capacity()`	a value at least as large as `size()`

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

Assignment operator. Sets the contents of this string to be the same as `s` up to, but not including, the `traits::eos()` character.

```
basic_string&
operator= (charT c);
```

Assignment operator. Sets the contents of this string to be equal to the single `charT c`.

```
const_reference
operator[] (size_type pos) const;
reference
operator[] (size_type pos);
```

If `pos < size()`, returns the element at position `pos` in this string. If `pos == size()`, the `const` version returns `traits::eos()`, the behavior of the non-`const` version is undefined. The reference returned by either version is invalidated by any call to `c_str()`, `data()`, or any non-`const` member function for the object.

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

Concatenates a string onto the current contents of this string. The second member operator uses `traits::length()` to determine the number of elements from `s` to add. The third member operator adds the single character `c`. All return a reference to this string after completion.

Iterators

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

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

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

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

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

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

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

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

Allocator

```
const allocator_type get_allocator () const;
```

Returns a copy of the allocator used by self for storage management.

Member Functions

```
basic_string&
append (const basic_string& s, size_type pos, size_type npos);
basic_string&
append (const basic_string& s);
basic_string&
append (const charT* s, size_type n);
basic_string&
append (const charT* s);
basic_string&
append (size_type n, charT c );
template<class InputIterator>
basic_string&
append (InputIterator first, InputIterator last);
```

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

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

```
basic_string&
assign (const basic_string& s);
basic_string&
assign (const  basic_string& s,
       size_type pos, size_type n);
basic_string&
assign (const charT* s, size_type n);
basic_string&
assign (const charT* s);
basic_string&
assign (size_type n, charT c );
template<class InputIterator>
basic_string&
assign (InputIterator first, InputIterator last);
```

Replace the value of this string with the value of another.

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

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

```
const_reference
at (size_type pos) const;
reference
at (size_type pos);
```

If `pos < size()`, returns the element at position `pos` in this string. Otherwise, an `out_of_range` exception is thrown.

```
size_type
capacity () const;
```

Returns the current storage capacity of the string. This is guaranteed to be at least as large as `size()`.

```
int
compare (const basic_string& str);
```

Returns the result of a lexicographical comparison between elements of this string and elements of `str`. The return value is:

```
<0    if size() < str.size()
0     if size() == str.size()
>0    if size() > str.size()
```

```
int
compare (size_type pos1, size_type n1,
         const basic_string& str) const;
int
compare (size_type pos1, size_type n1, const basic_string& str,
         size_type pos2, size_type n2) const;
int
compare (charT* s) const;
int
compare (size_type pos, size_type n1, charT* s) const;
int
compare (size_type pos, size_type n1, charT* s,
         size_type n2) const;
```

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

```
compare (str)
compare (basic_string (str, pos2, n2))
compare (basic_string(s))
compare (basic_string(s, npos))
compare (basic_string (s,n2))
```

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

Replaces elements in memory with copies of elements from this string. An `out_of_range` exception will be thrown if `pos > size()`. The lesser of `n` and `size() - pos` elements of this string, starting at position `pos` are copied into the array pointed to by `s`. No terminating null is appended to `s`.

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

Return a pointer to the initial element of an array whose first `size()` elements are copies of the elements in this string. A `traits::eos()` element is appended to the end. The elements of the array may not be altered, and the returned pointer is only valid until a non-`const` member function of this string is called. If `size()` is zero, the `data()` function returns a `NULL` pointer.

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

Returns `size() == 0`.

```
basic_string&
erase (size_type pos = 0, size_type n = npos);
iterator
erase (iterator p);
iterator
erase (iterator first, iterator last);
```

This function removes elements from the string, collapsing the remaining elements, as necessary, to remove any space left empty. The first version of the function removes the smaller of `n` and `size() - pos` starting at position `pos`. An `out_of_range` exception will be thrown if `pos > size()`. The second version requires that `p` is a valid iterator on this string, and removes the character referred to by `p`. The last version of `erase` requires that both `first` and `last` are valid iterators on this string, and removes the characters defined by the range `[first, last)`. The destructors for all removed characters are called. All versions of `erase` return a reference to this string after completion.

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

Searches for the first occurrence of the substring specified by `str` in this string, starting at position `pos`. If found, it returns the index of the first character of the matching substring. If not found, returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find (const charT* s, size_type pos, size_type n) const;
size_type
find (const charT* s, size_type pos = 0) const;
size_type
find (charT c, size_type pos = 0) const;
```

Search for the first sequence of characters in this string that match a specified string. The variations of this function return, respectively:

```
find(basic_string(s,n), pos)
find(basic_string(s), pos)
find(basic_string(1, c), pos)
```

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

Searches for the first element of this string at or after position `pos` that is not equal to any element of `str`. If found, `find_first_not_of` returns the index of the non-matching character. If all of the characters match, the function returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_first_not_of  (const charT* s,
                    size_type pos, size_type n) const;
size_type
find_first_not_of (const charT* s,
                   size_type pos = 0) const;
size_type
find_first_not_of (charT c, size_type pos = 0) const;
```

Search for the first element in this string at or after position `pos` that is not equal to any element of a given set of characters. The members return, respectively:

```
find_first_not_of(basic_string(s,n), pos)
find_first_not_of(basic_string(s), pos)
find_first_not_of(basic_string(1, c), pos)
```

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

Searches for the first occurrence at or after position `pos` of any element of `str` in this string. If found, the index of this matching character is returned. If not found, `npos` is returned. Equality is defined by `traits::eq()`.

```
size_type
find_first_of(const charT*  s,  size_type  pos,
              size_type n) const;
size_type
find_first_of(const charT* s, size_type pos = 0) const;
size_type
find_first_of (charT c, size_type pos = 0) const;
```

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

```
find_first_of(basic_string(s,n), pos)
find_first_of(basic_string(s), pos)
find_first_of(basic_string(1, c), pos)
```

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

Searches for the last element of this string at or before position `pos` that is not equal to any element of `str`. If `find_last_not_of` finds a non-matching element, it returns the index of the character. If all the elements match, the function returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_last_not_of(const charT* s,
                 size_type pos, size_type n) const;
```

```
size_type
find_last_not_of(const charT* s, size_type pos = npos) const;
```

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

Search for the last element in this string at or before position `pos` that is not equal to any element of a given set of characters. The members return, respectively:

```
find_last_not_of(basic_string(s,n), pos)
find_last_not_of(basic_string(s), pos)
find_last_not_of(basic_string(1, c), pos)
```

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

Searches for the last occurrence of any element of `str` at or before position `pos` in this string. If found, `find_last_of` returns the index of the matching character. If not found `find_last_of` returns `npos`. Equality is defined by `traits::eq()`.

```
size_type
find_last_of(const charT* s, size_type pos,
             size_type n) const;
size_type
find_last_of(const charT* s, size_type pos = npos) const;
size_type
find_last_of(charT c, size_type pos = npos) const;
```

Search for the last occurrence in this string of any element in a specified string. The members return, respectively:

```
find_last_of(basic_string(s,n), pos)
find_last_of(basic_string(s), pos)
find_last_of(basic_string(1, c), pos)
```

```
basic_string&
insert(size_type pos1, const basic_string& s);
basic_string&
insert(size_type pos, const  basic_string& s,
       size_type pos2 = 0, size_type n = npos);
basic_string&
insert(size_type pos, const charT* s, size_type n);
basic_string&
insert(size_type pos, const charT* s);
basic_string&
insert(size_type pos, size_type n, charT c);
```

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

The second variation of this function inserts the lesser of `n` and `s.size() - pos2` characters of `s`, beginning at position `pos2` in this string. This version will throw an `out_of_range` exception if `pos2 > s.size()`. The third version inserts `n` characters of the array pointed to by `s`. The fourth inserts elements from the array pointed to by `s` up to, but not including, a `traits::eos()` character. Finally, the fifth variation inserts `n` repetitions of `c`.

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

Insert additional elements in this string immediately before the character referred to by `p`. All of these versions of `insert` require that `p` is a valid iterator on this string. The first version inserts a copy of `c`. The second version inserts `n` repetitions of `c`. The third version inserts characters in the range `[first, last)`. The first version returns `p`.

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

Return the number of elements contained in this string.

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

Returns the maximum possible size of the string.

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

Searches for the last occurrence of the substring specified by `str` in this string, starting at position `pos`. Note that only the first character of the substring must be `<= pos;` the remaining characters may extend beyond `pos`. If found, the index of the first character of that matches substring is returned. If not found, `npos` is returned. Equality is defined by `traits::eq()`.

```
size_type
rfind(const charT* s, size_type pos,  size_type n) const;
```

```
size_type
rfind(const charT* s, size_type pos = npos) const;
```

```
size_type
rfind(charT c, size_type pos = npos) const;
```

Searches for the last sequence of characters in this string matching a specified string. The `rfind` variations return, respectively:

```
rfind(basic_string(s,n), pos)
rfind(basic_string(s), pos)
rfind(basic_string(1, c), pos)
```

```
basic_string&
replace(size_type pos, size_type n1, const basic_string& s);
```

```
basic_string&
replace(size_type pos1, size_type n1, const basic_string& str,
        size_type pos2, size_type n2);
```

```
basic_string&
replace(size_type pos, size_type n1, const charT* s,
        size_type n2);
```

```
basic_string&
replace(size_type pos, size_type n1, const charT* s);
```

```
basic_string&
replace(size_type pos, size_type n1, size_type n2, charT c);
```

The `replace` function replaces selected elements of this string with an alternate set of elements. All of these versions insert the new elements in place of `n1` elements in this string, starting at position `pos`. They each throw an `out_of_range` exception if `pos1 > size()`and a `length_error` exception if the resulting string size exceeds `max_size()`.

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

```
basic_string&
replace(iterator i1, iterator i2,
        const basic_string& str);
basic_string&
replace(iterator i1, iterator i2, const charT* s,
        size_type n);
basic_string&
replace(iterator i1, iterator i2, const charT* s);
basic_string&
replace(iterator i1, iterator i2, size_type n,
        charT c);
template<class InputIterator>
basic_string&
replace(iterator  i1,  iterator  i2,
        InputIterator j1, InputIterator j2);
```

Replace selected elements of this string with an alternative set of elements. All of these versions of replace require iterators i1 and i2 to be valid iterators on this string. The elements specified by the range [i1, i2) are replaced by the new elements.

The first version shown here replaces with all members in str. The second version starts at position i1, and replaces the next n characters with n characters of the array pointed to by s. The third variation replaces string elements with elements from the array pointed to by s up to, but not including, a traits::eos() character. The fourth version replaces string elements with n repetitions of c. The last variation shown here replaces string elements with the members specified in the range [j1, j2).

```
void
reserve(size_type res_arg);
```

Assures that the storage capacity is at least res_arg.

```
void
resize(size_type n, charT c);
void
resize(size_type n);
```

Changes the capacity of this string to n. If the new capacity is smaller than the current size of the string, then it is truncated. If the capacity is larger, then the string is padded with c characters. The latter resize member pads the string with default characters specified by traits::eos().

```
size_type
size() const;
```

Return the number of elements contained in this string.

```
basic_string
substr(size_type pos = 0, size_type n = npos) const;
```

Returns a string composed of copies of the lesser of `n` and `size()` characters in this string starting at index `pos`. Throws an out_of_range exception if `pos <= size().`

```
void
swap(basic_string& s);
```

Swaps the contents of this string with the contents of `s`.

Non-member Operators

```
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string& lhs, const basic_string& rhs);
```

Returns a string of length `lhs.size() + rhs.size()`, where the first `lhs.size()` elements are copies of the elements of `lhs`, and the next `rhs.size()` elements are copies of the elements of `rhs`.

```
template<class charT, class traits, class Allocator>
basic_string
operator+(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(charT lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string&  lhs, const charT* rhs);
template<class charT, class traits, class Allocator>
basic_string
operator+(const basic_string& lhs, charT rhs);
```

Returns a string that represents the concatenation of two string-like entities. These functions return, respectively:

```
basic_string(lhs) + rhs
basic_string(1, lhs) + rhs
lhs + basic_string(rhs)
lhs + basic_string(1, rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator==(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value of `true` if `lhs` and `rhs` are equal, and `false` if they are not. Equality is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator==(const charT* lhs, const basic_string& rhs);
```

```
template<class charT, class traits, class Allocator>
bool
operator==(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value indicating whether `lhs` and `rhs` are equal. Equality is defined by the `compare()` member function. These functions return, respectively:

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

```
template<class charT, class traits, class Allocator>
bool
operator!=(const basic_string& lhs,
           const basic_string& rhs);
```

Returns a boolean value representing the inequality of `lhs` and `rhs`. Inequality is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator!=(const charT* lhs, const basic_string& rhs);
```

```
template<class charT, class traits, class Allocator>
bool
operator!=(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the inequality of `lhs` and `rhs`. Inequality is defined by the `compare()` member function. The functions return, respectively:

```
basic_string(lhs) != rhs
lhs != basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator<(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value representing the lexicographical less-than relationship of `lhs` and `rhs`. Less-than is defined by the `compare()` member.

```
template<class charT, class traits, class Allocator>
bool
operator<(const charT* lhs, const basic_string& rhs);
```

```
template<class charT, class traits, class Allocator>
bool
operator<(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical less-than relationship of `lhs` and `rhs`. Less-than is defined by the `compare()` member function. These functions return, respectively:

```
basic_string(lhs) < rhs
lhs < basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator>(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value representing the lexicographical greater-than relationship of `lhs` and `rhs`. Greater-than is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator>(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator>(const basic_string& lhs, const charT* rhs);
```

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

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

```
template<class charT, class traits, class Allocator>
bool
operator<=(const basic_string& lhs,
              const basic_string& rhs);
```

Returns a boolean value representing the lexicographical less-than-or-equal relationship of `lhs` and `rhs`. Less-than-or-equal is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator<=(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator<=(const basic_string& lhs, const charT* rhs);
```

Returns a boolean value representing the lexicographical less-than-or-equal relationship of `lhs` and `rhs`. Less-than-or-equal is defined by the `compare()` member function. These functions return, respectively:

```
basic_string(lhs) <= rhs
lhs <= basic_string(rhs)
```

```
template<class charT, class traits, class Allocator>
bool
operator>=(const basic_string& lhs, const basic_string& rhs);
```

Returns a boolean value representing the lexicographical greater-than-or-equal relationship of `lhs` and `rhs`. Greater-than-or-equal is defined by the `compare()` member function.

```
template<class charT, class traits, class Allocator>
bool
operator>=(const charT* lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
bool
operator>=(const basic_string& lhs, const charT* rhs);
```

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

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

```
template <class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& a,
          basic_string<charT,traits,Allocator>& b);
```

Swaps the contents of `a` and `b` by calling `a`'s swap function on `b`.

```
template<class charT, class traits, class Allocator>
istream&
operator>>(istream& is, basic_string& str);
```

Reads `str` from `is` using `traits::char_in` until a `traits::is_del()` element is read. All elements read, except the delimiter, are placed in `str`. After the read, the function returns `is`.

```
template<class charT, class traits, class Allocator>
ostream&
operator<<(ostream& os, const basic_string& str);
```

Writes all elements of `str` to `os` in order from first to last, using `traits::char_out()`. After the write, the function returns `os`.

Non-member Function

```
template <class Stream, class charT, class traits,
          class Allocator>
Stream&
getline(Stream& is, basic_string& str, charT delim);
```

An unformatted input function that extracts characters from `is` into `str` until `npos - 1` characters are read, the end of the input sequence is reached, or the character read is `delim`. The characters are read using `traits::char_in()`.

Example

```
//
// string.cpp
//
 #include<string>
 #include <iostream.h>

 int main()
 {
   string test;

   //Type in a string over five characters long
   while(test.empty() ||  test.size() <= 5)
   {
     cout << "Type a string between 5 and 100 characters long. "
          << endl;
     cin >> test;
   }

   //Test operator[] access
   cout << "Changing the third character from " << test[2] <<
           " to * " << endl;
   test[2] = '*';
   cout << "now its: " << test << endl << endl;

   //Try the insertion member function
   cout << "Identifying the middle: ";
   test.insert(test.size() / 2, "(the middle is here!)");
   cout << test << endl << endl;

   //Try replacement
   cout << "I didn't like the word 'middle',so instead,I'll say:"
            << endl;
   test.replace(test.find("middle",0), 6, "center");
   cout << test << endl;

   return 0;
 }

Output :
Type a string between 5 and 100 characters long.
roguewave
Changing the third character from g to *
now its: ro*uewave
Identifying the middle: ro*u(the middle is here!)ewave
I didn't like the word 'middle', so instead, I'll say:
ro*u(the center is here!)ewave
```

See Also ***Allocators***, ***string***, ***wstring***

Summary

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

Description

For a complete discussion of iterators, see the *Iterators* **section of this reference.**

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Bidirectional iterators can move both forwards and backwards through a container, and have the ability to both read and write data. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator descriptions listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Bidirectional Iterators

A bidirectional iterator must meet all the requirements listed below. Note that most of these requirements are also the requirements for forward iterators.

`X u`	`u` might have a singular value
`X()`	`X()` might be singular
`X(a)`	copy constructor, `a == X(a)`.
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`

`a == b, a != b`	return value convertible to `bool`
`a->m`	equivalent to `(*a).m`
`*a`	return value convertible to `T&`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return `value` convertible to `const X&`
`*r--`	returns `T&`

Like forward iterators, bidirectional iterators have the condition that `a == b` implies `*a== *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

See Also ***Containers, Iterators, Forward Iterators***

binary_function

Function Object

Summary

Base class for creating binary function objects.

Synopsis

```
#include <functional>

template <class Arg1, class Arg2, class Result>
    struct binary_function{
         typedef Arg1 first_argument_type;
         typedef Arg2 second_argument_type;
         typedef Result result_type;
    };
```

Description

Function objects are objects with an `operator()` defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an `operator()` defined or a pointer to a function. The Standard C++ Library provides both a standard set of function objects, and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take two arguments are called *binary function objects*. Binary function objects are required to provide the typedefs `first_argument_type`, `second_argument_type`, and `result_type`. The ***binary_function*** class makes the task of creating templated binary function objects easier by providing the necessary typedefs for a binary function object. You can create your own binary function objects by inheriting from ***binary_function***.

See Also

function objects¸ ***unary_function***, the Function Objects section of the User's Guide.

Summary

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

Synopsis

```
#include <functional>

template<class Predicate>
class binary_negate ;
```

Description

binary_negate is a function object class that provides a return type for the function adaptor ***not2***. ***not2*** is a function adaptor, known as a negator, that takes a binary predicate function object as its argument and returns a binary predicate function object that is the complement of the original.

Note that ***not2*** works only with function objects that are defined as subclasses of the class ***binary_function***.

Interface

```
template<class Predicate>
class binary_negate
  : public binary_function<typename
                           predicate::first_argument_type,
                           typename
                           Predicate::second_argument_type,
                           bool>
{
public:

  typedef typename binary_function<typename
   Predicate::first_argument_type, typename
   Predicate::second_argument_type, bool>::second_argument_type
                                            second_argument_type;
  typedef typename binary_function<typename
   Predicate::first_argument_type, typename
   Predicate::second_argument_type, bool>::first_argument_type
                                           first_argument_type;
  typedef typename binary_function<typename
   Predicate::first_argument_type, typename
   Predicate::second_argument_type, bool>::result_type
                                           result_type;

  explicit binary_negate (const Predicate&);
  bool operator() (const first_argument_type&,
                   const second_argument_type&) const;
};

// Non-member Functions

template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Constructor

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

Construct a binary_negate object from predicate `pred`.

Operator

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

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

See Also ***binary_function***, ***not2***, ***unary_negate***

binary_search

Algorithm

Summary

Performs a binary search for a value on a container.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class T>
bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value);

template <class ForwardIterator, class T, class Compare>
bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value, Compare comp);
```

Description

The ***binary_search*** algorithm, like other related algorithms (***equal_range***, ***lower_bound*** and ***upper_bound***) performs a binary search on ordered containers. All binary search algorithms have two versions. The first version uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version allows you to include a function object of type `Compare`, which it assumes was the function used to sort the sequence. The function object must be a binary predicate.

The ***binary_search*** algorithm returns `true` if a sequence contains an element equivalent to the argument `value`. The first version of ***binary_search*** returns `true` if the sequence contains at least one element that is equal to the search value. The second version of the ***binary_search*** algorithm returns `true` if the sequence contains at least one element that satisfies the conditions of the comparison function. Formally, ***binary_search*** returns `true` if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions:

```
!(*i < value) && !(value < *i)
```

or

```
comp(*i, value) == false && comp(value, *i) == false
```

Complexity

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

Example

```
  //
  // b_search.cpp
  //
#include <vector>
#include <algorithm>
#include <iostream.h>

int main()
{
  typedef vector<int>::iterator iterator;
  int d1[10] = {0,1,2,2,3,4,2,2,6,7};
  //
  // Set up a vector
  //
  vector<int> v1(d1,d1 + 10);
  //
  // Try binary_search variants
  //
  sort(v1.begin(),v1.end());
  bool b1 = binary_search(v1.begin(),v1.end(),3);
  bool b2 =
    binary_search(v1.begin(),v1.end(),11,less<int>());
  //
  // Output results
  //
  cout << "In the vector: ";
  copy(v1.begin(),v1.end(),
          ostream_iterator<int,char>(cout," "));

  cout << endl << "The number 3 was "
       << (b1 ? "FOUND" : "NOT FOUND");
  cout << endl << "The number 11 was "
       << (b2 ? "FOUND" : "NOT FOUND") << endl;
  return 0;
}

Output :
In the vector: 0 1 2 2 2 2 3 4 6 7
The number 3 was FOUND
The number 11 was NOT FOUND
```

Warnings

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

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

instead of:

```
vector<int>
```

See Also

equal_range, ***lower_bound***, ***upper_bound***

bind1st, bind2nd, binder1st, binder2nd

Function Object

Summary

Templatized utilities to bind values to function objects

Synopsis

```
#include <functional>

template <class Operation>
class binder1st : public unary_function<typename
                   Operation::second_argument_type,
                   typename Operation::result_type> ;

template <class Operation, class T>
binder1st<Operation> bind1st (const Operation&, const T&);

template <class Operation>
class binder2nd : public unary_function<typename
                   Operation::first_argument_type,
                   typename Operation::result_type> ;

template <class Operation, class T>
binder2nd<Operation> bind2nd (const Operation&, const T&);
```

Description

Because so many functions provided by the standard library take other functions as arguments, the library includes classes that let you build new function objects out of old ones. Both `bind1st()` and `bind2nd()` are functions that take as arguments a binary function object `f` and a value `x,` and return, respectively, classes ***binder1st*** and ***binder2nd***. The underlying function object must be a subclass of ***binary_function***.

Class ***binder1st*** binds the value to the first argument of the binary function, and ***binder2nd*** does the same thing for the second argument of the function. The resulting classes can be used in place of a unary predicate in other function calls.

For example, you could use the ***count_if*** algorithm to count all elements in a vector that are less than or equal to 7, using the following:

```
count_if (v.begin, v.end, bind1st(greater<int> (),7), littleNums)
```

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

Interface

```
// Class binder1st
 template <class Operation>
 class binder1st
   : public unary_function<typename
                           Operation::second_argument_type,
                           typename Operation::result_type>
```

```
{
public:

   typedef typename unary_function<typename
    Operation::second_argument_type, typename
    Operation::result_type>::argument_type argument_type;
   typedef typename unary_function<typename
    Operation::second_argument_type, typename
    Operation::result_type>::result_type result_type;

   binder1st(const Operation&,
             const typename Operation::first_argument_type&);
   result_type operator() (const argument_type&) const;
};

// Class binder2nd
 template <class Operation>
 class binder2nd
   : public unary_function<typename
                           Operation::first_argument_type,
                           typename Operation::result_type>
{
public:
   typedef typename unary_function<typename
    Operation::first_argument_type, typename
    Operation::result_type>::argument_type argument_type;
   typedef typename unary_function<typename
    Operation::first_argument_type, typename
    Operation::result_type>::result_type result_type;

   binder2nd(const Operation&,
             const typename Operation::second_argument_type&);
   result_type operator() (const argument_type&) const;
};

// Creator bind1st

   template <class Operation, class T>
   binder1st<Operation> bind1st (const Operation&, const T&);

// Creator bind2nd

   template<class Operation, class T>
   binder2nd <Operation> bind2nd(const Operation&, const T&);
```

Example

```
//
// binders.cpp
//
 #include <functional>
 #include <algorithm>
 #include <vector>
 #include <iostream.h>
 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[4] = {1,2,3,4};
   //
   // Set up a vector
```

```
    //
    vector<int> v1(d1,d1 + 4);
    //
    // Create an 'equal to 3' unary predicate by binding 3 to
    // the equal_to binary predicate.
    //
    binder1st<equal_to<int> > equal_to_3 =
       bind1st(equal_to<int>(),3);
    //
    // Now use this new predicate in a call to find_if
    //
    iterator it1 = find_if(v1.begin(),v1.end(),equal_to_3);
    //
    // Even better, construct the new predicate on the fly
    //
    iterator it2 =
       find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
    //
    // And now the same thing using bind2nd
    // Same result since == is commutative
    //
    iterator it3 =
       find_if(v1.begin(),v1.end(),bind2nd(equal_to<int>(),3));
    //
    // it3 = v1.begin() + 2
    //
    // Output results
    //
    cout << *it1 << " " << *it2 << " " << *it3 << endl;
    return 0;
  }
```

```
Output : 3 3 3
```

Warnings

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

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

instead of:

```
vector<int>
```

See Also

Function Object

Container

Summary

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

Synopsis

```
#include <bitset>

template <size_t N>
class bitset ;
```

Description

bitset<size_t N> is a class that describes objects that can store a sequence consisting of a fixed number of bits, `N.` Each bit represents either the value zero (`reset`) or one (`set`) and has a non-negative position `pos`.

Errors and exceptions

Bitset constructors and member functions may report the following three types of errors — each associated with a distinct exception:

- invalid-argument error or `invalid_argument()` exception;
- out-of-range error or `out_of_range()` exception;
- overflow error or `over-flow_error()` exception;

If exceptions are not supported on your compiler, you will get an assertion failure instead of an exception.

Interface

```
template <size_t N>
class bitset {

public:

// bit reference:

  class reference {
   friend class bitset<N>;
  public:

   ~reference();
   reference& operator= (bool);
   reference& operator= (const reference&);
   bool operator~() const;
   operator bool() const;
   reference& flip();
  };
```

```
// Constructors
  bitset ();
  bitset (unsigned long);
  explicit bitset (const string&, size_t = 0,
                   size_t = (size_t)-1);
  bitset (const bitset<N>&);
  bitset<N>& operator= (const bitset<N>&);

// Bitwise Operators and Bitwise Operator Assignment
   bitset<N>& operator&= (const bitset<N>&);
   bitset<N>& operator|= (const bitset<N>&);
   bitset<N>& operator^= (const bitset<N>&);
   bitset<N>& operator<<= (size_t);
   bitset<N>& operator>>= (size_t);

// Set, Reset, Flip
   bitset<N>& set ();
   bitset<N>& set (size_t, int = 1);
   bitset<N>& reset ();
   bitset<N>& reset (size_t);
   bitset<N> operator~() const;
   bitset<N>& flip ();
   bitset<N>& flip (size_t);

// element access
   reference operator[] (size_t);
   unsigned long to_ulong() const;
   string to_string() const;
   size_t count() const;
   size_t size() const;
   bool operator== (const bitset<N>&) const;
   bool operator!= (const bitset<N>&) const;
   bool test (size_t) const;
   bool any() const;
   bool none() const;
   bitset<N> operator<< (size_t) const;
   bitset<N> operator>> (size_t) const;

};

// Non-member operators

template <size_t N>
bitset<N> operator& (const bitset<N>&, const bitset<N>&);

template <size_t N>
bitset<N> operator| (const bitset<N>&, const bitset<N>&);

template <size_t N>
bitset<N> operator^ (const bitset<N>&, const bitset<N>&);

template <size_t N>
istream& operator>> (istream&, bitset<N>&);

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

Constructors

```
bitset();
```
Constructs an object of class `bitset<N>`, initializing all bit values to zero.

```
bitset(unsigned long val);
```
Constructs an object of class `bitset<N>`, initializing the first `M` bit values to the corresponding bits in `val`. `M` is the smaller of `N` and the value `CHAR_BIT * sizeof(unsigned long)`. If `M < N`, remaining bit positions are initialized to zero. Note: `CHAR_BIT` is defined in `<climits>`.

```
explicit
bitset(const string& str, size_t pos = 0,
       size_t n = (size_t)-1);
```
Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`. The function throws an `invalid_argument` exception if any of the `rlen` characters in `str`, beginning at position `pos`,is other than 0 or 1. Otherwise, the function constructs an object of class ***bitset<N>***, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`. This constructor requires that `pos <= str.size()`, otherwise it throws an `out_of_range` exception.

```
bitset(const bitset<N>& rhs);
```
Copy constructor. Creates a copy of `rhs`.

Assignment Operator

```
bitset<N>&
operator=(const bitset<N>& rhs);
```
Erases all bits in self, then inserts into self a copy of each bit in `rhs`. Returns a reference to `*this`.

Operators

```
bool
operator==(const bitset<N>& rhs) const;
```
Returns `true` if the value of each bit in `*this` equals the value of each corresponding bit in `rhs`. Otherwise returns `false`.

```
bool
operator!=(const bitset<N>& rhs) const;
```
Returns `true` if the value of any bit in `*this` is not equal to the value of the corresponding bit in `rhs`. Otherwise returns `false`.

```
bitset<N>&
operator&=(const bitset<N>& rhs);
```
Clears each bit in `*this` for which the corresponding bit in `rhs` is clear and leaves all other bits unchanged. Returns `*this`.

```
bitset<N>&
operator|=(const bitset<N>& rhs);
```
Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged. Returns `*this`.

```
bitset<N>&
operator^=(const bitset<N>& rhs);
```

Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged. Returns `*this`.

```
bitset<N>&
operator<<=(size_t pos);
```

Replaces each bit at position `I` with 0 if `I < pos` or with the value of the bit at `I - pos` if `I >= pos`. Returns `*this`.

```
bitset<N>&
operator>>=(size_t pos);
```

Replaces each bit at position `I` with 0 if `pos >= N-I` or with the value of the bit at position `I + pos` if `pos < N-I`. Returns `*this`.

```
bitset<N>&
operator>>(size_t pos) const;
```

Returns `bitset<N>(*this) >>= pos`.

```
bitset<N>&
operator<<(size_t pos) const;
```

Returns `bitset<N>(*this) <<= pos`.

```
bitset<N>
operator~() const;
```

Returns the bitset that is the logical complement of each bit in `*this`.

```
bitset<N>
operator&(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `AND` of `lhs` with `rhs`.

```
bitset<N>
operator|(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `OR` of `lhs` with `rhs`.

```
bitset<N>
operator^(const bitset<N>& lhs,
          const bitset<N>& rhs);
```

`lhs` gets logical `XOR` of `lhs` with `rhs`.

```
template <size_t N>
istream&
operator>>(istream& is, bitset<N>& x);
```

Extracts up to `N` characters (single-byte) from `is`. Stores these characters in a temporary object `str` of type `string`, then evaluates the expression `x = bitset<N>(str)`. Characters are extracted and stored until any of the following occurs:

- `N` characters have been extracted and stored

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

Returns `is`.

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

Returns `os << x.to_string()`

Member Functions

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

Returns `true` if any bit in `*this` is set. Otherwise returns `false`.

```
size_t
count() const;
```

Returns a count of the number of bits set in `*this`.

```
bitset<N>&
flip();
```

Flips all bits in `*this`, and returns `*this`.

```
bitset<N>&
flip(size_t pos);
```

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

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

Returns `true` if no bit in `*this` is set. Otherwise returns `false`.

```
bitset<N>&
reset();
```

Resets all bits in `*this`, and returns `*this`.

```
bitset<N>&
reset(size_t pos);
```

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

```
bitset<N>&
set();
```

Sets all bits in `*this`, and returns `*this`.

```
bitset<N>&
set(size_t pos, int val = 1);
```

Stores a new value in the bits at position `pos` in `*this.` If `val` is nonzero, the stored value is one, otherwise it is zero. Throws an `out_of_range` exception if `pos` does not correspond to a valid bit position.

```
size_t
size() const;
```

Returns the template parameter `N`.

```
bool
test(size_t pos) const;
```

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

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

Returns an object of type `string`, `N` characters long.

Each position in the new string is initialized with a character ('0' for zero and '1' for one) representing the value stored in the corresponding bit position of `*this`. Character position `N - 1` corresponds to bit position 0. Subsequent decreasing character positions correspond to increasing bit positions.

```
unsigned long
to_ulong() const;
```

Returns the integral value corresponding to the bits in `*this`. Throws an `overflow_error` if these bits cannot be represented as type `unsigned long`.

See Also ***Containers***

Summary

A traits class providing types and operations to the ***basic_string*** container and ***iostream*** classes.

Synopsis

```
#include <string>
template<class charT>
struct char_traits
```

Description

The template structure ***char_traits<charT>*** defines the types and functions necessary to implement the ***iostreams*** and ***string*** template classes. It is templatized on `charT`, which represents the character container type. Each specialized version of ***char_traits<charT>*** provides the default definitions corresponding to the specialized character container type.

Users have to provide specialization for ***char_traits*** if they use other character types than `char` and `wchar_t`.

Interface

```
template<class charT>
struct char_traits {

  typedef charT                     char_type;
  typedef INT_T                     int_type;
  typedef POS_T                     pos_type;
  typedef OFF_T                     off_type;
  typedef STATE_T                   state_type;

  static char_type        to_char_type(const int_type&);
  static int_type         to_int_type(const char_type&);
  static bool             eq(const char_type&,const char_type& );
  static bool             eq_int_type(const int_type&,const int_type&);

  static int_type         eof();
  static int_type         not_eof(const int_type&);

  static void             assign(char_type&,const char_type&);
  static bool             lt(const char_type&,const char_type&);
  static int              compare(const char_type*,const char_type*,size_t);
  static size_t           length(const char_type*);
  static const char_type* find(const char_type*,int n,const char_type&);

  static char_type*       move(char_type*,const char_type*,size_t);
  static char_type*       copy(char_type*,const char_type*, size_t);
  static char_type*       assign(char_type*,size_t,const char_type&);

};
```

Types

char_type

The type char_type represents the character container type. It must be convertible to int_type.

int_type

The type int_type is another character container type which can also hold an end-of-file value. It is used as the return type of some of the iostream class member functions. If char_type is either char or wchar_t, int_type is int or wint_t, respectively.

off_type

The type off_type represents offsets to positional information. It is used to represent:

- a signed displacement, measured in characters, from a specified position within a sequence.
- an absolute position within a sequence.

The value off_type(-1) can be used as an error indicator. Value of type off_type can be converted to type pos_type, but no validity of the resulting pos_type value is ensured.

If char_type is either char or wchar_t, off_type is streamoff or wstreamoff, respectively.

pos_type

The type pos_type describes an object that can store all the information necessary to restore an arbitrary sequence to a previous stream position and conversion state. The conversion pos_type(off_type(-1)) constructs the invalid pos_type value to signal error.

If char_type is either char or wchar_t, pos_type is streampos or wstreampos, respectively.

state_type

The type state_type holds the conversion state, and is compatible with the function locale::codecvt().

If char_type is either char or wchar_t, state_type is mbstate_t.

Types Default-Values

specialization type	on char	on wchar_t
char_type	char	wchar_t
int_type	int	wint_t
off_type	streamoff	wstreamoff
pos_type	streampos	wstreampos
state_type	mbstate_t	mbstate_t

Value Functions

```
void
assign(char_type& c1, const char_type& c2);
```

Assigns one character value to another. The value of `c2` is assigned to `c1`.

```
char_type*
assign(char_type* s,size_t n,const char_type& a);
```

Assigns one character value to `n` elements of a character array. The value of `a` is assigned to `n` elements of `s`.

```
char_type*
copy(char_type* s1, const char_type* s2, size_t n);
```

Copies `n` characters from the object pointed at by `s1` into the object pointed at by `s2`. The ranges of `(s1,s1+n)` and `(s2,s2+n)` may not overlap.

```
int_type
eof();
```

Returns an `int_type` value which represents the end-of-file. It is returned by several functions to indicate end-of-file state, or to indicate an invalid return value.

```
const char_type*
find(const char_type* s, int n, const char_type& a);
```

Looks for the value of `a` in `s`. Only `n` elements of `s` are examined. Returns a pointer to the matched element if one is found. Otherwise returns a pointer to the `n` element in `s`.

```
size_t
length(const char_type* s);
```

Returns the length of a null terminated character string pointed at by `s`.

```
char_type*
move(char_type* s1, const char_type* s2, size_t n);
```

Moves `n` characters from the object pointed at by `s1` into the object pointed at by `s2`. The ranges of `(s1,s1+n)` and `(s2,s2+n)` may overlap.

```
int_type
not_eof(const int_type& c);
```

Returns a value which is not equal to the end-of-file value.

Test Functions

```
int
compare(const char_type* s1,const char_type* s2,size_t n);
```

Compares n values from s1 with n values from s2. Returns 1 if s1 is greater than s2, -1 if s1 is less than s2, or 0 if they are equal.

```
bool
eq(const char_type& c1, const char_type& c2);
```

Returns true if `c1` and `c2` represent the same character.

```
bool
eq_int_type(const int_type& c1, const int_type& c2);
```

Returns true if `c1` and `c2` represents the same character.

```
bool
lt(const char_type& c1,const char_type& c2);
```

Returns true if c1 is less than c2.

Conversion Functions

```
char_type
to_char_type(const int_type& c);
```

Converts a valid character represented by a value of type `int_type` to the corresponding `char_type` value.

```
int_type
to_int_type(const char_type& c);
```

Converts a valid character represented by a value of type `char_type` to the corresponding `int_type` value.

See Also

iosfwd(3C++), ***fpos***(3C++)

Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++, Section 21.1.4, 21.1.5, 27.1.2.

Standards Conformance

ANSI X3J16/ISO WG21 Joint C++ Committee

compare

Summary A binary function or a function object that returns true or false. ***compare*** objects are typically passed as template parameters, and used for ordering elements within a container.

See Also ***binary_function***, ***function object***

Summary

C++ complex number library

Specializations

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

Synopsis

```
#include <complex>

template <class T>
class complex ;

class complex<float>;
class complex<double>;
class complex<long double>;
```

Description

complex<T> is a class that supports complex numbers. A complex number has a real part and an imaginary part. The ***complex*** class supports equality, comparison and basic arithmetic operations. In addition, mathematical functions such as exponents, logarithms, powers, and square roots are also available.

Interface

```
template <class T>
class complex {

public:
   typedef T value_type;

   complex (T = 0 , T = 0);
   template <class X> complex
    (const complex<X>&);

   T real () const;
   T imag () const;

   complex<T>& operator= (const T&);
   complex<T>& operator+=(const T&);
   complex<T>& operator-=(const T&);
   complex<T>& operator*=(const T&);
   complex<T>& operator/=(const T&);

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

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

```
   template <class X>
    complex<T>& operator*= (const complex<X>&);
   template <class X>
    complex<T>& operator/= (const complex<X>&);
};


// Non-member Operators

template<class T>
 complex<T> operator+ (const complex<T>&, const complex<T>&);
template<class T>
 complex<T> operator+ (const complex<T>&, T);
template<class T>
 complex<T> operator+ (T, const complex<T>&);

template<class T>
 complex<T> operator- (const complex<T>&, const complex<T>&);
template<class T>
 complex<T> operator- (const complex<T>&, T);
template<classT>
 complex<T> operator- (T, const complex<T>&);

template<class T>
 complex<T> operator* (const complex<T>&, const complex<T>&);
template<class T>
 complex<T> operator* (const complex<T>&, T);
template<class T>
 complex<T> operator* (T, const complex<T>&);

template<class T>
 complex<T> operator/ (const complex<T>&, const complex<T>&);
template<class T>
 complex<T> operator/ (const complex<T>&, T);
template<class T>
 complex<T> operator/ (T, const complex<T>&);

template<class T>
 complex<T> operator+ (const complex<T>&);
template<class T>
 complex<T> operator- (const complex<T>&);

template<class T>
 bool operator== (const complex<T>&, const complex<T>&);
template<class T>
 bool operator== (const complex<T>&, T);
template<class T>
 bool operator== (T, const complex<T>&);

template<class T>
 bool operator!= (const complex<T>&, const complex<T>&);
template<class T>
 bool operator!= (const complex<T>&, T);
template<class T>
 bool operator!= (T, const complex<T>&);

template <class X>
```

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

// Values

template<class T> T real (const complex<T>&);
template<class T> T imag (const complex<T>&);

template<class T> T abs (const complex<T>&);
template<class T> T arg (const complex<T>&);
template<class T> T norm (const complex<T>&);

template<class T> complex<T> conj (const complex<T>&);
template<class T> complex<T> polar (T, T);

// Transcendentals

template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp (const complex<T>&);
template<class T> complex<T> log (const complex<T>&);

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

template<class T> complex<T> pow (const complex<T>&, int);
template<class T> complex<T> pow (const complex<T>&, T);
template<class T> complex<T> pow (const complex<T>&,
                                  const complex<T>&);
template<class T> complex<T> pow (T, const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);
template<class T> complex<T> sinh (const complex<T>&);
template<class T> complex<T> sqrt (const complex<T>&);
template<class T> complex<T> tan (const complex<T>&);
template<class T> complex<T> tanh (const complex<T>&);
```

Constructors

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

Constructs an object of class ***complex***, initializing `re_arg` to the real part and `im_arg` to the imaginary part.

template <class X> complex
`(const complex<X>&);`

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

Assignment Operators

`complex<T>&` **operator=**`(const T& v);`

Assigns `v` to the real part of itself, setting the imaginary part to 0.

`complex<T>&` **operator+=**`(const T& v);`

Adds `v` to the real part of itself, then returns the result.

`complex<T>&` **operator-=**`(const T& v);`

Subtracts `v` from the real part of itself, then returns the result.

```
complex<T>& operator*=(const T& v);
```
Multiplies `v` by the real part of itself, then returns the result.

```
complex<T>& operator/=(const T& v);
```
Divides `v` by the real part of itself, then returns the result.

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

```
template <class X>
complex<T>
operator+=(const complex<X>& c);
```
Adds `c` to itself, then returns the result.

```
template <class X>
complex<T>
operator-=(const complex<X>& c);
```
Subtracts `c` from itself, then returns the result.

```
template <class X>
complex<T>
operator*=(const complex<X>& c);
```
Multiplies itself by `c` then returns the result.

```
template <class X>
complex<T>
operator/=(const complex<X>& c);
```
Divides itself by `c`, then returns the result.

Member Functions

```
T
imag() const;
```
Returns the imaginary part of the complex number.

```
T
real() const;
```
Returns the real part of the complex number.

Non-member Operators

```
template<class T> complex<T>
operator+(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator+(const complex<T>& lhs, T rhs);
template<class T> complex<T>
operator+(T lhs, const complex<T>& rhs);
```
Returns the sum of `lhs` and `rhs`.

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

Returns the difference of `lhs` and `rhs`.

```
template<class T> complex<T>
operator*(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator*(const complex<T>& lhs, T rhs);
template<class T> complex<T>
operator* (T lhs, const complex<T>& rhs);
```

Returns the product of `lhs` and `rhs`.

```
template<class T> complex<T>
operator/(const complex<T>& lhs,const complex<T>& rhs);
template<class T> complex<T>
operator/(const complex<T>& lhs, T rhs);
template<class T> complex<T>
operator/(T lhs, const complex<T>& rhs);
```

Returns the quotient of `lhs` divided by `rhs`.

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

Returns `rhs`.

```
template<class T> complex<T>
operator-(const complex<T>& lhs);
```

Returns `complex<T>(-lhs.real(), -lhs.imag())`.

```
template<class T> bool
operator==(const complex<T>& x, const complex<T>& y);
```

Returns `true` if the real and imaginary parts of `x` and `y` are equal.

```
template<class T> bool
operator==(const complex<T>& x, T y);
```

Returns `true` if `y` is equal to the real part of `x` and the imaginary part of `x` is equal to 0.

```
template<class T> bool
operator==(T x, const complex<T>& y);
```

Returns `true` if `x` is equal to the real part of `y` and the imaginary part of `y` is equal to 0.

```
template<class T> bool
operator!=(const complex<T>& x, const complex<T>& y);
```

Returns `true` if either the real or the imaginary part of `x` and `y` are not equal.

```
template<class T> bool
operator!=(const complex<T>& x, T y);
```

Returns `true` if `y` is not equal to the real part of `x` or the imaginary part of `x` is not equal to 0.

```
template<class T> bool
operator!=(T x, const complex<T>& y);
```

Returns `true` if `x` is not equal to the real part of `y` or the imaginary part of `y` is not equal to 0.

```
template <class X> istream&
operator>>(istream& is, complex<X>& x);
```

Reads a complex number `x` into the input stream `is`. `x` may be of the form `u`, `(u)`, or `(u,v)` where `u` is the real part and `v` is the imaginary part. If bad input is encountered, the `ios::badbit flag` is set.

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

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

Non-member Functions

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

Returns the absolute value or magnitude of `c` (the square root of the norm).

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

Returns the conjugate of `c`.

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

Returns the cosine of `c`.

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

Returns the hyperbolic cosine of `c`.

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

Returns `e` raised to the `x` power.

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

Returns the imaginary part of `c`.

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

Returns the natural logarithm of `x`. This function returns the complex value whose phase angle is greater than -pi and less than pi.

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

Returns the logarithm base 10 of `x`.

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

Returns the squared magnitude of `c`. (The sum of the squares of the real and imaginary parts.)

```
template<class T> complex<T>
polar(const T& m, const T& a);
```

Returns the complex value of a complex number whose magnitude is `m` and phase angle is `a`, measured in radians.

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

Returns `x` raised to the `y` power.

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

Returns the real part of `c`.

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

Returns the sine of `c`.

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

Returns the hyperbolic sine of `c`.

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

Returns the square root of `x`. This function returns the complex value whose phase angle is greater than -pi/2 and less than or equal to

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

Returns the tangent of `x`.

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

Returns the hyperbolic tangent of `x`.

Example

```
//
// complex.cpp
//
 #include <complex>
```

```
#include <iostream.h>

int main()
{
  complex<double> a(1.2, 3.4);
  complex<double> b(-9.8, -7.6);

  a += b;
  a /= sin(b) * cos(a);
  b *= log(a) + pow(b, a);

  cout << "a = " << a << ", b = " << b << endl;

  return 0;
}

Output :
a = (1.42804e-06,-0.0002873), b = (58.2199,69.7354)
```

On compilers that don't support member function templates, the arithmetic operators will not work on any arbitrary type. (They will work only on float, double and long doubles.) You also will only be able to perform binary arithmetic on types that are the same.

Warnings

Compilers that don't support non-converting constructors will permit unsafe downcasts (i.e., long double to double, double to float, long double to float).

Summary

A standard template library (STL) collection.

Description

Within the standard template library, collection classes are often described as containers. A container stores a collection of other objects and provides certain basic functionality that supports the use of generic algorithms. Containers come in two basic flavors: sequences, and associative containers. They are further distinguished by the type of iterator they support.

A *sequence* supports a linear arrangement of single elements. ***vector***, ***list***, ***deque***, ***bitset***, and ***string*** fall into this category. *Associative containers* map values onto keys, which provides efficient retrieval of the values based on the keys. The STL provides the ***map***, ***multimap***, ***set*** and ***multiset*** associative containers. ***map*** and ***multimap*** store the value and the key separately and allow for fast retrieval of the value, based upon fast retrieval of the key. ***set*** and ***multiset*** store only keys allowing fast retrieval of the key itself.

Container Requirements

Containers within the STL must meet the following requirements. Sequences and associative containers must also meet their own separate sets of requirements. The requirements for containers are:

- A container allocates all storage for the objects it holds.
- A container `X` of objects of type `T` provides the following types:

`X::value_type`	a `T`
`X::reference`	`lvalue` of `T`
`X::const_reference`	`const lvalue` of `T`
`X::iterator`	an iterator type pointing to `T`. `X::iterator` cannot be an output iterator.
`X::const_iterator`	an iterator type pointing to `const T`. `x::iterator` cannot be an output iterator.
`X::difference_type`	a signed integral type (must be the same as the distance type for `X::iterator` and `X::const_iterator`
`X::size_type`	an unsigned integral type representing any non-negative value of `difference_type`
`X::allocatr_type`	type of allocator used to obtain storage for elements stored in the container

- A container provides a default constructor, a copy constructor, an assignment operator, and a full complement of comparison operators (==, !=, <, >, <=, >=).
- A container provides the following member functions:

`begin()`	Returns an `iterator` or a `const_iterator` pointing to the first element in the collection.
`end()`	Returns an `iterator` or a `const_iterator` pointing just beyond the last element in the collection.
`swap(container)`	Swaps elements between this container and the swap's argument.
`clear()`	Deletes all the elements in the container.
`size()`	Returns the number of elements in the collection as a `size_type`.
`max_size()`	Returns the largest possible number of elements for this type of container as a `size_type`.
`empty()`	Returns `true` if the container is empty, `false` otherwise.
`get_allocator()`	Returns the allocator used by this container

Reversible Containers

A container may be reversible. Essentially, a reversible container provides a reverse iterator that allows traversal of the collection in a direction opposite that of the default iterator. A reversible container must meet the following requirements in addition to those listed above:

- A reversible container provides the following types:

`X::reverse_iterator`	An iterator type pointing to `T`.
`X::const_reverse_iterator`	An iterator type pointing to `const T`

- A reversible container provides the following member functions:

`rbegin()`	Returns a `reverse_iterator` or a `const_reverse_iterator` pointing past the end of the collection
`rend()`	Returns a `reverse_iterator` or a `const_reverse_iterator` pointing to the first

element in the collection.

Sequences

In addition to the requirements for containers, the following requirements hold for sequences:

- `iterator` and `const_iterator` must be forward iterators, bidirectional iterators or random access iterators.
- A sequence provides the following constructors:

`X(n, t)`	Constructs a container with `n` copies of `t`.
`X(i, j)`	Constructs a container with elements from the range `[i,j)`.

- A sequence provides the following member functions:

`insert(p,t)`	Inserts the element `t` in front of the position identified by the iterator `p`.
`insert(p,n,t)`	Inserts `n` copies of `t` in front of the position identified by the iterator `p`.
`insert(p,i,j)`	Inserts elements from the range `[i,j)` in front of the position identified by the iterator `p`.
`erase(q)`	Erases the element pointed to by the iterator `q`.
`erase(q1,q2)`	Erases the elements in the range `[q1,q2)`.

- A sequence may also provide the following member functions if they can be implemented with constant time complexity.

`front()`	Returns the element pointed to by `begin()`
`back()`	Returns the element pointed to by `end()`
`push_front(x)`	Inserts the element `x` at `begin()`
`push_back(x)`	Inserts the element `x` at `end()`
`pop_front()`	Erases the element at `begin()`
`pop_back()`	Erases the element at `end() -1`
`operator[](n)`	Returns the element at `a.begin() + n`

Associative Containers

In addition to the requirements for a container, the following requirements hold for associative containers:

- For an associative container `iterator` and `const_iterator` must be bidirectional iterators. Associative containers are inherently sorted. Their iterators proceed through the container in the non-descending order of keys (where non-descending order is defined by the comparison object that was used to construct the container).

- An associative container provides the following types:

`X::key_type`	the type of the `Key`
`X::key_compare`	the type of the comparison to use to put the keys in order
`X::value_compare`	the type of the comparison used on values

- The default constructor and copy constructor for associative containers use the template parameter comparison class.

- An associative container provides the following additional constructors:

`X(c)`	Construct an empty container using `c` as the comparison object
`X(i,j,c)`	Constructs a container with elements from the range `[i,j)` and the comparison object `c`.
`X(i, j)`	Constructs a container with elements from the range `[i,j)` using the template parameter comparison object.

- An associative container provides the following member functions:

`key_comp()`	Returns the comparison object used in constructing the associative container.
`value_comp()`	Returns the value comparison object used in constructing the associative container.
`insert(t)`	Inserts `t` if and only if there is no element in the container with key equal to the key of `t`. Returns a `pair<iterator,bool>`. The `bool` component of the returned pair indicates the success or failure of the operation and the `iterator` component points to the element with key equal to key of `t`.
`insert(p,t)`	If the container does *not* support redundant key values then this function only inserts `t` if there is no key present that is equal to the key of `t`. If the container *does* support redundant keys then this function always inserts the element `t`. The iterator `p` serves as a hint of where to start searching, allowing for some optimization of the insertion. It does not

	restrict the algorithm from inserting ahead of that location if necessary.
`insert(i,j)`	Inserts elements from the range `[i,j)`.
`erase(k)`	Erases all elements with key equal to `k`. Returns number of erased elements.
`erase(q)`	Erases the element pointed to by `q`.
`erase(q1,q2)`	Erases the elements in the range `[q1,q2)`.
`find(k)`	Returns an iterator pointing to an element with key equal to `k` or `end()` if such an element is not found.
`count(k)`	Returns the number of elements with key equal to `k`.
`lower_bound(k)`	Returns an iterator pointing to the first element with a key greater than or equal to `k`.
`upper_bound(k)`	Returns an iterator pointing to the first element with a key less than or equal to `k`.
`equal_range(k)`	Returns a pair of iterators such that the first element of the pair is equivalent to `lower_bound(k)` and the second element equivalent to `upper_bound(k)`.

See Also

bitset, deque, list, map, multimap, multiset, priority_queue, queue, set, stack, vector

copy, copy_backward

Algorithm

Summary Copies a range of elements

Synopsis

```
#include <algorithm>

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

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

Description

The ***copy*** algorithm copies values from the range specified by `[first , last)` to the range that specified by `[result, result + (last - first))`. ***copy*** can be used to copy values from one container to another, or to copy values from one location in a container to another location in the *same* container, as long as `result` is not within the range `[first-last)`. ***copy*** returns `result + (last - first)`. For each non-negative integer `n < (last - first)`, ***copy*** assigns `*(first + n)` to `*(result + n)`. The result of ***copy*** is undefined if `result` is in the range `[first, last)`.

Unless `result` is an insert iterator, ***copy*** assumes that at least as many elements follow `result` as are in the range `[first, last)`.

The ***copy_backward*** algorithm copies elements in the range specified by `[first, last)` into the range specified by `[result - (last - first), result)`, starting from the end of the sequence (`last-1`) and progressing to the front (`first`). Note that ***copy_backward*** does *not* reverse the order of the elements, it simply reverses the order of transfer. ***copy_backward*** returns `result - (last - first)`. You should use ***copy_backward*** instead of ***copy*** when `last` is in the range `[result - (last - first), result)`. For each positive integer `n <= (last - first)`, ***copy_backward*** assigns `*(last - n)` to `*(result - n)`. The result of ***copy_backward*** is undefined if `result` is in the range `[first, last)`.

Unless `result` is an insert iterator, ***copy_backward*** assumes that there are at least as many elements ahead of `result` as are in the range `[first, last)`.

Complexity

Both ***copy*** and ***copy_backward*** perform exactly `last - first` assignments.

Example

```
  //
  // stdlib/examples/manual.copyex.cpp
  //
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
  int d1[4] = {1,2,3,4};
  int d2[4] = {5,6,7,8};

  // Set up three vectors
  //
  vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4), v3(d2,d2 + 4);
  //
  // Set up one empty vector
  //
  vector<int> v4;
  //
  // Copy v1 to v2
  //
  copy(v1.begin(),v1.end(),v2.begin());
  //
  // Copy backwards v1 to v3
  //
  copy_backward(v1.begin(),v1.end(),v3.end());
  //
  // Use insert iterator to copy into empty vector
  //
  copy(v1.begin(),v1.end(),back_inserter(v4));
  //
  // Copy all four to cout
  //
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;
  copy(v3.begin(),v3.end(),out);
  cout << endl;
  copy(v4.begin(),v4.end(),out);
  cout << endl;

  return 0;
}

Output :
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

Warning

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

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

instead of:

```
vector <int>
```

Summary

Count the number of elements in a container that satisfy a given condition.

Synopsis

```
#include <algorithm>
template<class InputIterator, class T>
 iterator_trait<InputIterator>::distance_type
 count(InputIterator first, InputIterator last,
       const T& value);

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

template<class InputIterator, class Predicate>
 iterator_trait<InputIterator>::distance_type
 count_if(InputIterator first, InputIterator last,
          Predicate pred);

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

Description

The ***count*** algorithm compares `value` to elements in the sequence defined by iterators `first` and `last`. The first version of ***count*** return the number of matches. The second version increments a counting value `n` each time it finds a match. i.e., ***count*** returns (or adds to `n`) the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
*i == value
```

The ***count_if*** algorithm lets you specify a predicate, and returns the number of times an element in the sequence satisfies the predicate (or increments `n` that number of times). That is, ***count_if*** returns (or adds to `n`) the number of iterators `i` in the range `[first, last)` for which the following condition holds:

```
pred(*i) == true.
```

Complexity

Both ***count*** and ***count_if*** perform exactly `last-first` applications of the corresponding predicate.

Example

```
//
```

```
// count.cpp
//
// Does not demonstrate the partial specialization versions
// of count and count_if
//
 #include <vector>
 #include <algorithm>
 #include <iostream.h>

 int main()
 {
   int sequence[10] = {1,2,3,4,5,5,7,8,9,10};
   int i=0,j=0,k=0;
   //
   // Set up a vector
   //
   vector<int> v(sequence,sequence + 10);

   count(v.begin(),v.end(),5,i);  // Count fives
   count(v.begin(),v.end(),6,j);  // Count sixes
   //
   // Count all less than 8
   // I=2, j=0
   //
   count_if(v.begin(),v.end(),bind2nd(less<int>(),8),k);
   // k = 7

   cout << i << " " << j << " " << k << endl;
   return 0;
 }

Output :  2 0 7
```

If your compiler does not support partial specialization then the first version of both ***count*** and ***count_if*** (the one that returns the count) will not be available.

Warnings

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

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

instead of:

```
vector <int>
```

deque

Container

Summary

A sequence that supports random access iterators and efficient insertion / deletion at both beginning and end.

Synopsis

```
#include <deque>

template <class T, class Allocator = allocator<T> >
 class deque;
```

Description

deque<T, Allocator> is a type of sequence that supports random access iterators. It supports constant time insert and erase operations at the beginning or the end of the container. Insertion and erase in the middle take linear time. Storage management is handled by the `Allocator` template parameter.

Any type used for the template parameter `T` must provide the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

Default constructor	`T()`
Copy constructors	`T(t)` and `T(u)`
Destructor	`t.~T()`
Address of	`&t` and `&u` yielding `T*` and `const T*` respectively
Assignment	`t = a` where `a` is a (possibly `const`) value of `T`

Interface

```
template <class T, class Allocator = allocator<T> >
 class deque {

public:

 // Types

   class iterator;
   class const_iterator;
   typedef T value_type;
   typedef Allocator allocator_type;
   typename  reference;
   typename  const_reference;
   typename  size_type;
   typename  difference_type;
```

```
   typename reverse_iterator;
   typename  const_reverse_iterator;


 // Construct/Copy/Destroy

   explicit deque (const Allocator& = Allocator());
   explicit deque (size_type, const Allocator& = Allocator ());
   deque (size_type, const T& value,
          const Allocator& = Allocator ());
   deque (const deque<T,Allocator>&);
   template <class InputIterator>
    deque (InputIterator, InputIterator,
           const Allocator& = Allocator ());
   ~deque ();
   deque<T,Allocator>& operator= (const deque<T,Allocator>&);
   template <class InputIterator>
    void assign (InputIterator, InputIterator);
   template <class Size, class T>
    void assign (Size);
   template <class Size, class T>
    void assign (Size, const T&);
   allocator_type get allocator () const;

 // Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize (size_type, T);
   bool empty () const;

// Element access

   reference operator[] (size_type);
   const_reference operator[] (size_type) const;
   reference at (size_type);
   const_reference at (size_type) const;
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

 // Modifiers

   void push_front (const T&);
   void push_back (const T&);
```

```
    iterator insert (iterator);
    iterator insert (iterator, const T&);
    void insert (iterator, size_type, const T&);
    template <class InputIterator>
     void insert (iterator, InputIterator, InputIterator);

    void pop_front ();
    void pop_back ();

    iterator erase (iterator);
    iterator erase (iterator, iterator);
    void swap (deque<T, Allocator>&);
    void clear();
 };

  // Non-member Operators

 template <class T, class Allocator>
  bool operator== (const deque<T, Allocator>&,
                   const deque<T, Allocator>&);

 template <class T, class Allocator>
  bool operator!= (const deque<T, Allocator>&,
                   const deque<T, Allocator>&);


 template <class T, class Allocator>
  bool operator< (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);

 template <class T, class Allocator>
  bool operator> (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);

 template <class T, class Allocator>
  bool operator<= (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);

 template <class T, class Allocator>
  bool operator>= (const deque<T, Allocator>&,
                  const deque<T, Allocator>&);


 // Specialized Algorithms

 template <class T, class Allocator>
  voice swap (deque<T, Allocator>&, deque<T, Allocator>&);
```

explicit
deque(const Allocator& alloc = Allocator());

The default constructor. Creates a deque of zero elements. The deque will use the allocator `alloc` for all storage management.

Constructors and Destructor

```
explicit
deque(size_type n, const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of the default value for type `T`. Requires that `T` have a default constructor. The deque will use the allocator `alloc` for all storage management.

```
deque(size_type n, const T& value,
      const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of `value`. The deque will use the allocator `alloc` for all storage management.

```
deque(const deque<T, Allocator>& x);
```

Copy constructor. Creates a copy of `x`.

```
template <class InputIterator>
deque(InputIterator first, InputIterator last,
      const Allocator& alloc = Allocator());
```

Creates a deque of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The deque will use the allocator `alloc` for all storage management.

```
~deque();
```

The destructor. Releases any allocated memory for self.

Allocator

```
allocator
allocator_type get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator begin();
```

Returns a random access iterator that points to the first element.

```
const_iterator begin() const;
```

Returns a constant random access iterator that points to the first element.

```
iterator end();
```

Returns a random access iterator that points to the past-the-end value.

```
const_iterator end() const;
```

Returns a constant random access iterator that points to the past-the-end value.

```
reverse_iterator rbegin();
```

Returns a random access `reverse_iterator` that points to the past-the-end value.

```
const_reverse_iterator rbegin() const;
```

Returns a constant random access reverse iterator that points to the past-the-end value.

`reverse_iterator` **`rend`**`();`

Returns a random access `reverse_iterator` that points to the first element.

`const_reverse_iterator` **`rend`**`() const;`

Returns a constant random access reverse iterator that points to the first element.

Assignment Operator

`deque<T, Allocator>&`
`operator=``(const deque<T, Allocator>& x);`

Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to self.

Reference Operators

`reference` **`operator`**`[](size_type n);`

Returns a `reference` to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the size less one.

`const_reference` **`operator`**`[](size_type n) const;`

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size() - 1`.

Member Functions

`template <class InputIterator>`
`void`
`assign``(InputIterator first, InputIterator last);`

Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

`template <class Size, class T>`
`void`
`assign``(Size n);`

Erases all elements contained in self, then inserts `n` instances of the default value of type `T`.

`template <class Size, class T>`
`void`
`assign``(Size n, const T& t);`

Erases all elements contained in self, then inserts `n` instances of the value of `t.`

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

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the `size() - 1`.

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

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size() - 1`.

`reference`
`back``();`
Returns a reference to the last element.

`const_reference`
`back``() const;`
Returns a constant reference to the last element.

`void`
`clear``();`
Erases all elements from the self.

`bool`
`empty``() const;`
Returns `true` if the size of self is zero.

`reference`
`front``();`
Returns a reference to the first element.

`const_reference`
`front``() const;`
Returns a constant reference to the first element.

`iterator`
`erase``(iterator first, iterator last);`
Deletes the elements in the range (`first, last`). Returns an iterator pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

`iterator`
`erase``(iterator position);`
Removes the element pointed to by `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if there were no elements after the deleted range.

`iterator`
`insert``(iterator position);`
Inserts a copy of the default value of type `T` before `position`. The return value points to the inserted element. Requires that type `T` have a default constructor.

`iterator`
`insert``(iterator position, const T& x);`
Inserts `x` before `position`. The return value points to the inserted `x`.

`void`
`insert``(iterator position, size_type n, const T& x);`
Inserts `n` copies of `x` before `position`.

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

Inserts copies of the elements in the range `(first, last]` before `position`.

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

Returns `size()` of the largest possible deque.

```
void
pop_back();
```

Removes the last element. Note that this function does not return the element.

```
void
pop_front();
```

Removes the first element. Note that this function does not return the element

```
void
push_back(const T& x);
```

Appends a copy of `x` to the end.

```
void
push_front(const T& x);
```

Inserts a copy of `x` at the front.

```
void
resize(size_type sz);
```

Alters the size of self. If the new size (`sz`) is greater than the current size then `sz-size()` copies of the default value of type `T` are inserted at the end of the deque. If the new size is smaller than the current capacity, then the deque is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken. Requires that type `T` have a default constructor.

```
void
resize(size_type sz, T c);
```

Alters the size of self. If the new size (`sz`) is greater than the current size then `sz-size()` `c`'s are inserted at the end of the deque. If the new size is smaller than the current capacity, then the deque is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken.

```
size_type
size() const;
```

Returns the number of elements.

```
void
swap(deque<T,Allocator>& x);
```

Exchanges self with `x`.

Non-member Functions

```
template <class T, class Allocator>
bool operator==(const deque<T, Allocator>& x,
                 const deque<T, Allocator>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool operator!=(const deque<T, Allocator>& x,
                 const deque<T, Allocator>& y);
```

Inequality operator. Returns `true` if `x` is not the same as `y`.

```
template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

```
template <class T, class Allocator>
bool operator>(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically greater than the elements contained in `y`.

```
template <class T, class Allocator>
bool operator<=(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than or equal to the elements contained in `y`.

```
template <class T, class Allocator>
bool operator>=(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically greater than or equal to the elements contained in `y`.

```
template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x,
                const deque<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

Specialized Algorithms

```
template <class T, class Allocator>
void swap(deque<T, Allocator>& a, deque<T, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// deque.cpp
```

```
//
 #include <deque>
 #include <string>

 deque<string, allocator> deck_of_cards;
 deque<string, allocator> current_hand;

 void initialize_cards(deque<string, allocator>& cards) {
   cards.push_front("aceofspades");
   cards.push_front("kingofspades");
   cards.push_front("queenofspades");
   cards.push_front("jackofspades");
   cards.push_front("tenofspades");
   // etc.
 }

 template <class It, class It2>
 void print_current_hand(It start, It2 end)
 {
   while (start < end)
   cout << *start++ << endl;
 }


 template <class It, class It2>
 void deal_cards(It, It2 end) {
   for (int i=0;i<5;i++) {
     current_hand.insert(current_hand.begin(),*end);
     deck_of_cards.erase(end++);
   }
 }

 void play_poker() {
   initialize_cards(deck_of_cards);
   deal_cards(current_hand.begin(),deck_of_cards.begin());
 }

 int main()
 {
   play_poker();
   print_current_hand(current_hand.begin(),current_hand.end());
   return 0;
 }
```

```
Output :
aceofspades
kingofspades
queenofspades
jackofspades
tenofspades
```

Warnings

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

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

deque also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a ***deque*** in the following two ways:

```
int intarray[10];
deque<int> first_deque(intarray, intarray + 10);
deque<int> second_deque(first_deque.begin(),
                        first_deque.end());
```

But not this way:

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

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

Additionally, many compilers do not support default template arguments. If your compiler is one of these, you need to always supply the `Allocator` template argument. For instance, you'll have to write:

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

Summary

Computes the distance between two iterators

Synopsis

```
#include <iterator>

template <class ForwardIterator>
 iterator_traits<ForwardIterator>::distance_type
 distance (ForwardIterator first,
               ForwardIterator last);

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

Description

The ***distance*** template function computes the distance between two iterator. The first version returns that value, while the second version increments `n` by that value. The last iterator must be reachable from the first iterator.

Note that the second version of this function is obsolete. It is provided for backward compatibility and to support compilers that do not provide partial specialization. As you may have already deduced, the first version of the function is not available with compilers that do not support partial specialization since it depends on `iterator_traits`, which itself depends on that particular language feature.

Example

```
//
// distance.cpp
//
 #include <iterator>
 #include <vector>
 #include <iostream.h>
 int main()
 {
   //
   //Initialize a vector using an array
   //
   int arr[6] = {3,4,5,6,7,8};
   vector<int> v(arr,arr+6);
   //
   //Declare a list iterator, s.b. a ForwardIterator
   //
   vector<int>::iterator itr = v.begin()+3;
   //
   //Output the original vector
```

```
    //
    cout << "For the vector: ";
    copy(v.begin(),v.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl << endl;

    cout << "When the iterator is initialized to point to "
         << *itr << endl;
    //
    // Use of distance
    //
    vector<int>::difference_type dist = 0;
    distance(v.begin(), itr, dist);
    cout << "The distance between the beginning and itr is "
         << dist << endl;
    return 0;
  }

Output :
For the vector: 3 4 5 6 7 8
When the iterator is initialized to point to 6
The distance between the beginning and itr is 3
```

Warning

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

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

instead of:

```
vector <int>
```

Also, if your compiler does not support partial specialization then you will not be able to use the version of ***distance*** that returns the distance. Instead you'll have to use the version that increments a reference parameter.

See Also

sequence, ***random_iterator***

distance_type

Iterator primitive

Summary

Determine the type of distance used by an iterator. This function is now obsolete. It is retained in order to provide backward compatibility and support compilers that do not provide partial specialization.

Synopsis

```
#include <iterator>

template <class T, class Distance>
inline Distance* distance_type (const input_iterator<T,
      Distance>&)

template <class T, class Distance>
inline Distance* distance_type (const forward_iterator<T,
      Distance>&)

template <class T, class Distance>
inline Distance*
distance_type (const bidirectional_iterator<T, Distance>&)

template <class T, class Distance>
inline Distance*
distance_type (const random_access_iterator<T, Distance>&)

template <class T>
inline ptrdiff_t* distance_type (const T*)
```

Description

The ***distance_type*** family of function templates return a pointer to a value that is of the same type as that used to represent a distance between two iterators. The first four of these take an iterator of a particular type and return a pointer to a default value of the ***distance_type*** for that iterator. The `T*` form of the function returns `ptrdiff_t*`.

Generic algorithms use this function to create local variables of the correct type. The ***distance_type*** functions are typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,distance_type(first));
}

template <class Iterator, class Distance>
void __foo(Iterator first, Iterator last, Distance*>
{
  Distance d = Distance();
```

```
    distance(first,last,d);
    …
  }
```

The auxiliary function template allows the algorithm to extract a distance type from the first iterator and then use that type to perform some useful work.

See Also

Other iterator primitives: ***value_type, iterator_category, distance, advance***

Summary

Returns the result of dividing its first argument by its second.

Synopsis

```
#include <functional>

template <class T>
struct divides;
```

Description

divides is a binary function object. Its `operator()` returns the result of dividing `x` by `y`. You can pass a ***divides*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***divides*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vecResult.begin(),
          divides<int>());
```

After this call to ***transform***, `vecResult[n]` will contain `vec1[n]` divided by `vec2[n]`.

Interface

```
template <class T>
   struct divides : binary_function<T, T, T>
{
  typedef typename binary_function<T, T, T>::second_argument_type
                                            second_argument_type;
  typedef typename binary_function<T, T, T>::first_argument_type
                                            first_argument_type;
  typedef typename binary_function<T, T, T>::result_type
                                            result_type;

  T operator() (const T&, const T&) const;
};
```

binary_function, function objects

Summary

Compares two ranges for equality.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2>
 bool equal(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2);

template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
 bool equal(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, BinaryPredicate binary_pred);
```

Description

The ***equal*** algorithm does a pairwise comparison of all of the elements in one range with all of the elements in another range to see if they match. The first version of ***equal*** uses the equal operator (==) as the comparison function, and the second version allows you to specify a binary predicate as the comparison function. The first version returns `true` if all of the corresponding elements are equal to each other. The second version of ***equal*** returns `true` if for each pair of elements in the two ranges, the result of applying the binary predicate is `true`. In other words, ***equal*** returns `true` if both of the following are true:

1. There are at least as many elements in the second range as in the first;
2. For every iterator `i` in the range `[first1, last1)` the following corresponding conditions hold:

   ```
   *i == *(first2 + (i - first1))
   ```

   or

   ```
   binary_pred(*i, *(first2 + (i - first1))) == true
   ```

Otherwise, ***equal*** returns `false`.

This algorithm assumes that there are at least as many elements available after `first2` as there are in the range `[first1, last1)`.

Complexity

equal performs at most `last1-first1` comparisons or applications of the predicate.

Example

```
//
// equal.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main()
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,2,4,3};
   //
   // Set up two vectors
   //
   vector<int> v1(d1+0, d1 + 4), v2(d2+0, d2 + 4);

   // Check for equality
   bool b1 = equal(v1.begin(),v1.end(),v2.begin());
   bool b2 = equal(v1.begin(),v1.end(),
                   v2.begin(),equal_to<int>());

   // Both b1 and b2 are false
   cout << (b1 ? "TRUE" : "FALSE")  << " "
        << (b2 ? "TRUE" : "FALSE") << endl;
   return 0;
 }

Output :
FALSE FALSE
```

Warnings

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

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

instead of:

```
vector<int>
```

equal_range

Algorithm

Summary

Find the largest subrange in a collection into which a given value can be inserted without violating the ordering of the collection.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class T>
 pair<ForwardIterator, ForwardIterator>
 equal_range(ForwardIterator first, ForwardIterator last,
             const T& value);

 template <class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
  equal_range(ForwardIterator first, ForwardIterator last,
              const T& value, Compare comp);
```

Description

The ***equal_range*** algorithm performs a binary search on an ordered container to determine where the element `value` can be inserted without violating the container's ordering. The library provides two versions of the algorithm. The first version uses the less than operator (`operator <`) to search for the valid insertion range, and assumes that the sequence was sorted using the less than operator. The second version allows you to specify a function object of type `Compare`, and assumes that `Compare` was the function used to sort the sequence. The function object must be a binary predicate.

equal_range returns a pair of iterators, `i` and `j` that define a range containing elements equivalent to `value`, i.e., the first and last valid insertion points for `value`. If `value` is not an element in the container, `i` and `j` are equal. Otherwise, `i` will point to the first element not "less" than value, and `j` will point to the first element greater than value. In the second version, "less" is defined by the comparison object. Formally, ***equal_range*** returns a sub-range `[i, j)` such that `value` can be inserted at any iterator `k` within the range. Depending upon the version of the algorithm used, `k` must satisfy one of the following conditions:

```
!(*k <  value)  &&  !(value  <  *k)
```

or

```
comp(*k,value) == false && comp(value, *k) == false
```

The range [`first, last`) is assumed to be sorted.

Complexity

equal_range performs at most `2 * log(last - first) + 1` comparisons.

Example

```
//
// eqlrange.cpp
//
 #include <vector>
 #include <algorithm>
 #include <iostream.h>
 int main()
 {
   typedef vector<int>::iterator iterator;
   int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};
   //
   // Set up a vector
   //
   vector<int> v1(d1+0, d1 + 11);
   //
   // Try equal_range variants
   //
   pair<iterator,iterator> p1 =
       equal_range(v1.begin(),v1.end(),3);
   // p1 = (v1.begin() + 4,v1.begin() + 5)
   pair<iterator,iterator> p2 =
       equal_range(v1.begin(),v1.end(),2,less<int>());
   // p2 = (v1.begin() + 4,v1.begin() + 5)
   // Output results
   cout << endl  << "The equal range for 3 is: "
        << "( " << *p1.first << " , "
        << *p1.second << " ) " << endl << endl;
   cout << endl << "The equal range for 2 is: "
        << "( " << *p2.first << " , "
        << *p2.second << " ) " << endl;
   return 0;
 }
Output :
The equal range for 3 is: ( 3 , 4 )
The equal range for 2 is: ( 2 , 3 )
```

Warnings

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

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

instead of:

```
vector<int>
```

See Also

binary_function, lower_bound, upper_bound

Summary

Binary function object that returns `true` if its first argument equals its second

Synopsis

```
#include <functional>

template <class T>
struct equal_to;
```

Description

equal_to is a binary function object. Its `operator()` returns `true` if `x` is equal to `y`. You can pass an ***equal_to*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***equal_to*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vecResult.begin(),
          equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was equal to `vec2(n)` or a "0" if `vec1(n)` was not equal to `vec2(n)`.

Interface

```
template <class T>
    struct equal_to : binary_function<T, T, bool>
{
   typedef typename binary_function<T, T, bool>::second_argument_type
     second_argument_type;
   typedef typename binary_function<T, T, bool>::first_argument_type
     first_argument_type;
   typedef typename binary_function<T, T, bool>::result_type
     result_type;
   bool operator() (const T&, const T&) const;
};
```

See Also

binary_function, function objects

Summary

Classes supporting logic and runtime errors.

Synopsis

```
#include <exception>

class exception;
```

Description

The class ***exception*** defines the base class for the types of objects thrown as exceptions by Standard C++ Library components, and certain expressions, to report errors detected during program execution. Users can also use these exceptions to report errors in their own programs.

Interface

```
class exception {

   public:
     exception () throw();
     exception (const exception&) throw();
     exception& operator= (const exception&) throw();
     virtual ~exception () throw();
     virtual const char* what () const throw();
 };

 class logic_error : public exception {
   public:
     logic_error (const string& what_arg);
 };

 class domain_error : public logic_error {
   public:
     domain_error (const string& what_arg);
 };

 class invalid_argument : public logic_error {
   public:
     invalid_argument (const string& what_arg);
 };

 class length_error : public logic_error {
   public:
     length_error (const string& what_arg);
 };

 class out_of_range : public logic_error {
   public:
     out_of_range (const string& what_arg);
 };


 class runtime_error : public exception {
```

```
  public:
    runtime_error (const string& what_arg);
};

class range_error : public runtime_error {
  public:
    range_error (const string& what_arg);
};

class overflow_error : public runtime_error {
  public:
    overflow_error (const string& what_arg);
};

class underflow_error : public runtime_error {
  public:
    underflow_error (const string& what_arg);
};
```

Constructors

exception()
throws();

Constructs an object of class ***exception***.

exception(const exception&)
throws();

The copy constructor. Copies an `exception` object.

Destructor

virtual
~**exception**()
throws();

Destroys an object of class ***exception***.

Operators

exception&
operator=(const exception&)
throws();

The assignment operator. Copies an `exception` object.

Member Function

virtual const char*
what()const
throws();

Returns an implementation-defined, null-terminated byte string representing a human-readable message describing the exception. The message may be a null-terminated multibyte string, suitable for conversion and display as a `wstring`.

Constructors for Derived Classes

logic_error::**logic_error**(const string& what_arg);

Constructs an object of class `logic_error`.

domain_error::**domain_error**(const string& what_arg);

Constructs an object of class `domain_error.`

```
invalid_argument::invalid_argument(const string& what_arg);
```
Constructs an object of class `invalid_argument`.

```
length_error::length_error(const string& what_arg);
```
Constructs an object of class `length_error`.

```
out_of_range::out_of_range(const string& what_arg);
```
Constructs an object of class `out_of_range`.

```
runtime_error::runtime_error(const string& what_arg);
```
Constructs an object of class `runtime_error`.

```
range_error::range_error(const string& what_arg);
```
Constructs an object of class `range_error`.

```
overflow_error::overflow_error(const string& what_arg);
```
Constructs an object of class `overflow_error`.

```
underflow_error::underflow_error(
                      const string& what_arg);
```
Constructs an object of class `underflow_error`.

Example

```
//
// exception.cpp
//
#include <iostream.h>
#include <stdexcept>

static void f() { throw runtime_error("a runtime error"); }

int main ()
{
   //
   // By wrapping the body of main in a try-catch block
   // we can be assured that we'll catch all exceptions
   // in the exception hierarchy.  You can simply catch
   // exception as is done below, or you can catch each
   // of the exceptions in which you have an interest.
   //
   try
   {
       f();
   }
   catch (const exception& e)
   {
       cout << "Got an exception: " << e.what() << endl;
   }
   return 0;
}
```

fill, fill_n

Algorithm

Summary

Initializes a range with a given value.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last,
            const T& value);

template <class OutputIterator, class Size, class T>
  void fill_n(OutputIterator first, Size n, const T& value);
```

Description

The ***fill*** and ***fill_n*** algorithms are used to assign a value to the elements in a sequence. ***fill*** assigns the value to all the elements designated by iterators in the range `[first, last)`.

The ***fill_n*** algorithm assigns the value to all the elements designated by iterators in the range `[first, first + n)`. ***fill_n*** assumes that there are at least `n` elements following `first`, unless `first` is an insert iterator.

Complexity

fill makes exactly `last - first` assignments, and ***fill_n*** makes exactly `n` assignments.

Example

```
//
// fill.cpp
//
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
  int d1[4] = {1,2,3,4};
  //
  // Set up two vectors
  //
  vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
  //
  // Set up one empty vector
  //
  vector<int> v3;
  //
  // Fill all of v1 with 9
  //
  fill(v1.begin(),v1.end(),9);
```

```
   //
   // Fill first 3 of v2 with 7
   //
   fill_n(v2.begin(),3,7);

   //
   // Use insert iterator to fill v3 with 5 11's
   //
   fill_n(back_inserter(v3),5,11);
   //
   // Copy all three to cout
   //
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;
   copy(v3.begin(),v3.end(),out);
   cout << endl;
   //
   // Fill cout with 3 5's
   //
   fill_n(ostream_iterator<int,char>(cout," "),3,5);
   cout << endl;

   return 0;
 }

Output :
9 9 9 9
7 7 7 4
11 11 11 11 11
5 5 5
```

Warnings

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

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

instead of:

```
vector<int>
```

Algorithm

Summary

Find an occurrence of value in a sequence

Synopsis

```
#include <algorithm>

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

Description

The ***find*** algorithm lets you search for the first occurrence of a particular value in a sequence. ***find*** returns the first iterator `i` in the range `[first, last)` for which the following condition holds:

```
*i == value.
```

If ***find*** does not find a match for `value`, it returns the iterator `last`.

Complexity

find performs at most `last-first` comparisons.

Example

```
//
// find.cpp
//
 #include <vector>
 #include <algorithm>

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

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

   // Try find
   iterator it1 = find(v1.begin(),v1.end(),3);
   // it1 = v1.begin() + 4;

   // Try find_if
   iterator it2 =
      find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
   // it2 = v1.begin() + 4

   // Try both adjacent_find variants
   iterator it3 = adjacent_find(v1.begin(),v1.end());
   // it3 = v1.begin() +2
```

```
   iterator it4 =
      adjacent_find(v1.begin(),v1.end(),equal_to<int>());
   // v4 = v1.begin() + 2

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

   return 0;
 }
```

```
Output : 3 3 2 2
```

Warning

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

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

instead of:

```
vector<int>
```

See Also

adjacent_find, ***find_first_of***, ***find_if***

find_end

Algorithm

Summary Finds the last occurrence of a sub-sequence in a sequence.

Synopsis

```
#include <algorithm>

template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 find_end(ForwardIterator1 first1,
                            ForwardIterator1 last1,
                            ForwardIterator2 first2,
                            ForwardIterator2 last2);
  template <class Forward Iterator1, class ForwardIterator2,
            class BinaryPredicate>
    ForwardIterator1 find_end(ForwardIterator1 first1,
                              ForwardIterator1 last1,
                              ForwardIterator2 first2,
                              ForwardIterator2 last2,
                              BinaryPredicate pred);
```

Description

The ***find_end*** algorithm finds the last occurrence of a sub-sequence, indicated by `[first2, last2)`, in a sequence, `[first1,last1)`. The algorithm returns an iterator pointing to the first element of the found sub-sequence, or `last1` if no match is found.

More precisely, the ***find_end*** algorithm returns the last iterator `i` in the range `[first1, last1 - (last2-first2))` such that for any non-negative integer `n < (last2-first2)`, the following corresponding conditions hold:

```
*(i+n)  ==  *(first2+n),
pred(*(i+n),*(first2+n)) == true.
```

Or returns `last1` if no such iterator is found.

Two versions of the algorithm exist. The first uses the equality operator as the default binary predicate, and the second allows you to specify a binary predicate.

Complexity At most `(last2-first2)*(last1-first1-(last2-first2)+1)` applications of the corresponding predicate are done.

Example

```
//
// find_end.cpp
//
#include<vector>
#include<iterator>
#include<algorithm>
#include<iostream.h>
```

```
int main()
{
   typedef vector<int>::iterator iterator;
   int d1[10] = {0,1,6,5,3,2,2,6,5,7};
   int d2[4] = {6,5,0,0}
   //
   // Set up two vectors.
   //
   vector<int> v1(d1+0, d1+10), v2(d2+0, d2+2);
   //
   // Try both find_first_of variants.
   //
   iterator it1 = find_first_of (v1.begin(), v1.end(), v2.begin(),
                                 v2.end());
   iterator it2 = find_first_of (v1.begin(), v1.end(), v2.begin(),
                                 v2.end(), equal_to<int>());
   //
   // Try both find_end variants.
   //
   iterator it3 = find_end (v1.begin(), v1.end(), v2.begin(),
                            v2.end());
   iterator it4 = find_end (v1.begin(), v1.end(), v2.begin(),
                            v2.end(), equal_to<int>());
   //
   // Output results of find_first_of.
   // Iterator now points to the first element that matches one of
   // a set of values
   //
   cout << "For the vectors: ";
   copy (v1.begin(), v1.end(), ostream_iterator<int>(cout," "));
   cout << " and ";
   copy (v2.begin(), v2.end(), ostream_iterator<int>(cout," "));
   cout<< endl ,, endl
       << "both versions of find_first_of point to: "
       << *it1 << endl << "with first_of address = " << it1
       << endl ;
   //
   //Output results of find_end.
   // Iterator now points to the first element of the last find
   //sub-sequence.
   //
   cout << endl << endl
        << "both versions of find_end point to: "
        << *it3 << endl << "with find_end address = " << it3
        << endl ;

   return 0;
}

Output :
For the vectors: 0 1 6 5 3 2 2 6 5 7  and 6 5
both versions of find_first_of point to: 6
with first_of address = 0x100005c0
both versions of find_end point to: 6
with find_end address = 0x100005d4
```

Warnings

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

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

instead of:

```
vector<int>
```

See Also

Algorithms, find, find_if, adjacent_find

Summary

Finds the first occurrence of any value from one sequence in another sequence.

```
#include <algorithm>
```

Synopsis

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

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

Description

The ***find_first_of*** algorithm finds a the first occurrence of a value from a sequence, specified by `first2, last2`, in a sequence specified by `first1, last1`. The algorithm returns an iterator in the range `[first1, last1)` that points to the first matching element. If the first sequence `[first1, last1)` does not contain any of the values in the second sequence, ***find_first_of*** returns `last1`.

In other words, ***find_first_of*** returns the first iterator `i` in the `[first1, last1)`such that for some integer `j` in the range `[first2, last2)`:the following conditions hold:

```
*i == *j, pred(*i,*j) == true.
```

Or ***find_first_of*** returns `last1` if no such iterator is found.

Two versions of the algorithm exist. The first uses the equality operator as the default binary predicate, and the second allows you to specify a binary predicate.

Complexity

At most `(last1 - first1)*(last2 - first2)` applications of the corresponding predicate are done.

```
//
// find_f_o.cpp
//
 #include <vector>
```

Example

```
#include <iterator>
#include <algorithm>
#include <iostream.h>

int main()
{
  typedef vector<int>::iterator iterator;
  int d1[10] = {0,1,2,2,3,4,2,2,6,7};
  int d2[2] = {6,4};
  //
  // Set up two vectors
  //
  vector<int> v1(d1,d1 + 10), v2(d2,d2 + 2);
  //
  // Try both find_first_of variants
  //
  iterator it1 =
    find_first_of(v1.begin(),v1.end(),v2.begin(),v2.end());
  find_first_of(v1.begin(),v1.end(),v2.begin(),v2.end(),
                equal_to<int>());
  //
  // Output results
  //
  cout << "For the vectors: ";
  copy(v1.begin(),v1.end(),
       ostream_iterator<int,char>(cout," " ));
  cout << " and ";
  copy(v2.begin(),v2.end(),
       ostream_iterator<int,char>(cout," " ));
  cout << endl << endl
      << "both versions of find_first_of point to: "
      << *it1;

  return 0;
}

Output :
For the vectors: 0 1 2 2 3 4 2 2 6 7  and 6 4
both versions of find_first_of point to: 4
```

Warnings

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

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

instead of:

```
vector<int>
```

See Also

Algorithms, adjacent_find, find, find_if, find_next, find_end

Summary

Find an occurrence of value in a sequence that satisfies a specifed predicate.

Synopsis

```
#include <algorithm>

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

Description

The ***find_if*** algorithm allows you to search for the first element in a sequence that satisfies a particular condition. The sequence is defined by iterators `first` and `last`, while the condition is defined by the third argument: a predicate function that returns a boolean value. ***find_if*** returns the first iterator `i` in the range `[first, last)` for which the following condition holds:

```
pred(*i) == true.
```

If no such iterator is found, ***find_if*** returns `last`.

Complexity

find_if performs at most `last-first` applications of the corresponding predicate.

Example

```
/
// find.cpp
//
#include <vector>
#include <algorithm>
#include <iostream.h>

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

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

   // Try find
   iterator it1 = find(v1.begin(),v1.end(),3);
   // it1 = v1.begin() + 4;

   // Try find_if
   iterator it2 =
      find_if(v1.begin(),v1.end(),bind1st(equal_to<int>(),3));
   // it2 = v1.begin() + 4
```

```
   // Try both adjacent_find variants
   iterator it3 = adjacent_find(v1.begin(),v1.end());
   // it3 = v1.begin() +2

   iterator it4 =
      adjacent_find(v1.begin(),v1.end(),equal_to<int>());
   // v4 = v1.begin() + 2

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

   return 0;
 }
```

```
Output : 3 3 2 2
```

Warning

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

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

instead of:

```
vector<int>
```

See Also

adjacent_find, ***Algorithms***, ***find***, ***find_end***, ***find_first_of***

Summary

Applies a function to each element in a range.

Synopsis

```
#include <algorithm>

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

Description

The ***for_each*** algorithm applies function `f` to all members of the sequence in the range `[first, last)`, where `first` and `last` are iterators that define the sequence. Since this a non-mutating algorithm, the function `f` cannot make any modifications to the sequence, but it can achieve results through side effects (such as copying or printing). If `f` returns a result, the result is ignored.

Complexity

The function `f` is applied exactly `last - first` times.

Example

```
//
 // for_each.cpp
 //
#include <vector>
#include <algorithm>
#include <iostream.h>

 // Function class that outputs its argument times x
 template <class Arg>
 class out_times_x :  private unary_function<Arg,void>
 {
   private:
      Arg multiplier;

   public:
      out_times_x(const Arg& x) : multiplier(x) { }
      void operator()(const Arg& x)
         { cout << x * multiplier << " " << endl; }
 };

 int main()
 {
   int sequence[5] = {1,2,3,4,5};


   // Set up a vector
```

```
    vector<int> v(sequence,sequence + 5);

    // Setup a function object
    out_times_x<int> f2(2);

    for_each(v.begin(),v.end(),f2);   // Apply function

    return 0;
  }

Output : 2 4 6 8 10
```

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

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

instead of:

```
vector<int>
```

See Also ***Algorithms***, ***function object***

Summary

A forward-moving iterator that can both read and write.

Description

For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Forward iterators are forward moving, and have the ability to both read and write data. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Forward Iterators

The following expressions must be valid for forward iterators:

`X u`	`u` might have a singular value
`X()`	`X()` might be singular
`X(a)`	copy constructor, `a == X(a)`.
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`

`*r++`	returns `T&`

Forward iterators have the condition that `a == b` implies `*a == *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

See Also ***Iterators***, ***Bidirectional Iterators***

Summary

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

Synopsis

```
#include <iterator>

template <class Container>
class front_insert_iterator : public output_iterator ;
```

Description

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

Note that a ***front_insert_iterator*** makes each element that it inserts the new front of the container. This has the effect of reversing the order of the inserted elements. For example, if you use a ***front_insert_iterator*** to insert "1" then "2" then "3" onto the front of container `exmpl`, you will find, after the three insertions, that the first three elements of `exmpl` are "3 2 1".

Interface

```
template <class Container>
 class front_insert_iterator : public output_iterator {

public:
   explicit front_insert_iterator (Container&);
   front_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   front_insert_iterator<Container>& operator* ();
   front_insert_iterator<Container>& operator++ ();
   front_insert_iterator<Container> operator++ (int);
};

 template <class Container>
  front_insert_iterator<Container> front_inserter (Container&);
```

Constructor

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

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

Operators

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

Assignment Operator. Inserts a copy of `value` on the front of the container, and returns `*this`.

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

Returns `*this` (the input iterator itself).

```
front_insert_iterator<Container>&
operator++();
```

```
front_insert_iterator<Container>
operator++(int);
```

Increments the insert iterator and returns `*this`.

Non-member Function

```
template <class Container>
front_insert_iterator<Container>
front_inserter(Container& x)
```

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

Example

```
//
// ins_itr.cpp
//
#include <iterator>
#include <deque>
#include <iostream.h>

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

```
    //
    // Insert d2 at front of d.
    //
    copy(d2.begin(), d2.end(), front_inserter(d));
    //
    // Output the new deque.
    //
    cout << endl << endl;
    cout << "Use a front_inserter: " << endl << "     ";
    copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
    //
    // Insert d2 at back of d.
    //
    copy(d2.begin(), d2.end(), back_inserter(d));
    //
    // Output the new deque.
    //
    cout << endl << endl;
    cout << "Use a back_inserter: " << endl << "     ";
    copy(d.begin(), d.end(), ostream_iterator<int>(cout," "));
    cout << endl;

    return 0;
  }

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

Warnings

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

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

See Also

Insert Iterators

Summary

Objects with an operator() defined. Function objects are used in place of pointers to functions as arguments to templated algorithms.

Synopsis

```
#include<functional>

// typedefs

  template <class Arg, class Result>
   struct unary_function;

  template <class Arg1, class Arg2, class Result>
   struct binary_function;
```

Description

Function objects are objects with an operator() defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an operator() defined, or a pointer to a function. The Standard C++ Library provides both a standard set of function objects, and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take one argument are called *unary function objects*. Unary function objects are required to provide the typedefs argument_type and result_type. Similarly, function objects that take two arguments are called *binary function objects* and, as such, are required to provide the typedefs first_argument_type, second_argument_type, and result_type.

The classes unary_function and binary_function make the task of creating templated function objects easier. The necessary typedefs for a unary or binary function object are provided by inheriting from the appropriate function object class.

The function objects provided by the standard library are listed below, together with a brief description of their operation. This class reference also includes an alphabetic entry for each function.

Name	Operation
arithmetic functions	
plus	addition x + y
minus	subtraction x - y
multiplies	multiplication x * y
divides	division x / y
modulus	remainder x % y
negate	negation - x
comparison functions	
equal_to	equality test x == y
not_equal_to	inequality test x != y
greater	greater comparison x > y
less	less-than comparison x < y
greater_equal	greater than or equal comparison x >= y
less_equal	less than or equal comparison x <= y
logical functions	
logical_and	logical conjunction x && y
logical_or	logical disjunction x || y
logical_not	logical negation ! x

Interface

```
template <class Arg, class Result>
struct unary_function{
     typedef Arg argument_type;
     typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function{
     typedef Arg1 first_argument_type;
     typedef Arg2 second_argument_type;
     typedef Result result_type;
};

// Arithmetic Operations

template<class T>
struct plus : binary_function<T, T, T> {
     T operator() (const T&, const T&) const;
};

template <class T>
struct minus : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};
```

```
template <class T>
struct multiplies : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct divides : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct modulus : binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};

template <class T>
struct negate : unary_function<T, T> {
        T operator() (const T&) const;
};

 // Comparisons

template <class T>
struct equal_to : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct not_equal_to : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct greater : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct less : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct greater_equal : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct less_equal : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 // Logical Comparisons

 template <class T>
 struct logical_and : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
```

```
};

 template <class T>
 struct logical_or : binary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};

 template <class T>
 struct logical_not : unary_function<T, T, bool> {
         bool operator() (const T&, const T&) const;
};
```

Example

```
//
// funct_ob.cpp
//
 #include<functional>
 #include<deque>
 #include<vector>
 #include<algorithm>
 #include <iostream.h>

 //Create a new function object from unary_function
 template<class Arg>
 class factorial : public unary_function<Arg, Arg>
 {
   public:

   Arg operator()(const Arg& arg)
   {
     Arg a = 1;
     for(Arg i = 2; i <= arg; i++)
       a *= i;
     return a;
   }
 };

 int main()
 {
   //Initialize a deque with an array of ints
   int init[7] = {1,2,3,4,5,6,7};
   deque<int> d(init, init+7);

   //Create an empty vector to store the factorials
   vector<int> v((size_t)7);

   //Transform the numbers in the deque to their factorials and
   // store in the vector
   transform(d.begin(), d.end(), v.begin(), factorial<int>());

   //Print the results
   cout << "The following numbers: " << endl << "     ";
   copy(d.begin(),d.end(),ostream_iterator<int,char>(cout," "));

   cout << endl << endl;
   cout << "Have the factorials: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
```

```
    return 0;
  }

Output :
The following numbers:
     1 2 3 4 5 6 7
Have the factorials:
     1 2 6 24 120 720 5040
```

Warnings

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

`vector<int, allocator<int> > and deque<int, allocator<int> >`
instead of :

`vector<int> and deque<int>`

See Also ***binary_function, unary_function***

Summary

Initialize a container with values produced by a value-generator class.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last,
                Generator gen);

template <class OutputIterator, class Size, class Generator>
  void generate_n(OutputIterator first, Size n, Generator gen);
```

Description

A value-generator function returns a value each time it is invoked. The algorithms ***generate*** and ***generate_n*** initialize (or reinitialize) a sequence by assigning the return value of the generator function `gen` to all the elements designated by iterators in the range `[first, last)` or `[first, first + n)`. The function `gen` takes no arguments. (`gen` can be a function or a class with an `operator ()` defined that takes no arguments.)

generate_n assumes that there are at least `n` elements following `first`, unless `first` is an insert iterator.

Complexity

The ***generate*** and ***generate_n*** algorithms invoke `gen` and assign its return value exactly `last - first` (or `n`) times.

Example

```
//
// generate.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 // Value generator simply doubles the current value
 // and returns it
 template <class T>
 class generate_val
 {
   private:
      T val_;
   public:
      generate_val(const T& val) : val_(val) {}
      T& operator()() { val_ += val_; return val_; }
 };

 int main()
 {
   int d1[4] = {1,2,3,4};
```

```
    generate_val<int> gen(1);

    // Set up two vectors
    vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
    // Set up one empty vector
    vector<int> v3;

    // Generate values for all of v1
    generate(v1.begin(),v1.end(),gen);

    // Generate values for first 3 of v2
    generate_n(v2.begin(),3,gen);

    // Use insert iterator to generate 5 values for v3
    generate_n(back_inserter(v3),5,gen);

    // Copy all three to cout
    ostream_iterator<int,char> out(cout," ");
    copy(v1.begin(),v1.end(),out);
    cout << endl;
    copy(v2.begin(),v2.end(),out);
    cout << endl;
    copy(v3.begin(),v3.end(),out);
    cout << endl;

    // Generate 3 values for cout
    generate_n(ostream_iterator<int>(cout," "),3,gen);
    cout << endl;

    return 0;
  }

Output :
2 4 8 16
2 4 8 4
2 4 8 16 32
2 4 8
```

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

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

instead of :

```
vector<int>
```

See Also ***function objects***

Summary

Pointer based primitive for handling memory

Synopsis

```
#include <memory>

template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer (ptrdiff_t, T*);
```

Description

The ***get_temporary_buffer*** templated function reserves from system memory the largest possible buffer that is less than or equal to the size requested (`n*sizeof(T)`), and returns a `pair<T*, ptrdiff_t>` containing the address and size of that buffer. The units used to describe the capacity are in `sizeof(T)`.

See Also

allocate, ***construct***, ***deallocate***, ***pair***, ***return_temporary_buffer***.

greater

Function Object

Summary

Binary function object that returns `true` if its first argument is greater than its second.

Synopsis

```
#include <functional>

template <class T>
struct greater : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                                 second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                                 first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                                 result_type;
  bool operator() (const T&, const T&) const;
};
```

Description

greater is a binary function object. Its `operator()` returns `true` if `x` is greater than `y`. You can pass a ***greater*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***greater*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),  vecResult.begin(), greater<int>());
```

Warnings

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was greater than `vec2(n)` or a "0" if `vec1(n)` was less than or equal to `vec2(n)`.

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

`vector<int, allocator<int> >`

instead of

`vector<int>`

See Also

function objects

Summary

Binary function object that returns `true` if its first argument is greater than or equal to its second

Synopsis

```
#include <functional>

template <class T>
struct greater_equal ; : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                                second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                                first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                                result_type;
  bool operator() (const T&, const T&) const;
};
```

Description

greater_equal is a binary function object. Its `operator()` returns `true` if `x` is greater than or equal to `y`. You can pass a ***greater_equal*** object to any algorithm that requires a binary function. For example, the ***sort*** algorithm can accept a binary function as an alternate comparison object to sort a sequence. ***greater_equal*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
.
.
sort(vec1.begin(), vec1.end(),greater_equal<int>());
```

After this call to ***sort***, `vec1` will be sorted in descending order.

Warnings

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

`vector<int, allocator<int> >`

instead of

`vector<int>`

See Also

function objects

Heap Operations

Algorithm

See the entries for ***make_heap***, ***pop_heap***, ***push_heap*** and ***sort_heap***

includes

Algorithm

Summary

Basic set operation for sorted sequences.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2>
 bool includes (InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
 bool includes (InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                Compare comp);
```

Description

The ***includes*** algorithm compares two sorted sequences and returns `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. It returns `false` otherwise. ***include*** assumes that the sequences are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
 //
 // includes.cpp
 //
  #include <algorithm>
  #include <set>
  #include <iostream.h>

  int main()
  {

    //Initialize some sets
    int a1[10] = {1,2,3,4,5,6,7,8,9,10};
    int a2[6]  = {2,4,6,8,10,12};
    int a3[4]  = {3,5,7,8};
    set<int, less<int> > all(a1, a1+10), even(a2, a2+6),
                         small(a3,a3+4);

   //Demonstrate includes
   cout << "The set: ";
   copy(all.begin(),all.end(),
         ostream_iterator<int,char>(cout," "));
   bool answer = includes(all.begin(), all.end(),
                 small.begin(), small.end());
   cout << endl
        << (answer ? "INCLUDES " : "DOES NOT INCLUDE ");
   copy(small.begin(),small.end(),
```

```
        ostream_iterator<int,char>(cout," "));
  answer = includes(all.begin(), all.end(),
                    even.begin(), even.end());
  cout << ", and" << endl
       << (answer ? "INCLUDES" : "DOES NOT INCLUDE ");
  copy(even.begin(),even.end(),
        ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  return 0;
 }

Output :
The set: 1 2 3 4 5 6 7 8 9 10
INCLUDES 3 5 7 8 , and
DOES NOT INCLUDE 2 4 6 8 10 12
```

Warnings

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

`set<int, less<int>, allocator<int> >`

instead of

`set<int>`

See Also

set, set_union, ***set_intersection***, ***set_difference***, ***set_symmetric_difference***

Generalized Numeric Operation

Summary

Computes the inner product A X B of two ranges A and B.

Synopsis

```
#include <numeric>
template <class InputIterator1, class InputIterator2,
          class T>
T inner_product (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, T init);
template <class InputIterator1, class InputIterator2,
          class T,
          class BinaryOperation1,
          class BinaryOperation2>
T inner_product (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, T init,
                 BinaryOperation1 binary_op1,
                 BinaryOperation2 binary_op2);
```

Description

There are two versions of ***inner_product***. The first computes an inner product using the default multiplication and addition operators, while the second allows you to specify binary operations to use in place of the default operations.

The first version of the function computes its result by initializing the accumulator acc with the initial value init and then modifying it with:

```
acc = acc + ((*i1) * (*i2))
```

for every iterator i1 in the range [first1, last1) and iterator i2 in the range [first2, first2 + (last1 - first1)). The algorithm returns acc.

The second version of the function initializes acc with init, then computes the result:

```
acc  =  binary_op1(acc, binary_op2(*i1,  *i2))
```

for every iterator i1 in the range [first1, last1) and iterator i2 in the range [first2, first2 + (last1 - first1)).

Complexity

The ***inner_product*** algorithm computes exactly (last1 - first1) applications of either:

```
acc + (*i1) * (*i2)
```

or

```
binary_op1(acc, binary_op2(*i1, *i2)).
```

Example

```
//
// inr_prod.cpp
```

```
//
 #include <numeric>       //For inner_product
 #include <list>          //For list
 #include <vector>        //For vectors
 #include <functional>    //For plus and minus
 #include <iostream.h>
 int main()
 {
   //Initialize a list and an int using arrays of ints
   int a1[3] = {6, -3, -2};
   int a2[3] = {-2, -3, -2};
   list<int>   l(a1, a1+3);
   vector<int> v(a2, a2+3);
   //Calculate the inner product of the two sets of values
   int inner_prod =
        inner_product(l.begin(), l.end(), v.begin(), 0);
   //Calculate a wacky inner product using the same values
   int wacky =
         inner_product(l.begin(), l.end(), v.begin(), 0,
                      plus<int>(), minus<int>());
   //Print the output
   cout << "For the two sets of numbers: " << endl
        << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << " and  ";
   copy(l.begin(),l.end(),ostream_iterator<int,char>(cout," "));
   cout << "," << endl << endl;
   cout << "The inner product is: " << inner_prod << endl;
   cout << "The wacky result is: " << wacky << endl;
   return 0;
 }
Output :
For the two sets of numbers:
     -2 -3 -2
 and  6 -3 -2 ,
The inner product is: 1
The wacky result is: 8
```

Warnings

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

`list<int, allocator<int> >` and `vector<int, allocator<int> >`

instead of

`list<int>` and `vector<int>`

Summary

Merge two sorted sequences into one.

Synopsis

```
#include <algorithm>
template <class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
```

Description

The ***inplace_merge*** algorithm merges two sorted consecutive ranges `[first, middle)` and `[middle, last)`, and puts the result of the merge into the range `[first, last)`. The merge is stable, that is, if the two ranges contain equivalent elements, the elements from the first range always precede the elements from the second.

There are two versions of the ***inplace_merge*** algorithm. The first version uses the less than operator (`operator<`) as the default for comparison, and the second version accepts a third argument that specifies a comparison operator.

Complexity

When enough additional memory is available, ***inplace_merge*** does at most `(last - first) - 1` comparisons. If no additional memory is available, an algorithm with `O(NlogN)` complexity (where `N` is equal to `last-first`) may be used.

Example

```
//
// merge.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main()
 {
   int d1[4] = {1,2,3,4};
   int d2[8] = {11,13,15,17,12,14,16,18};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
```

```
   // Set up four destination vectors
   vector<int> v3(d2,d2 + 8),v4(d2,d2 + 8),
               v5(d2,d2 + 8),v6(d2,d2 + 8);
   // Set up one empty vector
   vector<int> v7;

   // Merge v1 with v2
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());
   // Now use comparator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v4.begin(),
         less<int>());

   // In place merge v5
   vector<int>::iterator mid = v5.begin();
   advance(mid,4);
   inplace_merge(v5.begin(),mid,v5.end());
   // Now use a comparator on v6
   mid = v6.begin();
   advance(mid,4);
   inplace_merge(v6.begin(),mid,v6.end(),less<int>());

   // Merge v1 and v2 to empty vector using insert iterator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         back_inserter(v7));

   // Copy all cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;
   copy(v3.begin(),v3.end(),out);
   cout << endl;
   copy(v4.begin(),v4.end(),out);
   cout << endl;
   copy(v5.begin(),v5.end(),out);
   cout << endl;
   copy(v6.begin(),v6.end(),out);
   cout << endl;
   copy(v7.begin(),v7.end(),out);
   cout << endl;

   // Merge v1 and v2 to cout
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }

Output:
1 2 3 4
1 2 3 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
11 12 13 14 15 16 17 18
11 12 13 14 15 16 17 18
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
```

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

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

instead of

```
vector<int>
```

See Also ***merge***

input iterator

Iterator

Summary

A read-only, forward moving iterator.

Description

For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Input iterators are read-only, forward moving iterators that satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirement descriptions listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Input Iterators

The following expressions must be valid for input iterators:

`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b`, `a != b`	return value convertible to `bool`
`*a`	`a == b` implies `*a == *b`
`++r`	returns `X&`
`r++`	return `value` convertible to `const X&`
`*r++`	returns type `T`
`a -> m`	returns `(*a).m`

For input iterators, `a == b` does not imply that `++a == ++b`.

Algorithms using input iterators should be single pass algorithms. That is they should not pass through the same iterator twice.

The value of type `T` does not have to be an `lvalue`.

See Also ***iterators*, *output iterators***

Insert Iterator

Summary

Iterator adaptor that allows an iterator to insert into a container rather than overwrite elements in the container.

Synopsis

```
#include <iterator>

template <class Container>
class insert_iterator : public output_iterator;

template <class Container>
class back_insert_iterator:public output_iterator;

template <class Container>
class front_insert_iterator : public output_iterator;
```

Description

Insert iterators are iterator adaptors that let an iterator *insert* new elements into a collection rather than overwrite existing elements when copying to a container. There are several types of insert iterator classes.

- The class ***back_insert_iterator*** is used to insert items at the end of a collection. The function `back_inserter` can be used with an iterator inline, to create an instance of a ***back_insert_iterator*** for a particular collection type.
- The class ***front_insert_iterator*** is used to insert items at the start of a collection. The function `front_inserter` creates an instance of a ***front_insert_iterator*** for a particular collection type.
- An ***insert_iterator*** inserts new items into a collection at a location defined by an iterator supplied to the constructor. Like the other insert iterators, ***insert_iterator*** has a helper function called `inserter`, which takes a collection and an iterator into that collection, and creates an instance of the ***insert_iterator***.

Interface

```
template <class Container>
 class insert_iterator : public output_iterator {

public:
   insert_iterator (Container&, typename Container::iterator);
   insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   insert_iterator<Container>& operator* ();
   insert_iterator<Container>& operator++ ();
   insert_iterator<Container>& operator++ (int);
};

template <class Container>
```

```
 class back_insert_iterator : public output_iterator {

public:
   explicit back_insert_iterator (Container&);
   back_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   back_insert_iterator<Container>& operator* ();
   back_insert_iterator<Container>& operator++ ();
   back_insert_iterator<Container> operator++ (int);
};

template <class Container>
 class front_insert_iterator : public output_iterator {

public:
   explicit front_insert_iterator (Container&);
   front_insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   front_insert_iterator<Container>& operator* ();
   front_insert_iterator<Container>& operator++ ();
   front_insert_iterator<Container> operator++ (int);
};

 template <class Container, class Iterator>
 insert_iterator<Container> inserter (Container&, Iterator);

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

 template <class Container>
 front_insert_iterator<Container> front_inserter (Container&);
```

See Also ***back_insert_iterator*, *front_insert_iterator, insert_iterator***

Summary

An insert iterator used to insert items into a collection rather than overwrite the collection.

Synopsis

```
#include <iterator>

template <class Container>
class insert_iterator : public output_iterator;
```

Description

Insert iterators let you *insert* new elements into a collection rather than copy a new element's value over the value of an existing element. The class ***insert_iterator*** is used to insert items into a specified location of a collection. The function `inserter` creates an instance of an ***insert_iterator*** given a particular collection type and iterator. An ***insert_iterator*** can be used with ***vector***s, ***deque***s, ***list***s, ***map***s and ***set***s.

Interface

```
template <class Container>
class insert_iterator : public output_iterator {

public:
   insert_iterator (Container&, typename Container::iterator);
   insert_iterator<Container>&
    operator= (const typename Container::value_type&);
   insert_iterator<Container>& operator* ();
   insert_iterator<Container>& operator++ ();
   insert_iterator<Container>& operator++ (int);
};

template <class Container, class Iterator>
insert_iterator<Container> inserter (Container&, Iterator)
```

Constructor

```
insert_iterator(Container& x, typename Container::iterator i);
```

Constructor. Creates an instance of an ***insert_iterator*** associated with container `x` and iterator `i`.

Operators

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

Assignment operator. Inserts a copy of `value` into the container at the location specified by the `insert_iterator`, increments the iterator, and returns `*this`.

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

Returns `*this` (the input iterator itself).

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

Increments the insert iterator and returns `*this`.

Non-member Function

```
template <class Container, class Iterator>
insert_iterator<Container>
inserter(Container& x, Iterator i);
```

Returns an ***insert_iterator*** that will insert elements into container `x` at location `i`. This function allows you to create insert iterators inline.

Example

```
#include <iterator>
#include <vector>
#include <iostream.h>

int main()
{
  //Initialize a vector using an array
  int arr[4] = {3,4,7,8};
  vector<int> v(arr,arr+4);

  //Output the original vector
  cout << "Start with a vector: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

  //Insert into the middle
  insert_iterator<vector<int> >  ins(v, v.begin()+2);
  *ins = 5;
  *ins = 6;

  //Output the new vector
  cout << endl << endl;
  cout << "Use an insert_iterator: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

  return 0;
}
```

Warnings

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

`vector<int, allocator<int> >`

instead of:

`vector<int>`

See Also

back_insert_iterator, front_insert_iterator, Insert Iterators

Summary

Stream iterator that provides iterator capabilities for istreams. This iterator allows generic algorithms to be used directly on streams.

Synopsis

```
#include <iterator>

template <class T, class charT, class traits = ios_traits<charT>,
          class Distance = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag,
                                     T,Distance>;
```

Description

Stream iterators provide the standard iterator interface for input and output streams.

The class ***istream_iterator*** reads elements from an input stream (using operator `>>`). A value of type `T` is retrieved and stored when the iterator is constructed and each time `operator++` is called. The iterator will be equal to the end-of-stream iterator value if the end-of-file is reached. Use the constructor with no arguments to create an end-of-stream iterator. The only valid use of this iterator is to compare to other iterators when checking for end of file. Do not attempt to dereference the end-of-stream iterator; it plays the same role as the past-the-end iterator provided by the `end()` function of containers. Since an ***istream_iterator*** is an input iterator, you cannot assign to the value returned by dereferencing the iterator. This also means that ***istream_iterators*** can only be used for single pass algorithms.

Since a new value is read every time the `operator++` is used on an ***istream_iterator***, that operation is not equality-preserving. This means that `i == j` does *not* mean that `++i == ++j` (although two end-of-stream iterators are always equal).

Interface

```
 template <class T, class charT, class traits = ios_traits<charT>
           class Distance = ptrdiff_t>
  class istream_iterator : public iterator<input_iterator_tag,
                                        T, Distance>
  {

  public:
     typedef T value_type;
     typedef charT char_type;
     typedef traits traits_type;
     typedef basic_istream<charT,traits> istream_type;

     istream_iterator();
     istream_iterator (istream_type&);
     istream_iterator
           (const stream_iterator<T,charT,traits,Distance>&);
     ~istream_itertor ();

     const T& operator*() const;
     const T* operator ->() const;
```

```
     istream_iterator <T,charT,traits,Distance>& operator++();
     istream_iterator <T,charT,traits,Distance>  operator++ (int)
  };

 // Non-member Operators

 template <class T, class charT, class traits, class Distance>
 bool operator==(const istream_iterator<T,charT,traits,Distance>&,
                 const istream_iterator<T,charT,traits,Distance>&);

 template <class T, class charT, class traits,  class Distance>
 bool operator!=(const istream_iterator<T,charT,traits,Distance>&,
                 const istream_iterator<T,charT,traits,Distance>&);
```

Types

value_type;
Type of value to stream in.

char_type;
Type of character the stream is built on.

traits_type;
Traits used to build the stream.

istream_type;
Type of stream this iterator is constructed on.

Constructors

istream_iterator();
Construct an end-of-stream iterator. This iterator can be used to compare against an end-of-stream condition. Use it to provide end iterators to algorithms

istream_iterator(istream& s);
Construct an ***istream_iterator*** on the given stream.

istream_iterator(const istream_iterator& x);
Copy constructor.

Destructors

~istream_iterator();
Destructor.

Operators

const T&
operator*() const;
Return the current value stored by the iterator.

const T*
**operator->(**) const;
Return a pointer to the current value stored by the iterator.

istream_iterator& **operator**++()
istream_iterator **operator**++(int)
Retrieve the next element from the input stream.

Non-member Operators

```
bool
operator==(const istream_iterator<T,charT,traits,Distance>& x,
             const istream_iterator<T,charT,traits,Distance>& y)
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
bool
operator!=(const istream_iterator<T,charT,traits,Distance>& x,
             const istream_iterator<T,charT,traits,Distance>& y)
```

Inequality operator. Returns `true` if `x` is not the same as `y`.

Example

```
//
// io_iter.cpp
//
 #include <iterator>
 #include <vector>
 #include <numeric>
 #include <iostream.h>

 int main ()
 {
   vector<int> d;
   int total = 0;
   //
   // Collect values from cin until end of file
   // Note use of default constructor to get ending iterator
   //
   cout << "Enter a sequence of integers (eof to quit): " ;
   copy(istream_iterator<int,char>(cin),
        istream_iterator<int,char>(),
        inserter(d,d.begin()));
   //
   // stream the whole vector and the sum to cout
   //
   copy(d.begin(),d.end()-1,
        ostream_iterator<int,char>(cout," + "));
   if (d.size())
     cout << *(d.end()-1) << " = " <<
          accumulate(d.begin(),d.end(),total) << endl;
   return 0;
 }
```

Warning

If your compiler does not support default template parameters, then you will need to always supply the `Allocator` template argument. And you'll have to provide all parameters to the istream_iterator template. For instance, you'll have to write :

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

iterators, ***ostream_iterators***

iterator

Iterator base class

Summary

Base iterator class.

Synopsis

```
#include <iterator>

template <class Category, class T,  class Distance
RWSTD_SIMPLE_DEFAULT(ptrdiff_t)>
struct iterator
{
   typedef T value_type;
   typedef Distance distance_type;
   typedef Category iterator_category;
};
```

Description

The ***iterator*** structure provides a base class from which all other iterator types can be derived. This structure defines an interface that consists of three public types: `value_type`, `distance_type`, and `iterator_category`. These types are used primarily by classes derived from ***iterator*** and by the ***iterator_traits*** class.

See the ***iterators*** section in the Class Reference for a description of iterators and the capabilities associated with various types.

See Also

iterator_traits

iterator_traits

Iterator traits class

Summary

Provides basic information about an iterator.

Synopsis

```
template <class Iterator> struct iterator_traits
{
   typedef Iterator::value_type value_type;
   typedef Iterator::distance_type distance_type;
   typedef Iterator::iterator::category iterator_category;
};

// Specialization
template <class T> struct iterator_traits<T*>
{
   typedef T value_type;
   typedef Distance ptrdiff_t;
   typedef Category random_access_iterator_tag;
};
```

Description

The ***iterator_traits*** template and specialization provides a uniform way for algorithms to access information about a particular iterator. The template depends on an iterator providing a basic interface consisting of the types `value_type`, `distance_type`, and `iterator_category`, or on there being a specialization for the iterator. The library provides one specialization (partial) to handle all pointer iterator types.

iterator_traits are used within algorithms to provide local variables of the type pointed to by the iterator, or of the iterator's distance type. The traits are is also used to improve the efficiency of algorithms by making use of knowledge about basic iterator categories provided by the `iterator_category` member. An algorithm can use this "tag" to select the most efficient implementation an iterator is capable of handling without sacrificing the ability to work with a wide range of iterator types. For instance, both the `advance` and `distance` primitives use `iterator_category` to maximize their efficiency by using the tag to select from one of several different auxiliary functions. The `iterator_category` must therefore be one of the iterator tags provided by the library.

Tag Types

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

`iterator_traits::iterator_category` is typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,
        iterator_traits<Iterator>::iterator_category);
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           input_iterator_tag>
{
  // Most general implementation
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           bidirectional_iterator_tag>
{
  // Implementation takes advantage of bi-diretional
  // capability of the iterators
}

…etc.
```

See the ***iterator*** section in the Class Reference for a description of iterators and the capabilities associated with each type of iterator tag.

Warning

If your compiler does not support partial specialization then this template and specialization will not be available to you. Instead you will need to use the `distance_type`, `value_type`, and `iterator_category` families of function templates. The Rogue Wave *Standard C++ Library* also provides alternate implementations of the distance, advance, and count functions when partial specialization is not supported by a particular compiler.

See Also

value_type, distance_type, iterator_category, distance,advance, iterator

iterator_category

Iterator primitive

Summary

Determines the category that an iterator belongs to. This function is now obsolete. It is retained in order to provide backward compatibility and support compilers that do not provide partial specialization.

Synopsis

```
#include <iterator>

template <class T, class Distance>
inline input_iterator_tag
iterator_category (const input_iterator<T, Distance>&)

inline output_iterator_tag iterator_category (const output_iterator&)

template <class T, class Distance>
inline forward_iterator_tag
iterator_category (const forward_iterator<T, Distance>&)

template <class T, class Distance>
inline bidirectional_iterator_tag
iterator_category (const bidirectional_iterator<T, Distance>&)

template <class T, class Distance>
inline random_access_iterator_tag
iterator_category (const random_access_iterator<T, Distance>&)

template <class T>
inline random_access_iterator_tag iterator_category (const T*)
```

Description

The ***iterator_category*** family of function templates allows you to determine the category that any iterator belongs to. The first five functions take an iterator of a specific type and return the tag for that type. The last takes a `T*` and returns `random_access_iterator_tag`.

Tag Types

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

The ***iterator_category*** function is particularly useful for improving the efficiency of algorithms. An algorithm can use this function to select the most efficient implementation an iterator is capable of handling without sacrificing the ability to work with a wide range of iterator types. For instance, both the `advance` and `distance` primitives use ***iterator_category*** to maximize their efficiency by using the tag returned from

iterator_category to select from one of several different auxiliary functions. Because this is a compile time selection, use of this primitive incurs no significant runtime overhead.

iterator_category is typically used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,iterator_category(first));
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           input_iterator_tag>
{
  // Most general implementation
}

template <class Iterator>
void __foo(Iterator first, Iterator last,
           bidirectional_iterator_tag>
{
  // Implementation takes advantage of bi-diretional
  // capability of the iterators
}

…etc.
```

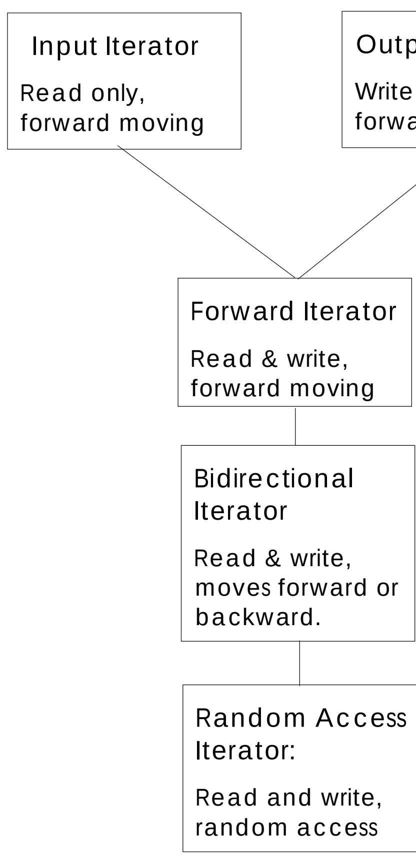
See the ***iterator*** section in the Class Reference for a description of iterators and the capabilities associated with each type of iterator tag.

See Also

Other iterator primitives: ***value_type, distance_type, distance,advance, iterator***

Summary Pointer generalizations for traversal and modification of collections.

Description Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. The illustration below displays the five iterator categories defined by the standard library, and shows their hierarchical relationship. Because standard library iterator categories are hierarchical, each category includes all the requirements of the categories above it.



Because iterators are used to traverse and access containers, the nature of the container determines what type of iterator it generates. And, because algorithms require specific iterator types as arguments, it is iterators that, for the most part, determine which standard library algorithms can be used with which standard library containers.

To conform to the C++ standard, all container and sequence classes must provide their own iterators. An instance of a container or sequence's iterator may be declared using either of the following:

```
class name ::iterator
class name ::const_iterator
```

Containers and sequences must also provide `const` iterators to the beginning and end of their collections. These may be accessed using the class members, `begin()` and `end()`.

The semantics of iterators are a generalization of the semantics of C++ pointers. Every template function that takes iterators will work using C++ pointers for processing typed contiguous memory sequences.

Iterators may be constant or mutable depending upon whether the result of the `operator*` behaves as a reference or as a reference to a constant. Constant iterators cannot satisfy the requirements of an `output_iterator`.

Every iterator type guarantees that there is an iterator value that points past the last element of a corresponding container. This value is called the *past-the-end value*. No guarantee is made that this value is dereferencable.

Every function provided by an iterator is required to be realized in amortized constant time.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Input Iterators

The following expressions must be valid for input iterators:

`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b`, `a != b`	
	return value convertible to `bool`

`*a`	`a == b` implies `*a == *b`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return `value` convertible to `const X&`
`*r++`	returns type `T`

For input iterators, `a == b` does not imply that `++a == ++b`.

Algorithms using input iterators should be single pass algorithms. That is they should not pass through the same iterator twice.

The value of type `T` does not have to be an `lvalue`.

Requirements for Output Iterators

The following expressions must be valid for output iterators:

`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`*a = t`	result is not used
`++r`	returns `X&`
`r++`	return value convertible to `const X&`
`*r++ = t`	result is not used

The only valid use for the `operator*` is on the left hand side of the assignment statement.

Algorithms using output iterators should be single pass algorithms. That is they should not pass through the same iterator twice.

Requirements for Forward Iterators

The following expressions must be valid for forward iterators:

`X u`	`u` might have a singular value
`X()`	`X()` might be singular
`X(a)`	copy constructor, `a == X(a)`
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`

`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`
`*r++`	returns `T&`

Forward iterators have the condition that `a == b` implies `*a== *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

Requirements for Bidirectional Iterators

A bidirectional iterator must meet all the requirements for forward iterators. In addition, the following expressions must be valid:

`--r`	returns `X&`
`r--`	return `value` convertible to `const X&`
`*r--`	returns `T&`

Requirements for Random Access Iterators

A random access iterator must meet all the requirements for bidirectional iterators. In addition, the following expressions must be valid:

`r += n`	Semantics of `--r` or `++r n` times depending on the sign of `n`
`a + n, n + a`	returns type `X`
`r -= n`	returns `X&`, behaves as `r += -n`
`a - n`	returns type `X`
`b - a`	returns `Distance`
`a[n]`	`*(a+n)`, return value convertible to `T`
`a < b`	total ordering relation
`a > b`	total ordering relation opposite to `<`
`a <= b`	`!(a > b)`
`a >= b`	`!(a < b)`

All relational operators return a value convertible to `bool`.

iter_swap

Algorithm

Summary Exchange values pointed at in two locations

Synopsis

```
#include <algorithm>

template <class ForwardIterator1, class ForwardIterator2>
void iter_swap (ForwardIterator1, ForwardIterator2);
```

Description

The ***iter_swap*** algorithm exchanges the values pointed at by the two iterators `a` and `b`.

Example

```
#include <vector>
#include <algorithm>
#include <iostream.h>

 int main ()
 {
   int d1[] = {6, 7, 8, 9, 10, 1, 2, 3, 4, 5};
   //
   // Set up a vector.
   //
   vector<int> v(d1+0, d1+10);
   //
   // Output original vector.
   //
   cout << "For the vector: ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   //
   // Swap the first five elements with the last five elements.
   //
   swap_ranges(v.begin(), v.begin()+5, v.begin()+5);
   //
   // Output result.
   //
   cout << endl << endl
        << "Swapping the first 5 elements with the last 5 gives: "
        << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   //
   // Now an example of iter_swap -- swap first and last elements.
   //
   iter_swap(v.begin(), v.end()-1);
   //
   // Output result.
   //
   cout << endl << endl
        << "Swapping the first and last elements gives: "
        << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl;
```

```
   return 0;
 }

Output :
For the vector: 6 7 8 9 10 1 2 3 4 5
Swapping the first five elements with the last five gives:
     1 2 3 4 5 6 7 8 9 10
Swapping the first and last elements gives:
     10 2 3 4 5 6 7 8 9 1
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

Iterators, swap, swap_ranges

Summary Binary function object that returns `true` if its first argument is less than its second

Synopsis

```
#include<functional>

template <class T>
struct less : public binary_function<T, T, bool> ;
```

Description ***less*** is a binary function object. Its `operator()` returns `true` if `x` is less than `y`. You can pass a ***less*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***less*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), less<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was less than `vec2(n)` or a "0" if `vec1(n)` was greater than or equal to `vec2(n)`.

Interface

```
template <class T>
struct less : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                                second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                                first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                                result_type;
  bool operator() (const T&, const T&) const;
};
```

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

`vector<int, allocator<int> >`

instead of

`vector<int>`

See Also ***binary_function, function objects***

Function Object

Summary Binary function object that returns `true` if its first argument is less than or equal to its second

Synopsis

```
#include<functional>

template <class T>
struct less_equal : public binary_function<T, T, bool>;
```

Description ***less_equal*** is a binary function object. Its `operator()` returns `true` if `x` is less than or equal to `y`. You can pass a ***less_equal*** object to any algorithm that requires a binary function. For example, the ***sort*** algorithm can accept a binary function as an alternate comparison object to sort a sequence. ***less_equal*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
.
.
.
sort(vec1.begin(), vec1.end(),less_equal<int>());
```

After this call to ***sort***, `vec1` will be sorted in ascending order.

Interface

```
template <class T>
struct less_equal : binary_function<T, T, bool> {
 typedef typename binary_function<T, T, bool>::second_argument_type
                                              second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                              first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                              result_type;
  bool operator() (const T&, const T&) const;
};
```

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

`vector<int, allocator<int> >`

instead of

`vector<int>`

See Also ***binary_function, function objects***

Summary Compares two ranges lexicographically.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2>
 bool
 lexicographical_compare(InputIterator1 first,
                         InputIterator2 last1,
                         InputIterator2 first2,
                         InputIterator last2);

template <class InputIterator1, class InputIterator2, class Compare>
 bool
 lexicographical_compare(InputIterator1 first,
                         InputIterator2 last1,
                         InputIterator2 first2,
                         InputIterator last2, Compare comp);
```

Description The ***lexicographical_compare*** functions compare each element in the range `[first1, last1)` to the corresponding element in the range `[first2, last2)` using iterators `i` and `j`.

The first version of the algorithm uses `operator<` as the default comparison operator. It immediately returns `true` if it encounters any pair in which `*i` is less than `*j`, and immediately returns `false` if `*j` is less than `*i`. If the algorithm reaches the end of the first sequence before reaching the end of the second sequence, it also returns `true`.

The second version of the function takes an argument `comp` that defines a comparison function that is used in place of the default `operator<`.

The ***lexicographical_compare*** functions can be used with all the datatypes provided by the standard library.

Complexity ***lexicographical_compare*** performs at most `min((last1 - first1), (last2 - first2))` applications of the comparison function.

Example

```
//
// lex_comp.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
```

```
{
  int d1[5] = {1,3,5,32,64};
  int d2[5] = {1,3,2,43,56};

  // set up vector
  vector<int> v1(d1,d1 + 5), v2(d2,d2 + 5);

  // Is v1 less than v2 (I think not)
  bool b1 =  lexicographical_compare(v1.begin(),
             v1.end(), v2.begin(), v2.end());

  // Is v2 less than v1 (yup, sure is)
  bool b2 =  lexicographical_compare(v2.begin(),
           v2.end(), v1.begin(), v1.end(), less<int>());
  cout << (b1 ? "TRUE" : "FALSE") << " "
       << (b2 ? "TRUE" : "FALSE") << endl;

  return 0;
}

Output:
 FALSE TRUE
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

limits

Numeric Limits library

Refer to the ***numeric_limits*** section of this reference guide.

Summary

A sequence that supports bidirectional iterators

Synopsis

```
#include <list>

template <class T, class Allocator = allocator<T> >
class list;
```

Description

list<T,Allocator> is a type of sequence that supports bidirectional iterators. A ***list<T,Allocator>*** allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Constant time random access is not supported.

Any type used for the template parameter `T` must provide the following (where `T` is the type, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
Default constructor   T()
Copy constructors     T(t) and T(u)
Destructor            t.~T()
Address of            &t and &u yielding T* and
                       const T* respectively
Assignment            t = a where a is a
                       (possibly const) value of T
```

Interface

```
template <class T, class Allocator = allocator<T> >
 class list {

public:

// typedefs

   class iterator;
   class const_iterator;
   typename reference;
   typename const_reference;
   typename size_type;
   typename difference_type;
   typedef T value_type;
   typedef Allocator allocator_type;

   typename reverse_iterator;
   typename const_reverse_iterator;

// Construct/Copy/Destroy

   explicit list (const Allocator& = Allocator());
   explicit list (size_type, const Allocator& = Allocator());
```

```
   list (size_type, const T&, const Allocator& = Allocator())
   template <class InputIterator>
   list (InputIterator, InputIterator,
         const Allocator& = Allocator());
   list(const list<T, Allocator>& x);
   ~list();
   list<T,Allocator>& operator= (const list<T,Allocator>&);
   template <class InputIterator>
    void assign (InputIterator, InputIterator);
   template <class Size, class T>
    void assign (Size n);
   template <class Size, class T>
    void assign (Size n, const T&);

   allocator_type get allocator () const;

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   bool empty () const;
   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize  (size_type, T);

// Element Access

   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

// Modifiers

   void push_front (const T&);
   void pop_front ();
   void push_back (const T&);
   void pop_back ();

   iterator insert (iterator);
   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
   template <class InputIterator>
    void insert  (iterator, InputIterator, InputIterator);

   iterator erase (iterator);
   iterator erase (iterator, iterator);
```

```
   void swap (list<T, Allocator>&);
   void clear ();

// Special mutative operations on list

   void splice (iterator, list<T, Allocator>&);
   void splice (iterator, list<T, Allocator>&, iterator);
   void splice (iterator, list<T, Allocator>&, iterator,
iterator);

   void remove (const T&);
   template <class Predicate>
    void remove_if (Predicate);

   void unique ();
   template <class BinaryPredicate>
    void unique (BinaryPredicate);

   void merge (list<T, Allocator>&);
   template <class Compare>
    void merge (list<T, Allocator>&, Compare);

   void sort ();
   template <class Compare>
    void sort (Compare);

   void reverse();
};

// Non-member List Operators

template <class T, class Allocator>
 bool operator== (const list<T, Allocator>&,
                  const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator!= (const list<T, Allocator>&,
                  const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator< (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator> (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator<= (const list<T, Allocator>&,
                 const list<T, Allocator>&);

template <class T, class Allocator>
 bool operator>= (const list<T, Allocator>&,
                 const list<T, Allocator>&);


// Specialized Algorithms
```

```
template <class T, class Allocator>
void swap (list<T,Allocator>&, list<T, Allocator>&);
```

Constructors and Destructors

`explicit list(const Allocator& alloc = Allocator());`

Creates a list of zero elements. The list will use the allocator `alloc` for all storage management.

```
explicit list(size_type n,
              const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of the default value for type `T`. Requires that `T` have a default constructor. The list will use the allocator `alloc` for all storage management.

```
list(size_type n, const T& value,
     const Allocator& alloc = Allocator());
```

Creates a list of length `n`, containing `n` copies of `value`. The list will use the allocator `alloc` for all storage management.

```
template <class InputIterator>
list(InputIterator first, InputIterator last,
     const Allocator& alloc = Allocator());
```

Creates a list of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The list will use the allocator `alloc` for all storage management.

`list(const list<T, Allocator>& x);`

Copy constructor. Creates a copy of `x`.

`~list();`

The destructor. Releases any allocated memory for this list.

Assignment Operator

```
list<T, Allocator>&
operator=(const list<T, Allocator>& x)
```

Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to `*this`.

Allocator

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin();
```

Returns a bidirectional iterator that points to the first element.

```
const_iterator
begin() const;
```

Returns a constant bidirectional iterator that points to the first element.

```
iterator
end();
```
Returns a bidirectional iterator that points to the past-the-end value.

```
const_iterator
end() const;
```
Returns a constant bidirectional iterator that points to the past-the-end value.

```
reverse_iterator
rbegin();
```
Returns a bidirectional iterator that points to the past-the-end value.

```
const_reverse_iterator
rbegin() const;
```
Returns a constant bidirectional iterator that points to the past-the-end value.

```
reverse_iterator
rend();
```
Returns a bidirectional iterator that points to the first element.

```
const_reverse_iterator
rend() const;
```
Returns a constant bidirectional iterator that points to the first element.

Member Functions

```
template <class InputIterator>
void
assign(InputIterator first, InputIterator last);
```
Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

```
template <class Size, class T>
void
assign(Size n);
```
Erases all elements contained in self, then inserts `n` instances of the default `value` of `t.`

```
template <class Size, class T>
void
assign(Size n, const T& t);
```
Erases all elements contained in self, then inserts `n` instances of the `value` of `t.`

```
reference
back();
```
Returns a `reference` to the last element.

```
const_reference
back() const;
```
Returns a constant reference to the last element.

```
void
clear();
```

Erases all elements from the list.

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

Returns `true` if the `size` is zero.

```
iterator
erase(iterator position);
```

Removes the element pointed to by `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
iterator
erase(iterator first, iterator last);
```

Removes the elements in the range (`first, last`). Returns an iterator pointing to the element following the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
reference
front();
```

Returns a reference to the first element.

```
const_reference
front() const;
```

Returns a constant reference to the first element.

```
iterator
insert(iterator position);
```

Inserts a copy of the default value for type `T` before `position`. Returns an iterator that points to the inserted value. Requires that type `T` have a default constructor.

```
iterator
insert(iterator position, const T& x);
```

Inserts `x` before `position`. Returns an iterator that points to the inserted `x`.

```
void
insert(iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` before `position`.

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

Inserts copies of the elements in the range `[first, last)` before `position`.

```
size_type
max_size() const;
```
Returns `size()` of the largest possible list.

```
void merge(list<T, Allocator>& x);
```
Merges a sorted `x` with a sorted self using `operator<`. For equal elements in the two lists, elements from self will always precede the elements from `x`. The `merge` function leaves `x` empty.

```
template <class Compare>
void
merge(list<T, Allocator>& x, Compare comp);
```
Merges a sorted `x` with sorted self using a compare function object, `comp`. For same elements in the two lists, elements from self will always precede the elements from `x`. The `merge` function leaves `x` empty.

```
void
pop_back();
```
Removes the last element.

```
void
pop_front();
```
Removes the first element.

```
void
push_back(const T& x);
```
Appends a copy of `x` to the end of the list.

```
void
push_front(const T& x);
```
Appends a copy of `x` to the front of the list.

```
void
remove(const T& value);
template <class Predicate>
void
remove_if(Predicate pred);
```
Removes all elements in the list referred by the list iterator `i` for which `*i == value` or `pred(*i) == true`, whichever is applicable. This is a stable operation, the relative order of list items that are not removed is preserved.

```
void
resize(size_type sz);
```
Alters the size of self. If the new size (`sz`) is greater than the current size, `sz-size()` copies of the default value of type `T` are inserted at the end of the list. If the new size is smaller than the current capacity, then the list is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken. Requires that type `T` have a default constructor.

```
void
resize(size_type sz, T c);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, `sz`-`size()` `c`'s are inserted at the end of the list. If the new size is smaller than the current capacity, then the list is truncated by erasing `size()-sz` elements off the end. Otherwise, no action is taken.

```
void
reverse();
```

Reverses the order of the elements.

```
size_type
size() const;
```

Returns the number of elements.

```
void
sort();
```

Sorts self according to the `operator<`. `sort` maintains the relative order of equal elements.

```
template <class Compare>
void
sort(Compare comp);
```

Sorts self according to a comparison function object, `comp`. This is also a stable sort.

```
void
splice(iterator position, list<T, Allocator>& x);
```

Inserts `x` before `position` leaving `x` empty.

```
void
splice(iterator position, list<T, Allocator>&  x, iterator i);
```

Moves the elements pointed to by iterator `i` in `x` to self, inserting it before `position`. The element is removed from `x`.

```
void
splice(iterator position, list<T, Allocator >&  x,
        iterator first, iterator last);
```

Moves the elements in the range `[first, last)` in `x` to self, inserting before `position`. The elements in the range `[first, last)` are removed from `x`.

```
void
swap(list <T, Allocator>& x);
```

Exchanges self with `x`.

```
void
unique();
```

Erases copies of consecutive repeated elements leaving the first occurrence.

```
template <class BinaryPredicate>
void
unique(BinaryPredicate binary_pred);
```

Erases consecutive elements matching a true condition of the `binary_pred`. The first occurrence is not removed.

Non-member Operators

```
template <class T, class Allocator>
bool operator==(const list<T, Allocator>& x,
                 const list<T, Allocator>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool operator!=(const list<T, Allocator>& x,
                 const list<T, Allocator>& y);
```

Inequality operator. Returns `!(x==y)`.

```
template <class T, class Allocator>
bool operator<(const list<T, Allocator>& x,
                const list<T,Allocator>& y);
```

Returns `true` if the sequence defined by the elements contained in `x` is lexicographically less than the sequence defined by the elements contained in `y`.

```
template <class T, class Allocator>
bool operator>(const list<T, Allocator>& x,
                const list<T,Allocator>& y);
```

Returns `y < x`.

```
template <class T, class Allocator>
bool operator<=(const list<T, Allocator>& x,
                const list<T,Allocator>& y);
```

Returns `!(y < x)`.

```
template <class T, class Allocator>
bool operator>=(const list<T, Allocator>& x,
                const list<T,Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class T, class Allocator>
void swap(list<T, Allocator>& a, list<T, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// list.cpp
//
 #include <list>
 #include <string>
 #include <iostream.h>

 // Print out a list of strings
 ostream& operator<<(ostream& out, const list<string>& l)
 {
```

```
  copy(l.begin(), l.end(),
       ostream_iterator<string,char>(cout," "));
  return out;
}
int main(void)
{
  // create a list of critters
  list<string> critters;
  int i;
  // insert several critters
  critters.insert(critters.begin(),"antelope");
  critters.insert(critters.begin(),"bear");
  critters.insert(critters.begin(),"cat");

  // print out the list
  cout << critters << endl;

  // Change cat to cougar
  *find(critters.begin(),critters.end(),"cat") = "cougar";
  cout << critters << endl;

  // put a zebra at the beginning
  // an ocelot ahead of antelope
  // and a rat at the end
  critters.push_front("zebra");
  critters.insert(find(critters.begin(),critters.end(),
                  "antelope"),"ocelot");
  critters.push_back("rat");
  cout << critters << endl;

  // sort the list (Use list's sort function since the
  // generic algorithm requires a random access iterator
  // and list only provides bidirectional)
  critters.sort();
  cout << critters << endl;

  // now let's erase half of the critters
  int half = critters.size() >> 1;
  for(i = 0; i < half; ++i) {
    critters.erase(critters.begin());
  }
  cout << critters << endl;
  return 0;
}

Output :
cat bear antelope
cougar bear antelope
zebra cougar bear ocelot antelope rat
antelope bear cougar ocelot rat zebra
ocelot  rat zebra
```

Warnings

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

```
template <class InputIterator>
list (InputIterator, InputIterator,
      const Allocator& = Allocator());
```

list also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a list in the following two ways:

```
int intarray[10];
list<int> first_list(intarray,intarray + 10);
list<int> second_list(first_list.begin(),first_list.end());
```

But not this way:

```
list<long> long_list(first_list.begin(),first_list.end());
```

since the `long_list` and `first_list` are not the same type.

Additionally, ***list*** provides a `merge` function of this type.

```
template <class Compare> void merge (list<T, Allocator>&,
  Compare);
```

This function allows you to specify a compare function object to be used in merging two lists. In this case, we were unable to provide a substitute function in addition to the merge that uses the `operator<` as the default. Thus, if your compiler does not support member function templates, all list mergers will use `operator<`.

Also, many compilers do not support default template arguments. If your compiler is one of these, you need to always supply the `Allocator` template argument. For instance, you'll have to write:

```
list<int, allocator<int> >
```

instead of:

```
list<int>
```

See Also

allocator, Containers, Iterators

logical_and

Function Object

Summary Binary function object that returns `true` if both of its arguments are `true`.

Synopsis

```
#include <functional>

template <class T>
struct logical_and : public binary_function<T, T, bool>;
```

Description ***logical_and*** is a binary function object. Its `operator()` returns `true` if both `x` and `y` are `true`. You can pass a ***logical_and*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***logical_and*** is used in that algorithm in the following manner:

```
vector<bool> vec1;
vector<bool> vec2;
vector<bool> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), logical_and<bool>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" (`true`) if both `vec1(n)` and `vec2(n)` are `true` or a "0" (`false`) if either `vec1(n)` or `vec2(n)` is `false`.

Interface

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                                 second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                                 first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                                 result_type;
  bool operator() (const T&, const T&) const;
};
```

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

`vector<bool, allocator<bool> >`

instead of:

```
vector<bool>
```

See Also ***binary_function, function objects***

logical_not

Function Object

Summary Unary function object that returns `true` if its argument is `false`.

Synopsis

```
#include <functional>

template <class T>
struct logical_not : unary_function<T, bool> ;
```

Description ***logical_not*** is a unary function object. Its `operator()` returns `true` if its argument is `false`. You can pass a ***logical_not*** object to any algorithm that requires a unary function. For example, the ***replace_if*** algorithm replaces an element with another value if the result of a unary operation is true. ***logical_not*** is used in that algorithm in the following manner:

```
  vector<int> vec1;
  .
  .
  .
  void replace_if(vec1.begin(), vec1.end(),
                  logical_not<int>(),1);
```

This call to ***replace_if*** replaces all zeros in the `vec1` with "1".

Interface

```
template <class T>
struct logical_not : unary_function<T, bool> {
  typedef typename unary_function<T, bool>::argument_type
                                            argument_type;
  typedef typename unary_function<T, bool>::result_type result_type;
  bool operator() (const T&) const;
};
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***function objects, unary_function***

logical_or

Function Object

Summary

Binary function object that returns true if either of its arguments are true.

Synopsis

```
#include <functional>

template <class T>
struct logical_or : binary_function<T, T, bool> ;
```

Description

logical_or is a binary function object. Its operator() returns true if either x or y are true. You can pass a ***logical_or*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result of the function. ***logical_or*** is used in that algorithm in the following manner:

```
  vector<bool> vec1;
  vector<bool> vec2;
  vector<bool> vecResult;
  .
  .
  .
  transform(vec1.begin(), vec1.end(),
            vec2.begin(),
            vecResult.begin(), logical_or<bool>());
```

After this call to ***transform***, vecResult(n) will contain a "1" (true) if either vec1(n) or vec2(n) is true or a "0" (false) if both vec1(n) and vec2(n) are false.

Interface

```
template <class T>
struct logical_or : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                               second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                               first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                               result_type;
  bool operator() (const T&, const T&) const;
};
```

Warning

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

```
vector<bool, allocator<bool> >
```

instead of:

```
vector<bool>
```

See Also ***binary_function, function objects***

Summary

Determine the first valid position for an element in a sorted container.

Synopsis

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

 template <class ForwardIterator, class T, class Compare>
  ForwardIterator lower_bound(ForwardIterator first,
                              ForwardIterator last,
                              const T& value, Compare comp);
```

Description

The ***lower_bound*** algorithm compares a supplied `value` to elements in a sorted container and returns the first position in the container that `value` can occupy without violating the container's ordering. There are two versions of the algorithm. The first uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version lets you include a function object of type `Compare`, and assumes that `Compare` is the function used to sort the sequence. The function object must be a binary predicate.

lower_bound's return value is the iterator for the first element in the container that is *greater than or equal to* `value`, or, when the comparison operator is used, the first element that does not satisfy the comparison function. Formally, the algorithm returns an iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold:

```
*j  <  value
```

or

```
comp(*j,  value) == true
```

Complexity

lower_bound performs at most `log(last - first) + 1` comparisons.

Example

```
//
// ul_bound.cpp
//
 #include <vector>
 #include <algorithm>
```

```
#include <iostream.h>

int main()
{
  typedef vector<int>::iterator iterator;
  int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};

  // Set up a vector
  vector<int> v1(d1,d1 + 11);

  // Try lower_bound variants
  iterator it1 = lower_bound(v1.begin(),v1.end(),3);
  // it1 = v1.begin() + 4

  iterator it2 =
      lower_bound(v1.begin(),v1.end(),2,less<int>());
  // it2 = v1.begin() + 4

  // Try upper_bound variants
  iterator it3 = upper_bound(v1.begin(),v1.end(),3);
  // it3 = vector + 5

  iterator it4 =
     upper_bound(v1.begin(),v1.end(),2,less<int>());
  // it4 = v1.begin() + 5

  cout << endl << endl
       << "The upper and lower bounds of 3: ( "
       << *it1 << " , " << *it3 << " ]" << endl;

  cout << endl << endl
       << "The upper and lower bounds of 2: ( "
       << *it2 << " , " << *it4 << " ]" << endl;

  return 0;
}
```

```
Output :
The upper and lower bounds of 3: ( 3 , 4 ]
The upper and lower bounds of 2: ( 2 , 3 ]
```

Warning

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

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

instead of:

```
vector<int>
```

See Also

upper_bound, equal_range

Summary

Creates a heap.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
  void
  make_heap(RandomAccessIterator first,
            RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  make_heap(RandomAccessIterator first,
            RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** algorithm, or a new element can be added by the ***push_heap*** algorithm, in `O(logN)` time.

These properties make heaps useful as priority queues.

The heap algorithms use less than (`operator<`) as the default comparison. In all of the algorithms, an alternate comparison operator can be specified.

The first version of the ***make_heap*** algorithm arranges the elements in the range `[first, last)` into a heap using less than (`operator<`) to perform comparisons. The second version uses the comparison operator `comp` to perform the comparisons. Since the only requirements for a heap are the two listed above, `make_heap` is not required to do anything within the range `(first, last - 1)`.

Complexity

This algorithm makes at most `3 * (last - first)` comparisons.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
```

```
{
  int d1[4] = {1,2,3,4};
  int d2[4] = {1,3,2,4};

  // Set up two vectors
  vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

  // Make heaps
  make_heap(v1.begin(),v1.end());
  make_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
  // Note that x, y and z represent the remaining
  // values in the container (other than 4).
  // The definition of the heap and heap operations
  // does not require any particular ordering
  // of these values.

  // Copy both vectors to cout
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // Now let's pop
  pop_heap(v1.begin(),v1.end());
  pop_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (3,x,y,4) and v2 = (3,x,y,4)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // And push
  push_heap(v1.begin(),v1.end());
  push_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (4,x,y,z) and v2 = (4,x,y,z)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // Now sort those heaps
  sort_heap(v1.begin(),v1.end());
  sort_heap(v2.begin(),v2.end(),less<int>());
  // v1 = v2 = (1,2,3,4)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  return 0;
```

```
 }
Output :
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

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

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

instead of:

```
vector<int>
```

See Also ***pop_heap***, ***push_heap*** and ***sort_heap***

Summary

An associative container providing access to non-key values using unique keys. A ***map*** supports bidirectional iterators.

Synopsis

```
#include <map>

template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<T> >
class map;
```

Description

map <Key, T, Compare, Allocator> provides fast access to stored values of type `T` which are indexed by unique keys of type ***Key***. The default operation for key comparison is the `<` operator.

map provides bidirectional iterators that point to an instance of `pair<const Key x, T y>` where `x` is the key and `y` is the stored value associated with that key. The definition of ***map*** provides a `typedef` to this pair called `value_type`.

The types used for both the template parameters `Key` and `T` must provide the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
Copy constructors -  T(t) and T(u)
Destructor        -  t.~T()
Address of        -  &t and &u yielding T* and
                     const T* respectively
Assignment        -  t = a where a is a
                     (possibly const) value of T
```

The type used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class T, class Compare = less<Key>
          class Allocator = allocator<T> >
 class map {

public:

// types

   typedef Key key_type;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;
   typename reference;
   typename const_reference;
   typename iterator;
```

```
   typename const_iterator;
   typename size_type;
   typename difference_type;
   typename reverse_iterator;
   typename const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type, bool>
   {
     friend class map<Key, T, Compare, Allocator>;

     public :
       bool operator() (const value_type&,
                        const value_type&) const;
   };

// Construct/Copy/Destroy

   explicit map (const Compare& = Compare(),
                 const Allocator& = Allocator ());
   template <class InputIterator>
    map (InputIterator, InputIterator,
         const Compare& = Compare(),
         const Allocator& = Allocator ());
   map (const map<Key, T, Compare, Allocator>&);
   ~map();
   map<Key, T, Compare, Allocator>&
    operator= (const map<Key, T, Compare, Allocator>&);
   allocator_type get_allocator () const;

// Iterators

   iterator begin();
   const_iterator begin() const;
   iterator end();
   const_iterator end() const;
   reverse_iterator rbegin();
   const_reverse_iterator rbegin() const;
   reverse_iterator rend();
   const_reverse_iterator rend() const;

// Capacity

   bool empty() const;
   size_type size() const;
   size_type max_size() const;

// Element Access

   mapped_type& operator[] (const key_type&);
   const mapped_type& operator[] (const key_type&) const;

// Modifiers

   pair<iterator, bool> insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);
```

```
   iterator erase (iterator);
   size_type erase (const key_type&);
   iterator erase (iterator, iterator);
   void swap (map<Key, T, Compare, Allocator>&);

// Observers

   key_compare key_comp() const;
   value_compare value_comp() const;

// Map operations

   iterator find (const key_value&);
   const_iterator find (const key_value&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const key_type&);
   pair<const_iterator, const_iterator>
     equal_range (const key_type&) const;
};

// Non-member Map Operators

template <class Key, class T, class Compare, class Allocator>
 bool operator== (const map<Key, T, Compare, Allocator>&,
                  const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator!= (const map<Key, T, Compare, Allocator>&,
                  const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator< (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator> (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator<= (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator>= (const map<Key, T, Compare, Allocator>&,
                 const map<Key, T, Compare, Allocator>&);


// Specialized Algorithms

template <class Key, class T, class Compare, class Allocator>
 void swap (map<*Key,T,Compare,Allocator>&,
            map<Key,T,Compare,Allocator>&);
```

Constructors and Destructors

```
explicit map(const Compare& comp = Compare(),
             const Allocator& alloc = Allocator());
```

Default constructor. Constructs an empty map that will use the relation `comp` to order keys, if it is supplied. The map will use the allocator `alloc` for all storage management.

```
template <class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& alloc = Allocator());
```

Constructs a map containing values in the range `[first, last)`. Creation of the new map is only guaranteed to succeed if the iterators `first` and `last` return values of type `pair<class Key, class Value>` and all values of `Key` in the `range[first, last)` are unique. The map will use the relation `comp` to order keys, and the allocator `alloc` for all storage management.

```
map(const map<Key,T,Compare,Allocator>& x);
```

Copy constructor. Creates a new map by copying all pairs of `key` and `value` from `x`.

```
~map();
```

The destructor. Releases any allocated memory for this map.

Allocator

```
allocator_type get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin() ;
```

Returns an `iterator` pointing to the first element stored in the map. "First" is defined by the map's comparison operator, `Compare`.

```
const_iterator
begin() const;
```

Returns a `const_iterator` pointing to the first element stored in the map.

```
iterator
end() ;
```

Returns an `iterator` pointing to the last element stored in the map, i.e., the off-the-end value.

```
const_iterator
end() const;
```

Returns a `const_iterator` pointing to the last element stored in the map.

```
reverse_iterator
rbegin();
```

Returns a `reverse_iterator` pointing to the first element stored in the map. "First" is defined by the map's comparison operator, `Compare`.

```
const_reverse_iterator
rbegin() const;
```

Returns a `const_reverse_iterator` pointing to the first element stored in the map.

```
reverse_iterator
rend() ;
```

Returns a `reverse_iterator` pointing to the last element stored in the map, i.e., the off-the-end value.

```
const_reverse_iterator
rend() const;
```

Returns a `const_reverse_iterator` pointing to the last element stored in the map.

Member Operators

```
map<Key, T, Compare, Allocator>&
operator=(const map<Key, T, Compare, Allocator>& x);
```

Assignment. Replaces the contents of `*this` with a copy of the map `x`.

```
mapped_type&
operator[](const key_type& x);
```

If an element with the key `x` exists in the map, then a reference to its associated value will be returned. Otherwise the pair `x,T()` will be inserted into the map and a reference to the default object `T()` will be returned.

Allocator

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Member Functions

```
void
clear();
```

Erases all elements from the self.

```
size_type
count(const key_type& x) const;
```

Returns a 1 if a value with the key `x` exists in the map, otherwise returns a 0.

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

Returns `true` if the map is empty, `false` otherwise.

```
pair<iterator, iterator>
equal_range (const  key_type& x);
```

Returns the pair, `(lower_bound(x), upper_bound(x))`.

```
pair<const_iterator,const_iterator>
equal_range(const key_type& x) const;
```

Returns the pair, `(lower_bound(x), upper_bound(x))`.

```
iterator
erase(iterator position);
```

Deletes the map element pointed to by the iterator `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
iterator
erase(iterator first, iterator last);
```

Providing the iterators `first` and `last` point to the same map and last is reachable from first, all elements in the range (`first, last`) will be deleted from the map. Returns an iterator pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
size_type
erase(const key_type& x);
```

Deletes the element with the key value `x` from the map, if one exists. Returns 1 if `x` existed in the map, 0 otherwise.

```
iterator
find(const key_type& x);
```

Searches the map for a pair with the key value `x` and returns an `iterator` to that pair if it is found. If such a pair is not found the value `end()` is returned.

```
const_iterator find(const key_type& x) const;
```

Same as `find` above but returns a `const_iterator`.

```
pair<iterator, bool>
insert(const value_type& x);
```

```
iterator
insert(iterator position, const value_type& x);
```

If a `value_type` with the same key as `x` is not present in the map, then `x` is inserted into the map. Otherwise, the pair is not inserted. A position may be supplied as a hint regarding where to do the insertion. If the insertion may be done right after `position` then it takes amortized constant time. Otherwise it will take `O(log N)` time.

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

Copies of each element in the range `[first, last)` which possess a unique key, one not already in the map, will be inserted into the map. The iterators `first` and `last` must return values of `type pair<T1,T2>`. This operation takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current map.

```
iterator
lower_bound(const key_type& x);
```

Returns a reference to the first entry with a key greater than or equal to `x`.

```
const_iterator
lower_bound(const key_type& x) const;
```

Same as `lower_bound` above but returns a `const_iterator`.

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

Returns the maximum possible size of the map. This size is only constrained by the number of unique keys which can be represented by the type `Key`.

```
size_type
size() const;
```

Returns the number of elements in the map.

```
void
swap(map<Key, T, Compare, Allocator>& x);
```

Swaps the contents of the map `x` with the current map, `*this`.

```
iterator
upper_bound(const key_type& x);
```

Returns a reference to the first entry with a key less than or equal to `x`.

```
const_iterator
upper_bound(const key_type& x) const;
```

Same as `upper_bound` above but returns a `const_iterator.`

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing `pair<const Key, T>` values using the comparison operation, `Compare`, of the current map. This function is identical to `key_comp` for sets.

Non-member Operators

```
template <class Key, class T, class Compare, class Allocator>
bool operator==(const map<Key, T, Compare, Allocator>& x,
                 const map<Key, T, Compare, Allocator>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==)`. Otherwise it returns `false`.

```
template <class Key, class T, class Compare, class Allocator>
bool operator!=(const map<Key, T, Compare, Allocator>& x,
                 const map<Key, T, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

```
template <class Key, class T, class Compare, class Allocator>
bool operator>(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns `y < x`.

```
template <class Key, class T, class Compare, class Allocator>
bool operator<=(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class Key, class T, class Compare, class Allocator>
bool operator>=(const map<Key, T, Compare, Allocator>& x,
                const map<Key, T, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key, T, Compare, Allocator>& a,
          map<Key, T, Compare, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// map.cpp
//
 #include <string>
 #include <map>
 #include <iostream.h>

 typedef map<string, int, less<string> > months_type;

 // Print out a pair
 template <class First, class Second>
 ostream& operator<<(ostream& out,
                     const pair<First,Second> & p)
 {
   cout << p.first << " has " << p.second << " days";
   return out;
 }

 // Print out a map
 ostream& operator<<(ostream& out, const months_type & l)
 {
   copy(l.begin(),l.end(), ostream_iterator
                 <months_type::value_type,char>(cout,"\n"));
   return out;
 }
```

```
int main(void)
{
  // create a map of months and the number of days
  // in the month
  months_type months;

  typedef months_type::value_type value_type;

  // Put the months in the multimap
  months.insert(value_type(string("January"),   31));
  months.insert(value_type(string("February"),   28));
  months.insert(value_type(string("February"),   29));
  months.insert(value_type(string("March"),     31));
  months.insert(value_type(string("April"),     30));
  months.insert(value_type(string("May"),       31));
  months.insert(value_type(string("June"),      30));
  months.insert(value_type(string("July"),      31));
  months.insert(value_type(string("August"),    31));
  months.insert(value_type(string("September"), 30));
  months.insert(value_type(string("October"),   31));
  months.insert(value_type(string("November"),  30));
  months.insert(value_type(string("December"),  31));

  // print out the months
  // Second February is not present
  cout << months << endl;

  // Find the Number of days in June
  months_type::iterator p = months.find(string("June"));

  // print out the number of days in June
  if (p != months.end())
    cout << endl << *p << endl;

  return 0;
}

Output :
April has 30 days
August has 31 days
December has 31 days
February has 28 days
January has 31 days
July has 31 days
June has 30 days
March has 31 days
May has 31 days
November has 30 days
October has 31 days
September has 30 days
```

Warning Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for `map<Key,T,Compare,Allocator>` that takes two templated iterators:

```
template <class InputIterator>
 map (InputIterator, InputIterator, const Compare& = Compare(),
```

```
        const Allocator& = Allocator());
```

map also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a ***map*** in the following two ways:

```
map<int, int, less<int> >::value_type intarray[10];
map<int, int, less<int> > first_map(intarray, intarray + 10);
map<int, int, less<int> > second_map(first_map.begin(),
                                     first_map.end());
```

But not this way:

```
map<long, long, less<long> > long_map(first_map.begin(),
                                      first_map.end());
```

Since the `long_map` and `first_map` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these, you need to always supply the `Compare` template argument and the `Allocator` template argument. For instance, you'll have to write:

```
map<int, int, less<int>,  allocator<int> >
```

instead of:

```
map<int, int>
```

See Also ***allocator, Containers, Iterators, multimap***

Summary Find and return the maximum of a pair of values

Synopsis

```
#include <algorithm>

template <class T>
 const T& max(const T&, const T&);

template <class T, class Compare>
 const T& max(const T&, const T&, Compare);
```

Description The ***max*** algorithm determines and returns the maximum of a pair of values. The optional argument `Compare` defines a comparison function that can be used in place of the default `operator<`. This function can be used with all the datatypes provided by the standard library.

max returns the first argument when the arguments are equal.

Example

```
//
// max.cpp
//
 #include <algorithm>
 #include <iostream.h>
 #include <iostream.h>

 int main(void)
 {
   double  d1 = 10.0, d2 = 20.0;

   // Find minimum
   double val1 = min(d1, d2);
   // val1 = 10.0

   // the greater comparator returns the greater of the
   // two values.
   double val2 = min(d1, d2, greater<double>());
   // val2 = 20.0;

   // Find maximum
   double val3 = max(d1, d2);
   // val3 = 20.0;

   // the less comparator returns the smaller of the two values.
   // Note that, like every comparison in the STL, max is
   // defined in terms of the < operator, so using less here
   // is the same as using the max algorithm with a default
   // comparator.
   double val4 = max(d1, d2, less<double>());
   // val4 = 20
```

```
   cout << val1 << " " << val2 << " "
        << val3 << " " << val4 << endl;

   return 0;
 }

Output :
10 20 20 20
```

See Also ***max_element***, ***min***, ***min_element***

Summary Finds maximum value in a range.

Synopsis

```
#include <algorithm>

template <class ForwardIterator>
 ForwardIterator
 max_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
 ForwardIterator
 max_element(ForwardIterator first, ForwardIterator last,
             Compare comp);
```

Description The ***max_element*** algorithm returns an iterator that denotes the maximum element in a sequence. If the sequence contains more than one copy of the element, the iterator points to its first occurrence. The optional argument `comp` defines a comparison function that can be used in place of the default `operator<`. This function can be used with all the datatypes provided by the standard library.

Algorithm ***max_element*** returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the same range the following corresponding conditions hold:

```
!(*i < *j)
```

or

```
comp(*i, *j) == false.
```

Complexity Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done for ***max_element***.

Example

```
//
// max_elem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
 {
   typedef vector<int>::iterator iterator;
```

```
  int d1[5] = {1,3,5,32,64};

  // set up vector
  vector<int>      v1(d1,d1 + 5);

  // find the largest element in the vector
  iterator it1 = max_element(v1.begin(), v1.end());
  // it1 = v1.begin() + 4

  // find the largest element in the range from
  // the beginning of the vector to the 2nd to last
  iterator it2 = max_element(v1.begin(), v1.end()-1,
                   less<int>());
  // it2 = v1.begin() + 3

  // find the smallest element
  iterator it3 = min_element(v1.begin(), v1.end());
  // it3 = v1.begin()

  // find the smallest value in the range from
  // the beginning of the vector plus 1 to the end
  iterator it4 = min_element(v1.begin()+1, v1.end(),
                   less<int>());
  // it4 = v1.begin() + 1

  cout << *it1 << " " << *it2 << " "
       << *it3 << " " << *it4 << endl;

  return 0;
}

Output :
64 32 1 3
```

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

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

instead of:

```
vector<int>
```

See Also ***max***, ***min***, ***min_element***

mem_fun, mem_fun1, mem_fun_ref, mem_fun_ref1

Function Adaptors

Summary

Function objects that adapt a pointer to a member function to work where a global function is called for.

Synopsis

```
#include <functional>

template <class S, class T> class mem_fun_t;
template <class S, class T, class A> class mem_fun1_t;
template <class S, class T> class mem_fun_ref_t;
template <class S, class T, class A> class mem_fun1_ref_t;

template<class S, class T> mem_fun_t<S,T>
   mem_fun(S, (T::*f)());
template<class S, class T, class A> mem_fun1_t<S,T,A>
    mem_fun1(S, (T::*f)(A));
template<class S, class T> mem_fun_ref_t<S,T>
     mem_fun_ref(S, (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
     mem_fun1_ref(S, (T::*f)(A));
```

Description

The ***mem_fun*** group of templates each encapsulates a pointer to a member function. Each category of template (i.e. ***mem_fun***, ***mem_fun1***, ***mem_fun_ref***, or ***mem_fun1_ref***) provides both a class template and a function template, where the class is distinguished by the addition of `_t` on the end of the name to identify it as a type.

The class's constructor takes a pointer to a member function, and provides an `operator()` that forwards the call to that member function. In this way the resulting object serves as a global function object for that member function.

The accompanying function template simplifies the use of this facility by constructing an instance of the class on the fly.

The library provides zero and one argument adaptors for containers of pointers and containers of references (`_ref`). This technique can be easily extended to include adaptors for two argument functions, and so on.

Interface

```
template <class S, class T> class mem_fun_t
          : public unary_function<T*, S> {
    public:
      explicit mem_fun_t(S (T::*p)());
      S operator()(T* p);
};

template <class S, class T, class A> class mem_fun1_t
          : public binary_function<T*, A, S> {
    public:
      explicit mem_fun1_t(S (T::*p)(A));
      S operator()(T* p, A x);
};

template<class S, class T> mem_fun_t<S,T>
   mem_fun(S, (T::*f)());

template<class S, class T, class A> mem_fun1_t<S,T,A>
    mem_fun1(S, (T::*f)(A));

template <class S, class T> class mem_fun_ref_t
          : public unary_function<T, S> {
    public:
      explicit mem_fun_ref_t(S (T::*p)());
      S operator()(T* p);
};

template <class S, class T, class A> class mem_fun1_ref_t
          : public binary_function<T, A, S> {
    public:
      explicit mem_fun1_ref_t(S (T::*p)(A));
      S operator()(T* p, A x);
};
template<class S, class T> mem_fun_ref_t<S,T>
     mem_fun_ref(S, (T::*f)());
template<class S, class T, class A> mem_fun1_ref_t<S,T,A>
     mem_fun1_ref(S, (T::*f)(A));
```

Example

```
  //
  // mem_fun  example
  //

  #include <functional>
  #include <list>

  int main(void)
  {
    int a1[] = {2,1,5,6,4};
    int a2[] = {11,4,67,3,14};
    list<int> s1(a1,a1+5);
    list<int> s2(a2,a2+5);

    // Build a list of lists
    list<list<int>* > l;
    l.insert(l.begin(),s1);
    l.insert(l.begin(),s2);
```

```
  // Sort each list in the list
  for_each(l.begin(),l.end(),mem_fun(&list<int>::sort));
}
```

See Also ***binary_function, function_objects, pointer_to_unary_function, ptr_fun***

Summary

Merge two sorted sequences into a third sequence.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2,
          class OutputIterator>
  OutputIterator
  merge(InputIterator first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator last2,
        OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator
  merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator last2,
        OutputIterator result, Compare comp);
```

Description

The ***merge*** algorithm merges two sorted sequences, specified by `[first1, last1)` and `[first2, last2)`, into the sequence specified by `[result, result + (last1 - first1) + (last2 - first2))`. The first version of the ***merge*** algorithm uses the less than operator (`<`) to compare elements in the two sequences. The second version uses the comparison function provided by the function call. If a comparison function is provided, ***merge*** assumes that both sequences were sorted using that comparison function.

The merge is stable. This means that if the two original sequences contain equivalent elements, the elements from the first sequence will always precede the matching elements from the second in the resulting sequence. The size of the result of a ***merge*** is equal to the sum of the sizes of the two argument sequences. ***merge*** returns an iterator that points to the end of the resulting sequence, i.e., `result + (last1 - first1) + (last2 -first2)`. The result of ***merge*** is undefined if the resulting range overlaps with either of the original ranges.

merge assumes that there are at least `(last1 - first1) + (last2 - first2)` elements following `result`, unless `result` has been adapted by an insert iterator.

Complexity

For ***merge*** at most `(last - first1) + (last2 - first2) - 1` comparisons are performed.

Example

```
//
// merge.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main()
 {
   int d1[4] = {1,2,3,4};
   int d2[8] = {11,13,15,17,12,14,16,18};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
   // Set up four destination vectors
   vector<int> v3(d2,d2 + 8),v4(d2,d2 + 8),
               v5(d2,d2 + 8),v6(d2,d2 + 8);
   // Set up one empty vector
   vector<int> v7;

   // Merge v1 with v2
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());
   // Now use comparator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),v4.begin(),
         less<int>());

   // In place merge v5
   vector<int>::iterator mid = v5.begin();
   advance(mid,4);
   inplace_merge(v5.begin(),mid,v5.end());
   // Now use a comparator on v6
   mid = v6.begin();
   advance(mid,4);
   inplace_merge(v6.begin(),mid,v6.end(),less<int>());

   // Merge v1 and v2 to empty vector using insert iterator
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
         back_inserter(v7));

   // Copy all cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;
   copy(v3.begin(),v3.end(),out);
   cout << endl;
   copy(v4.begin(),v4.end(),out);
   cout << endl;
   copy(v5.begin(),v5.end(),out);
   cout << endl;
   copy(v6.begin(),v6.end(),out);
   cout << endl;
   copy(v7.begin(),v7.end(),out);
   cout << endl;

   // Merge v1 and v2 to cout
   merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
```

```
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }

Output :
1 2 3 4
1 2 3 4
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
11 12 13 14 15 16 17 18
11 12 13 14 15 16 17 18
1 1 2 2 3 3 4 4
1 1 2 2 3 3 4 4
```

Warning

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

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

instead of:

```
vector<int>
```

See Also

Containers, inplace_merge

min

Algorithm

Summary Find and return the minimum of a pair of values

Synopsis

```
#include <algorithm>

template <class T>
 const T& min(const T&, const T&);

template <class T, class Compare>
 const T& min(const T& a, const T&, Compare);
```

Description The ***min*** algorithm determines and returns the minimum of a pair of values. In the second version of the algorithm, the optional argument `Compare` defines a comparison function that can be used in place of the default `operator<`. This function can be used with all the datatypes provided by the standard library.

min returns the first argument when the two arguments are equal.

Example

```
//
// max.cpp
//
 #include <algorithm>
 #include <iostream.h>

 int main(void)
 {
   double  d1 = 10.0, d2 = 20.0;

   // Find minimum
   double val1 = min(d1, d2);
   // val1 = 10.0

   // the greater comparator returns the greater of the
   // two values.
   double val2 = min(d1, d2, greater<double>());
   // val2 = 20.0;

   // Find maximum
   double val3 = max(d1, d2);
   // val3 = 20.0;

   // the less comparator returns the smaller of the
   // two values.
   // Note that, like every comparison in the STL, max is
   // defined in terms of the < operator, so using less here
   // is the same as using the max algorithm with a default
   // comparator.
```

```
   double val4 = max(d1, d2, less<double>());
   // val4 = 20

   cout << val1 << " " << val2 << " "
        << val3 << " " << val4 << endl;

   return 0;
 }

Output :
10 20 20 20
```

See Also ***max***, ***max_element***, ***min_element***

min_element

Algorithm

Summary

Finds the minimum value in a range.

Synopsis

```
#include <algorithm>

template <class ForwardIterator>
 ForwardIterator
 min_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
 InputIterator
 min_element(ForwardIterator first, ForwardIterator last,
             Compare comp);
```

Description

The ***min_element*** algorithm returns an iterator that denotes the minimum element in a sequence. If the sequence contains more than one copy of the minimum element, the iterator points to the first occurrence of the element. In the second version of the function, the optional argument `comp` defines a comparison function that can be used in place of the default `operator<`. This function can be used with all the datatypes provided by the standard library.

Algorithm ***min_element*** returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range same range, the following corresponding conditions hold:

```
!(*j < *i)
```

or

```
comp(*j, *i) == false.
```

Complexity

min_element performs exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

Example

```
//
// max_elem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int d1[5] = {1,3,5,32,64};
```

```
   // set up vector
   vector<int>      v1(d1,d1 + 5);

   // find the largest element in the vector
   iterator it1 = max_element(v1.begin(), v1.end());
   // it1 = v1.begin() + 4

   // find the largest element in the range from
   // the beginning of the vector to the 2nd to last
   iterator it2 = max_element(v1.begin(), v1.end()-1,
                      less<int>());
   // it2 = v1.begin() + 3

   // find the smallest element
   iterator it3 = min_element(v1.begin(), v1.end());
   // it3 = v1.begin()

   // find the smallest value in the range from
   // the beginning of the vector plus 1 to the end
   iterator it4 = min_element(v1.begin()+1, v1.end(),
                      less<int>());
   // it4 = v1.begin() + 1

   cout << *it1 << " " << *it2 << " "
        << *it3 << " " << *it4 << endl;

   return 0;
 }

Output :
64 32 1 3
```

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

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

instead of:

```
vector<int>
```

See Also ***max***, ***max_element***, ***min***

Summary

Returns the result of subtracting its second argument from its first.

Synopsis

```
#include<functional>

   template <class T>
   struct minus : public binary_function<T, T, T>;
```

Description

minus is a binary function object. Its `operator()` returns the result of `x` minus `y`. You can pass a ***minus*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***minus*** would be used in that algorithm in the following manner:

```
  vector<int> vec1;
  vector<int> vec2;
  vector<int> vecResult;
  .
  .
  .
  transform(vec1.begin(), vec1.end(),
            vec2.begin(),
            vecResult.begin(), minus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` minus `vec2(n)`.

Interface

```
template <class T>
struct minus : binary_function<T, T, T> {
  typedef typename binary_function<T, T, T>::second_argument_type
                                            second_argument_type;
  typedef typename binary_function<T, T, T>::first_argument_type
                                            first_argument_type;
  typedef typename binary_function<T, T, T>::result_type result_type;
  T operator() (const T&, const T&) const;
};
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***binary_function, function objects***

mismatch

Algorithm

Summary Compares elements from two sequences and returns the first two elements that don't match each other.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2>
  pair<InputIterator1,InputIterator2>
  mismatch(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
  pair<InputIterator1, Inputiterator2>
  mismatch(InputIterator first1, InputIterator1 last1,
           InputIterator2 first2,
           BinaryPredicate binary_pred);
```

Description The ***mismatch*** algorithm compares members of two sequences and returns two iterators (`i` and `j`) that point to the first location in each sequence where the sequences differ from each other. Notice that the algorithm denotes both a starting position and an ending position for the first sequence, but denotes only a starting position for the second sequence. ***mismatch*** assumes that the second sequence has at least as many members as the first sequence. If the two sequences are identical, ***mismatch*** returns a pair of iterators that point to the end of the first sequence and the corresponding location at which the comparison stopped in the second sequence.

The first version of ***mismatch*** checks members of a sequence for equality, while the second version lets you specify a comparison function. The comparison function must be a binary predicate.

The iterators `i` and `j` returned by ***mismatch*** are defined as follows:

```
j  == first2  +  (i  -  first1)
```

and `i` is the first iterator in the range `[first1, last1)` for which the appropriate one of the following conditions hold:

```
!(*i  ==  *(first2  +  (i  -  first1)))
```

or

```
binary_pred(*i, *(first2 + (i - first1))) == false
```

If all of the members in the two sequences match, ***mismatch*** returns a pair of `last1` and `first2 + (last1 - first1)`.

Complexity

At most `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// mismatch.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
 {
   typedef vector<int>::iterator  iterator;
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> vi1(d1,d1 + 4), vi2(d2,d2 + 4);

   // p1 will contain two iterators that point to the
   // first pair of elements that are different between
   // the two vectors
   pair<iterator, iterator> p1 = mismatch(vi1.begin(), vi1.end(),
                                         vi2.begin());

   // find the first two elements such that an element in the
   // first vector is greater than the element in the second
   // vector.
   pair<iterator, iterator> p2 = mismatch(vi1.begin(), vi1.end(),
                                        vi2.begin(),
                                        less_equal<int>());

   // Output results
   cout << *p1.first << ", " << *p1.second << endl;
   cout << *p2.first << ", " << *p2.second << endl;

   return 0;
 }
Output :
2, 3
3, 2
```

Warning

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

`vector<int, allocator<int> >`

instead of:

`vector<int>`

modulus

Function Object

Summary

Returns the remainder obtained by dividing the first argument by the second argument.

Synopsis

```
#include<functional>

   template <class T>
   struct modulus : public binary_function<T, T, T> ;
```

Description

modulus is a binary function object. Its `operator()` returns the remainder resulting from of `x` divided by `y`. You can pass a ***modulus*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***modulus*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), modulus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain the remainder of `vec1(n)` divided by `vec2(n)`.

Interface

```
template <class T>
struct modulus : binary_function<T, T, T> {
  typedef typename binary_function<T, T, T>::second_argument_type
                                          n    second_argument_type;
  typedef typename binary_function<T, T, T>::first_argument_type
                                               first_argument_type;
  typedef typename binary_function<T, T, T>::result_type result_type;
  T operator() (const T&, const T&) const;
};
```

Warning

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

`vector<int, allocator<int> >`

instead of

`vector<int>`

See Also ***binary_function, function object***

Summary

An associative container providing access to non-key values using keys. ***multimap*** keys are not required to be unique. A ***multimap*** supports bidirectional iterators.

Synopsis

```
#include <map>

template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<T> >
class multimap ;
```

Description

multimap <Key ,T, Compare, Allocator> provides fast access to stored values of type `T` which are indexed by keys of type `Key`. The default operation for key comparison is the `<` operator. Unlike ***map***, ***multimap*** allows insertion of duplicate keys.

multimap provides bidirectional iterators which point to an instance of `pair<const Key x, T y>` where `x` is the key and `y` is the stored value associated with that key. The definition of ***multimap*** provides a `typedef` to this pair called `value_type`.

The types used for both the template parameters `Key` and `T` must provide the following (where `T` is the `type`, `t` is a value of `T` and `u` is a `const value` of `T`):

```
  Copy constructors -  T(t) and T(u)
    Destructor        -  t.~T()
    Address of        -  &t and &u yielding T* and
                         const T* respectively
    Assignment        -  t = a where a is a
                         (possibly const) value of T
```

The type used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<T> >
 class multimap {

public:

// types

   typedef Key key_type;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;
```

```
   typename reference;
   typename const_reference;
   typename iterator;
   typename const_iterator;
   typename size_type;
   typename difference_type;
   typename reverse_iterator;
   typename const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type, bool>
    {
     friend class multimap<Key, T, Compare, Allocator>;

     public :
       bool operator() (const value_type&, const value_type&) const;
    };

// Construct/Copy/Destroy

   explicit multimap (const Compare& = Compare(), const Allocator& =
                     Allocator());
   template <class InputIterator>
    multimap (InputIterator, InputIterator,
              const Compare& = Compare(),
              const Allocator& = Allocator());
   multimap (const multimap<Key, T, Compare, Allocator>&);
   ~multimap ();
   multimap<Key, T, Compare, Allocator>& operator=
       (const multimap<Key, T, Compare, Allocator>&);

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modifiers

   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);

   iterator erase (iterator);
   size_type erase (const key_type&);
   iterator erase (iterator, iterator);
```

```
   void swap (multimap<Key, T, Compare, Allocator>&);

// Observers

   key_compare key_comp () const;
   value_compare value_comp () const;

// Multimap operations

   iterator find (const key_type&);
   const_iterator find (const key_type&) const;
   size_type count (const key_type&) const;

   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const key_type&);
   pair<const_iterator, const_iterator>
     equal_range (const key_type&) const;
};

// Non-member Operators

template <class Key, class T,class Compare, class Allocator>
 bool operator== (const multimap<Key, T, Compare, Allocator>&,
                  const multimap<Key, T, Compare, Allocator>&);

template <class Key, class T,class Compare, class Allocator>
 bool operator!= (const multimap<Key, T, Compare, Allocator>&,
                  const multimap<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator< (const multimap<Key, T, Compare, Allocator>&,
                 const multimap<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator> (const multimap<Key, T, Compare, Allocator>&,
                 const multimap<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator<= (const multimap<Key, T, Compare, Allocator>&,
                 const multimap<Key, T, Compare, Allocator>&);

template <class Key, class T, class Compare, class Allocator>
 bool operator>= (const multimap<Key, T, Compare, Allocator>&,
                 const multimap<Key, T, Compare, Allocator>&);

// Specialized Algorithms

template <class Key, class T, class Compare, class Allocator>
 void swap (multimap<Key, T, Compare, Allocator>&,
            multimap<Key, T, Compare, Allocator>&;
```

Constructors and Destructors

```
explicit multimap(const Compare& comp = Compare(),
                  const Allocator& alloc = Allocator());
```

Default constructor. Constructs an empty multimap that will use the optional relation `comp` to order keys and the allocator `alloc` for all storage management.

```
template <class InputIterator>
multimap(InputIterator first,
          InputIterator last,
          const Compare& comp = Compare()
          const Allocator& alloc = Allocator());
```

Constructs a multimap containing values in the range `[first, last)`. Creation of the new multimap is only guaranteed to succeed if the iterators `first` and `last` return values of type `pair<class Key, class T>.`

```
multimap(const multimap<Key, T, Compare, Allocator>& x);
```

Copy constructor. Creates a new multimap by copying all pairs of `key` and `value` from `x`.

```
~multimap();
```

The destructor. Releases any allocated memory for this multimap.

Assignment Operator

```
multimap<Key, T, Compare, Allocator>&
operator=(const multimap<Key, T, Compare, Allocator>& x);
```

Replaces the contents of `*this` with a copy of the multimap `x`.

Allocator

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Iterators

```
iterator
begin() ;
```

Returns a bidirectional `iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
const_iterator
begin() const;
```

Returns a `const_iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
iterator
end() ;
```

Returns a bidirectional `iterator` pointing to the last element stored in the multimap, i.e. the off-the-end value.

```
const_iterator
end() const;
```

Returns a `const_iterator` pointing to the last element stored in the multimap.

```
reverse_iterator
rbegin() ;
```

Returns a `reverse_iterator` pointing to the first element stored in the multimap. "First" is defined by the multimap's comparison operator, `Compare`.

```
const_reverse_iterator
rbegin() const;
```

Returns a `const_reverse_iterator` pointing to the first element stored in the multimap.

```
reverse_iterator
rend() ;
```

Returns a `reverse_iterator` pointing to the last element stored in the multimap, i.e., the off-the-end value.

```
const_reverse_iterator
rend() const;
```

Returns a `const_reverse_iterator` pointing to the last element stored in the multimap.

Member Functions

```
void
clear();
```

Erases all elements from the self.

```
size_type
count(const key_type& x) const;
```

Returns the number of elements in the multimap with the key value `x`.

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

Returns `true` if the multimap is empty, `false` otherwise.

```
pair<iterator,iterator>
equal_range(const key_type& x);
```

```
pair<const_iterator,const_iterator>
equal_range(const key_type& x) const;
```

Returns the pair `(lower_bound(x), upper_bound(x))`.

```
iterator
erase(iterator first, iterator last);
```

Providing the iterators `first` and `last` point to the same multimap and last is reachable from first, all elements in the range (`first, last`) will be deleted from the multimap. Returns an `iterator` pointing to the element following the last deleted element, or `end()`, if there were no elements after the deleted range.

```
iterator
erase(iterator position);
```

Deletes the multimap element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()`, if the deleted item was the last one in this list.

```
size_type
erase(const key_type& x);
```

Deletes the elements with the key value `x` from the map, if any exist. Returns the number of deleted elements, or 0 otherwise.

```
iterator
find(const key_type& x);
```

Searches the multimap for a pair with the key value `x` and returns an `iterator` to that pair if it is found. If such a pair is not found the value `end()` is returned.

```
const_iterator
find(const key_type& x) const;
```

Same as find above but returns a `const_iterator`.

```
iterator
insert(const value_type& x);
iterator
insert(iterator position, const value_type& x);
```

`x` is inserted into the multimap. A position may be supplied as a hint regarding where to do the insertion. If the insertion may be done right after `position` then it takes amortized constant time. Otherwise it will take `O(log N)` time.

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

Copies of each element in the range `[first, last)` will be inserted into the multimap. The iterators `first` and `last` must return values of type `pair<T1,T2>`. This operation takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multimap.

```
iterator
lower_bound(const key_type& x);
```

Returns an `iterator` to the first multimap element whose key is greater than or equal to `x`. If no such element exists then `end()` is returned.

```
const_iterator
lower_bound(const key_type& x) const;
```

Same as `lower_bound` above but returns a `const_iterator.`

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

Returns the maximum possible size of the multimap.

```
size_type
size() const;
```

Returns the number of elements in the multimap.

```
void
swap(multimap<Key, T, Compare, Allocator>& x);
```

Swaps the contents of the multimap `x` with the current multimap, `*this`.

```
iterator
upper_bound(const key_type& x);
```

Returns an `iterator` to the first element whose key is less than or equal to `x`. If no such element exists, then `end()` is returned.

```
const_iterator
upper_bound(const key_type& x) const;
```

Same as `upper_bound` above but returns a `const_iterator`.

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing `value_types` (`key,value` pairs) using the comparison operation, `Compare`, of the current multimap.

Non-member Operators

```
bool
operator==(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==).` Otherwise it returns `false`.

```
bool
operator!=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
bool
operator<(const multimap<Key, T, Compare, Allocator>& x,
          const multimap<Key, T, Compare, Allocator>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

```
bool
operator>(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `y < x`.

```
bool
operator<=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
bool
operator>=(const multimap<Key, T, Compare, Allocator>& x,
           const multimap<Key, T, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template<class Key, class T, class Compare, class Allocator>
void swap(multimap<Key, T, Compare, Allocator>& a,
           multimap<Key, T, Compare, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// multimap.cpp
//
 #include <string>
 #include <map>
 #include <iostream.h>

 typedef multimap<int, string, less<int> > months_type;

 // Print out a pair
 template <class First, class Second>
 ostream& operator<<(ostream& out,
                     const pair<First,Second>& p)
 {
   cout << p.second << " has " << p.first << " days";
   return out;
 }

 // Print out a multimap
 ostream& operator<<(ostream& out, months_type l)
 {
   copy(l.begin(),l.end(), ostream_iterator
             <months_type::value_type,char>(cout,"\n"));
   return out;
 }

 int main(void)
 {
   // create a multimap of months and the number of
   // days in the month
   months_type months;

   typedef months_type::value_type value_type;
```

```
   // Put the months in the multimap
   months.insert(value_type(31, string("January")));
   months.insert(value_type(28, string("February")));
   months.insert(value_type(31, string("March")));
   months.insert(value_type(30, string("April")));
   months.insert(value_type(31, string("May")));
   months.insert(value_type(30, string("June")));
   months.insert(value_type(31, string("July")));
   months.insert(value_type(31, string("August")));
   months.insert(value_type(30, string("September")));
   months.insert(value_type(31, string("October")));
   months.insert(value_type(30, string("November")));
   months.insert(value_type(31, string("December")));

   // print out the months
   cout << "All months of the year" << endl << months << endl;

   // Find the Months with 30 days
   pair<months_type::iterator,months_type::iterator> p =
          months.equal_range(30);

   // print out the 30 day months
   cout << endl << "Months with 30 days" << endl;
   copy(p.first,p.second,
     ostream_iterator<months_type::value_type,char>(cout,"\n"));

   return 0;
 }

Output :
All months of the year
February has 28 days
April has 30 days
June has 30 days
September has 30 days
November has 30 days
January has 31 days
March has 31 days
May has 31 days
July has 31 days
August has 31 days
October has 31 days
December has 31 days

Months with 30 days
April has 30 days
June has 30 days
September has 30 days
November has 30 days
```

Warnings

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

```
template <class InputIterator>
```

```
multimap (InputIterator, InputIterator,
          const Compare& = Compare(),
          const Allocator& = Allocator());
```

multimap also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a multimap in the following two ways:

```
multimap<int,int>::value_type intarray[10];
multimap<int,int> first_map(intarry, intarray + 10);
multimap<int,int>
  second_multimap(first_multimap.begin(), first_multimap.end());
```

but not this way:

```
multimap<long,long>
  long_multimap(first_multimap.begin(),first_multimap.end());
```

since the `long_multimap` and `first_multimap` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you need to always supply the `Compare` template argument and the `Allocator` template argument. For instance you'll have to write:

```
multimap<int, int, less<int>, allocator<int> >
```

instead of:

```
multimap<int, int>
```

See Also ***allocator, Containers, Iterators, map***

multiplies

Function Object

Summary A binary function object that returns the result of multiplying its first and second arguments.

Synopsis

```
#include<functional>

template <class T>
struct multiplies : binary_function<T, T, T> {
  typedef typename binary_function<T, T, T>::second_argument_type
                                             second_argument_type;
  typedef typename binary_function<T, T, T>::first_argument_type
                                             first_argument_type;
  typedef typename binary_function<T, T, T>::result_type result_type;
  T operator() (const T&, const T&) const;
};
```

Description ***multiplies*** is a binary function object. Its `operator()` returns the result of multiplying `x` and `y`. You can pass a ***multiplies*** object to any algorithm that uses a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***multiplies*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(), vec2.end(),
          vecResult.begin(), multiplies<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` times `vec2(n)`.

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***binary_function, function objects***

multiset

Container Class

Summary

An associative container providing fast access to stored key values. Storage of duplicate keys is allowed. A ***multiset*** supports bidirectional iterators.

Synopsis

```
#include <set>

template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
class multiset;
```

Description

multiset <Key, Compare, Allocator> provides fast access to stored key values. The default operation for key comparison is the `<` operator. Insertion of duplicate keys is allowed with a multiset.

multiset provides bidirectional iterators which point to a stored key.

Any type used for the template parameter `Key` must provide the following (where `T` is the `type`, `t` is a value of `T` and `u` is a `const value` of `T`):

```
    Copy constructors     T(t) and T(u)
    Destructor            t.~T()
    Address of            &t and &u yielding T* and
                          const T* respectively
    Assignment            t = a where a is a
                          (possibly const) value of T
```

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
 class multiset {

public:

// typedefs

   typedef Key key_type;
   typedef Key value_type;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef Allocator allocator_type;
   typename reference;
   typename const_reference;
```

```
   typename iterator;
   typename const_iterator;
   typename size_type;
   typename difference_type;
   typename reverse_iterator;
   typename const_reverse_iterator;

// Construct/Copy/Destroy

   explicit multiset (const Compare& = Compare(),
                      const Allocator& = Allocator());
   template <class InputIterator>
    multiset (InputIterator, InputIterator,
              const Compare& = Compare(),
              const Allocator& = Allocator());
   multiset (const multiset<Key, Compare, Allocator>&);
   ~multiset ();
   multiset<Key, Compare, Allocator>& operator= (const multiset<Key,
                                                 Compare,
                                                 Allocator>&);

// Iterators

   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacity

   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modifiers

   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
    void insert (InputIterator, InputIterator);

   iterator erase (iterator);
   size_type erase (const key_type&);
   iterator erase (iterator, iterator);
   void swap (multiset<Key, Compare, Allocator>&);
   void clear ();

// Observers

   key_compare key_comp () const;
   value_compare value_comp () const;

// Multiset operations
```

```
   iterator find (const key_type&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const key_type&) const;
   };

// Non-member Operators

template <class Key, class Compare, class Allocator>
 bool operator==
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator!=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator<
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator>
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator<=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator>=
    (const multiset<Key, Compare, Allocator>&,
     const multiset<Key, Compare, Allocator>&);

// Specialized Algorithms

template <class Key, class Compare, class Allocator>
 void swap ( multiset<Key, Compare, Allocator>&,
             multiset<Key, Compare, Allocator>&);
```

Constructor and Destructor

```
explicit multiset(const Compare& comp = Compare(),
                  const Allocator& alloc = Allocator());
```

Default constructor. Constructs an empty multiset which will use the optional relation `comp` to order keys, if it is supplied, and the allocator `alloc` for all storage management.

```
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
         const Compare& = Compare(),
         const Allocator& = Allocator());
```

Constructs a multiset containing values in the range `[first, last)`.

multiset(const multiset<Key, Compare, Allocator>& x);
Copy constructor. Creates a new multiset by copying all key values from `x`.

~multiset();
The destructor. Releases any allocated memory for this multiset.

Assignment Operator

multiset<Key, Compare, Allocator>&
operator=(const multiset<Key, Compare, Allocator>& x);
Replaces the contents of `*this` with a copy of the contents of `x`.

Allocator

allocator_type
get_allocator() const;
Returns a copy of the allocator used by self for storage management.

Iterators

iterator
begin();
Returns an `iterator` pointing to the first element stored in the multiset. "First" is defined by the multiset's comparison operator, `Compare`.

const_iterator
begin();
Returns a `const_iterator` pointing to the first element stored in the multiset.

iterator
end();
Returns an `iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

const_iterator
end();
Returns a `const_iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

reverse_iterator
rbegin();
Returns a `reverse_iterator` pointing to the first element stored in the multiset. "First" is defined by the multiset's comparison operator, `Compare`.

const_reverse_iterator
rbegin();
Returns a `const_reverse_iterator` pointing to the first element stored in the multiset.

reverse_iterator
rend();
Returns a `reverse_iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

`const_reverse_iterator`
`rend``();`

Returns a `const_reverse_iterator` pointing to the last element stored in the multiset, i.e., the off-the-end value.

Member Functions

`void`
`clear``();`

Erases all elements from the self.

`size_type`
`count``(const key_type& x) const;`

Returns the number of elements in the multiset with the key value `x`.

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

Returns `true` if the multiset is empty, `false` otherwise.

`pair<iterator,iterator>`
`equal_range``(const key_type& x)const;`

Returns the pair `(lower_bound(x), upper_bound(x))`.

`size_type`
`erase``(const key_type& x);`

Deletes all elements with the key value `x` from the multiset, if any exist. Returns the number of deleted elements.

`iterator`
`erase``(iterator position);`

Deletes the multiset element pointed to by the iterator `position`. Returns an iterator pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

`iterator`
`erase``(iterator first, iterator last);`

Providing the iterators `first` and `last` point to the same multiset and last is reachable from first, all elements in the range (`first, last`) will be deleted from the multiset. Returns an iterator pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

`iterator`
`find``(const key_type& x) const;`

Searches the multiset for a key value `x` and returns an iterator to that key if it is found. If such a value is not found the iterator `end()` is returned.

```
iterator
insert(const value_type& x);
```

```
iterator
insert(iterator position, const value_type& x);
```

`x` is inserted into the multiset. A position may be supplied as a hint regarding where to do the insertion. If the insertion may be done right after position, then it takes amortized constant time. Otherwise, it will take `O(log N)` time.

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

Copies of each element in the range `[first, last)` will be inserted into the multiset. This `insert` takes approximately `O(N*log(size()+N))` time.

```
key_compare
key_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multiset.

```
iterator
lower_bound(const key_type& x) const;
```

Returns an iterator to the first element whose key is greater than or equal to `x`. If no such element exists, `end()` is returned.

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

Returns the maximum possible size of the multiset `size_type.`

```
size_type
size() const;
```

Returns the number of elements in the multiset.

```
void
swap(multiset<Key, Compare, Allocator>& x);
```

Swaps the contents of the multiset `x` with the current multiset, `*this`.

```
iterator
upper_bound(const key_type& x) const;
```

Returns an iterator to the first element whose key is smaller than or equal to `x.` If no such element exists then `end()` is returned.

```
value_compare
value_comp() const;
```

Returns a function object capable of comparing key values using the comparison operation, `Compare`, of the current multiset.

Non-member Operators

```
template <class Key, class Compare, class Allocator>
operator==(const multiset<Key, Compare, Allocator>& x,
            const multiset<Key, Compare, Allocator>& y);
```

Returns `true` if all elements in `x` are element-wise equal to all elements in `y`, using `(T::operator==)`. Otherwise it returns `false`.

```
template <class Key, class Compare, class Allocator>
operator!=(const multiset<Key, Compare, Allocator>& x,
            const multiset<Key, Compare, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class Key, class Compare, class Allocator>
operator<(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `true` if `x` is lexicographically less than `y`. Otherwise, it returns `false`.

```
template <class Key, class Compare, class Allocator>
operator>(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `y < x`.

```
template <class Key, class Compare, class Allocator>
operator<=(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class Key, class Compare, class Allocator>
operator>=(const multiset<Key, Compare, Allocator>& x,
           const multiset<Key, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class Key, class Compare, class Allocator>
void swap(multiset<Key,Compare,Allocator>& a,
           multiset<Key,Compare,Allocator>&b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// multiset.cpp
//
#include <set>
#include <iostream.h>

 typedef multiset<int, less<int>, allocator> set_type;

 ostream& operator<<(ostream& out, const set_type& s)
 {
   copy(s.begin(),s.end(),
        ostream_iterator<set_type::value_type,char>(cout," "));
   return out;
 }
```

```
int main(void)
{
  // create a multiset of ints
  set_type  si;
  int  i;

  for (int j = 0; j < 2; j++)
  {
    for(i = 0; i < 10; ++i) {
      // insert values with a hint
      si.insert(si.begin(), i);
    }
  }

  // print out the multiset
  cout << si << endl;

  // Make another int multiset and an empty multiset
  set_type si2, siResult;
  for (i = 0; i < 10; i++)
     si2.insert(i+5);
  cout << si2 << endl;

  // Try a couple of set algorithms
  set_union(si.begin(),si.end(),si2.begin(),si2.end(),
         inserter(siResult,siResult.begin()));
  cout << "Union:" << endl << siResult << endl;

  siResult.erase(siResult.begin(),siResult.end());
  set_intersection(si.begin(),si.end(),
         si2.begin(),si2.end(),
         inserter(siResult,siResult.begin()));
  cout << "Intersection:" << endl << siResult << endl;

  return 0;
}
```

```
Output:
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 6 7 8 9 10 11 12 13 14
Union:
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 11 12 13 14
Intersection:
5 6 7 8 9
```

Warnings

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for ***multiset<Key, Compare, Allocator>***, which takes two templated iterators:

```
template <class InputIterator>
multiset (InputIterator, InputIterator,
          const Compare& = Compare(),
          const Allocator& = Allocator());
```

multiset also has an `insert` function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on). You can also use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates, you can construct a ***multiset*** in the following two ways:

```
int intarray[10];
multiset<int> first_multiset(intarray,
                                                   intarray +10);
multiset<int>
   second_multiset(first_multiset.begin(), first_multiset.end());
```

but not this way:

```
multiset<long>
   long_multiset(first_multiset.begin(),first_multiset.end());
```

since the `long_multiset` and `first_multiset` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you need to always supply the `Compare` template argument and the `Allocator` template argument. For instance, you'll have to write:

```
multiset<int, less<int>, allocator<int> >
```

instead of:

```
multiset<int>
```

See Also

allocator, Containers, Iterators, set

negate

Function Object

Summary Unary function object that returns the negation of its argument.

Synopsis

```
#include <functional>

   template <class T>
   struct negate : public unary_function<T, T>;
```

Description ***negate*** is a unary function object. Its `operator()` returns the negation of its argument, i.e., `true` if its argument is `false`, or `false` if its argument is `true`. You can pass a ***negate*** object to any algorithm that requires a unary function. For example, the ***transform*** algorithm applies a unary operation to the values in a collection and stores the result. ***negate*** could be used in that algorithm in the following manner:

```
  vector<int> vec1;
  vector<int> vecResult;
  .
  .
  .
  transform(vec1.begin(), vec1.end(),
            vecResult.begin(), negate<int>());
```

After this call to ***transform***, `vecResult(n)` will contain the negation of the element in `vec1(n)`.

Interface

```
template <class T>
struct negate : unary_function<T, T> {
  typedef typename unary_function<T,T>::argument_type argument_type;
  typedef typename unary_function<T,T>::result_type result_type;
  T operator() (const T&) const;
};
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***function objects, unary_function***

negators

Function Object

Summary

Function adaptors and function objects used to reverse the sense of predicate function objects.

Synopsis

```
#include <functional>

template <class Predicate>
class unary_negate;

template <class Predicate>
unary_negate<Predicate> not1(const Predicate&);

template <class Predicate>
class binary_negate;

template <class Predicate>
binary_negate<Predicate> not2(const Predicate&);
```

Description

Negators ***not1*** and ***not2*** are functions that take predicate function objects as arguments and return predicate function objects with the opposite sense. Negators work only with function objects defined as subclasses of the classes ***unary_function*** and ***binary_function***. ***not1*** accepts and returns unary predicate function objects. ***not2*** accepts and returns binary predicate function objects.

unary_negate and ***binary_negate*** are function object classes that provide return types for the negators, ***not1*** and ***not2***.

Interface

```
template <class Predicate>
class unary_negate
  : public unary_function<typename Predicate::argument_type, bool> {

public:
  typedef typename unary_function<typename Predicate::argument_type,
                                  bool>::argument_type argument_type;
  typedef typename unary_function<typename Predicate::argument_type,
                                  bool>::result_type result_type;
  explicit unary_negate (const Predicate&);
  bool operator() (const argument_type&) const;
};

template<class Predicate>
unary_negate <Predicate> not1 (const Predicate&);

template<class Predicate>
class binary_negate
  : public binary_function<typename Predicate::first_argument_type,
```

```
                              typename Predicate::second_argument_type,
                              bool>
{
public:
  typedef typename binary_function<typename
    Predicate::first_argument_type,
    typename Predicate::second_argument_type,
    bool>::second_argument_type second_argument_type;
  typedef typename binary_function<typename
    Predicate::first_argument_type,
    typename Predicate::second_argument_type,
    bool>::first_argument_type first_argument_type;
  typedef typename binary_function<typename
    Predicate::first_argument_type,
    typename Predicate::second_argument_type, bool>::result_type
                                                        result_type;
  explicit binary_negate (const Predicate&);
  bool operator() (const first_argument_type&,
                    const second_argument_type&) const;
};

template <class Predicate>
binary_negate<Predicate> not2 (const Predicate&);
```

Example

```
  //
  // negator.cpp
  //
   #include<functional>
   #include<algorithm>
   #include <iostream.h>

   //Create a new predicate from unary_function
   template<class Arg>
   class is_odd : public unary_function<Arg, bool>
   {
     public:
     bool operator()(const Arg& arg1) const
     {
       return (arg1 % 2 ? true : false);
     }
   };

   int main()
   {
     less<int> less_func;

     // Use not2 on less
     cout << (less_func(1,4) ? "TRUE" : "FALSE") << endl;
     cout << (less_func(4,1) ? "TRUE" : "FALSE") << endl;
     cout << (not2(less<int>())(1,4) ? "TRUE" : "FALSE")
          << endl;
     cout << (not2(less<int>())(4,1) ? "TRUE" : "FALSE")
          << endl;

     //Create an instance of our predicate
     is_odd<int> odd;
```

```
    // Use not1 on our user defined predicate
    cout << (odd(1) ? "TRUE" : "FALSE") << endl;
    cout << (odd(4) ? "TRUE" : "FALSE") << endl;
    cout << (not1(odd)(1) ? "TRUE" : "FALSE") << endl;
    cout << (not1(odd)(4) ? "TRUE" : "FALSE") << endl;

    return 0;
  }
Output :
TRUE
FALSE
FALSE
TRUE
TRUE
FALSE
FALSE
TRUE
```

See Also ***algorithm, binary_function, function_object, unary_function***

Summary

Generate successive permutations of a sequence based on an ordering function.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator>
bool next_permutation (BidirectionalIterator first,
                       BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
 bool next_permutation (BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

Description

The permutation-generating algorithms (***next_permutation*** and ***prev_permutation***) assume that the set of all permutations of the elements in a sequence is lexicographically sorted with respect to `operator<` or `comp`. So, for example, if a sequence includes the integers 1 2 3, that sequence has six permutations, which, in order from first to last are: 1 2 3 , 1 3 2, 2 1 3, 2 3 1, 3 1 2, and 3 2 1.

The ***next_permutation*** algorithm takes a sequence defined by the range `[first, last)` and transforms it into its next permutation, if possible. If such a permutation does exist, the algorithm completes the transformation and returns `true`. If the permutation does not exist, ***next_permutation*** returns `false`, and transforms the permutation into its "first" permutation (according to the lexicographical ordering defined by either `operator<`, the default used in the first version of the algorithm, or `comp`, which is user-supplied in the second version of the algorithm.)

For example, if the sequence defined by `[first, last)` contains the integers 3 2 1 (in that order), there is *not* a "next permutation." Therefore, the algorithm transforms the sequence into its first permutation (1 2 3) and returns `false`.

Complexity

At most `(last - first)/2` swaps are performed.

Example

```
//
// permute.cpp
//
 #include <numeric>    //for accumulate
 #include <vector>        //for vector
 #include <functional> //for less
```

```
#include <iostream.h>

int main()
{
  //Initialize a vector using an array of ints
  int  a1[] = {0,0,0,0,1,0,0,0,0,0};
  char a2[] = "abcdefghji";
  //Create the initial set and copies for permuting
  vector<int>  m1(a1, a1+10);
  vector<int>  prev_m1((size_t)10), next_m1((size_t)10);
  vector<char> m2(a2, a2+10);
  vector<char> prev_m2((size_t)10), next_m2((size_t)10);
  copy(m1.begin(), m1.end(), prev_m1.begin());
  copy(m1.begin(), m1.end(), next_m1.begin());
  copy(m2.begin(), m2.end(), prev_m2.begin());
  copy(m2.begin(), m2.end(), next_m2.begin());
  //Create permutations
  prev_permutation(prev_m1.begin(),
                   prev_m1.end(),less<int>());
  next_permutation(next_m1.begin(),
                   next_m1.end(),less<int>());
  prev_permutation(prev_m2.begin(),
                   prev_m2.end(),less<int>());
  next_permutation(next_m2.begin(),
                   next_m2.end(),less<int>());
  //Output results
  cout << "Example 1: " << endl << "     ";
  cout << "Original values:      ";
  copy(m1.begin(),m1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << "     ";
  cout << "Previous permutation: ";
  copy(prev_m1.begin(),prev_m1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl<< "     ";
  cout << "Next Permutation:     ";
  copy(next_m1.begin(),next_m1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  cout << "Example 2: " << endl << "     ";
  cout << "Original values: ";
  copy(m2.begin(),m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << "     ";
  cout << "Previous Permutation: ";
  copy(prev_m2.begin(),prev_m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << "     ";
  cout << "Next Permutation:     ";
  copy(next_m2.begin(),next_m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << endl;

  return 0;
}
```

```
Output :
Example 1:
     Original values:      0 0 0 0 1 0 0 0 0 0
     Previous permutation: 0 0 0 0 0 1 0 0 0 0
     Next Permutation:     0 0 0 1 0 0 0 0 0 0
Example 2:
     Original values: a b c d e f g h j i
     Previous Permutation: a b c d e f g h i j
     Next Permutation:     a b c d e f g i h j
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

prev_permutation

not1

Function Adaptor

Summary

Function adaptor used to reverse the sense of a unary predicate function object.

Synopsis

```
#include <functional>

template<class Predicate>
unary_negate <Predicate> not1 (const Predicate&);
```

Description

not1 is a function adaptor, known as a negator, that takes a unary predicate function object as its argument and returns a unary predicate function object that is the complement of the original. ***unary_negate*** is a function object class that provides a return type for the ***not1*** negator.

Note that ***not1*** works only with function objects that are defined as subclasses of the class ***unary_function***.

See Also

negators, ***not2***, ***unary_function***, ***unary_negate***, ***pointer_to_unary_function***

not2

Function Adaptor

Summary

Function adaptor used to reverse the sense of a binary predicate function object.

Synopsis

```
#include <functional>

template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred);
```

Description

not2 is a function adaptor, known as a negator, that takes a binary predicate function object as its argument and returns a binary predicate function object that is the complement of the original. ***binary_negate*** is a function object class that provides a return type for the ***not2*** negator.

Note that ***not2*** works only with function objects that are defined as subclasses of the class ***binary_function***.

See Also

binary_function, ***binary_negate***, ***negators***, ***not1***, ***pointer_to_binary_function, unary_negate***

Summary Binary function object that returns `true` if its first argument is not equal to its second.

Synopsis

```
#include <functional>

template <class T>
struct not_equal_to : public binary_function<T, T, bool> ;
```

Description

not_equal_to is a binary function object. Its `operator()` returns `true` if `x` is not equal to `y`. You can pass a ***not_equal_to*** object to any algorithm that requires a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***not_equal_to*** would be used in that algorithm in the following manner:

```
vector<int> vec1;
vector<int> vec2;
vector<int> vecResult;
.
.
.
transform(vec1.begin(), vec1.end(),
          vec2.begin(),
          vecResult.begin(), not_equal_to<int>());
```

After this call to ***transform***, `vecResult(n)` will contain a "1" if `vec1(n)` was not equal to `vec2(n)` or a "1" if `vec1(n)` was equal to `vec2(n)`.

Interface

```
template <class T>
struct not_equal_to : binary_function<T, T, bool> {
  typedef typename binary_function<T, T, bool>::second_argument_type
                                              second_argument_type;
  typedef typename binary_function<T, T, bool>::first_argument_type
                                              first_argument_type;
  typedef typename binary_function<T, T, bool>::result_type
                                              result_type;
  bool operator() (const T&, const T&) const;
};
```

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

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

instead of :

```
vector<int>
```

See Also ***binary_function, function object***

nth_element

Algorithm

Summary

Rearranges a collection so that all elements lower in sorted order than the nth element come before it and all elements higher in sorter order than the nth element come after it.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
 void nth_element (RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
 void nth_element (RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last,
                   Compare comp);
```

Description

The ***nth_element*** algorithm rearranges a collection according to either the default comparison operator (`>`) or the provided comparison operator. After the algorithm applies, three things are true:

- The element that would be in the nth position if the collection were completely sorted is in the nth position
- All elements prior to the nth position would precede that position in an ordered collection
- All elements following the nth position would follow that position in an ordered collection

That is, for any iterator `i` in the range `[first, nth)` and any iterator `j` in the range `[nth, last)` it holds that `!(*i > *j)` or `comp(*i, *j) == false`.

Note that the elements that precede or follow the nth position are not necessarily sorted relative to each other. The ***nth_element*** algorithm does *not* sort the entire collection.

Complexity

The algorithm is linear, on average, where `N` is the size of the range `[first, last)`.

Example

```
//
// nthelem.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>
```

```
template<class RandomAccessIterator>
void quik_sort(RandomAccessIterator start,
               RandomAccessIterator end)
{
  size_t dist = 0;
  distance(start, end, dist);

  //Stop condition for recursion
  if(dist > 2)
  {
    //Use nth_element to do all the work for quik_sort
    nth_element(start, start+(dist/2), end);

    //Recursive calls to each remaining unsorted portion
    quik_sort(start, start+(dist/2-1));
    quik_sort(start+(dist/2+1), end);
  }

  if(dist == 2 && *end < *start)
    swap(start, end);
}

int main()
{
  //Initialize a vector using an array of ints
  int arr[10] = {37, 12, 2, -5, 14, 1, 0, -1, 14, 32};
  vector<int> v(arr, arr+10);

  //Print the initial vector
  cout << "The unsorted values are: " << endl << "     ";
  vector<int>::iterator i;
  for(i = v.begin(); i != v.end(); i++)
    cout << *i << ", ";
  cout << endl << endl;

  //Use the new sort algorithm
  quik_sort(v.begin(), v.end());

  //Output the sorted vector
  cout << "The sorted values are: " << endl << "     ";
  for(i = v.begin(); i != v.end(); i++)
    cout << *i << ", ";
  cout << endl << endl;

  return 0;
}

Output :
The unsorted values are:
     37, 12, 2, -5, 14, 1, 0, -1, 14, 32,
The sorted values are:
     -5, -1, 0, 1, 2, 12, 14, 14, 32, 37,
```

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

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

instead of :

```
vector<int>
```

See Also ***Algorithms***

Summary

A class for representing information about scalar types.

Specializations

```
numeric_limits<float>
numeric_limits<double>
numeric_limits<long double>
numeric_limits<short>
numeric_limits<unsigned short>
numeric_limits<int>
numeric_limits<unsigned int>
numeric_limits<long>
numeric_limits<unsigned long>
numeric_limits<char>
numeric_limits<wchar_t>
numeric_limits<unsigned char>
numeric_limits<signed char>
numeric_limits<bool>
```

Synopsis

```
#include <limits>

template <class T>
class numeric_limits ;
```

Description

numeric_limits is a class for representing information about scalar types. Specializations are provided for each fundamental type, both floating point and integer, including `bool`.

This class encapsulates information that is contained in the `<climits>` and `<cfloat>` headers, as well as providing additional information that is not contained in any existing C or C++ header.

Not all of the information provided by members is meaningful for all specializations of ***numeric_limits***. Any value which is not meaningful for a particular type is set to `0` or `false`.

Interface

```
template <class T>
 class numeric_limits {

 public:

 // General -- meaningful for all specializations.

    static const bool is_specialized ;
    static T min ();
    static T max ();
    static const int radix ;
```

```
   static const int digits ;
   static const int digits10 ;
   static const bool is_signed ;
   static const bool is_integer ;
   static const bool is_exact ;
   static const bool traps ;
   static const bool is_modulo ;
   static const bool is_bounded ;

// Floating point specific.

   static T epsilon ();
   static T round_error ();
   static const int min_exponent10 ;
   static const int max_exponent10 ;
   static const int min_exponent ;

   static const int max_exponent ;
   static const bool has_infinity ;
   static const bool has_quiet_NaN ;
   static const bool has_signaling_NaN ;
   static const bool is_iec559 ;
   static const bool has_denorm ;
   static const bool tinyness_before ;
   static const float_round_style round_style ;
   static T denorm_min ();
   static T infinity ();
   static T quiet_NaN ();
   static T signaling_NaN ();
};

enum float_round_style {
  round_indeterminate       = -1,
  round_toward_zero         =  0,
  round_to_nearest          =  1,
  round_toward_infinity     =  2,
  round_toward_neg_infinity =  3
};
```

Member Fields and Functions

static T
denorm_min ();

Returns the minimum denormalized value. Meaningful for all floating point types. For types that do not allow denormalized values, this method must return the minimum normalized value.

static const int
digits ;

Number of radix digits which can be represented without change. For built-in integer types, `digits` will usually be the number of non-sign bits in the representation. For floating point types, `digits` is the number of radix digits in the mantissa. This member is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static const int
digits10 ;
```

Number of base 10 digits that can be represented without change. Meaningful for all specializations that declare `is_bounded` to be `true`.

```
static T
epsilon ();
```

Returns the machine epsilon (the difference between 1 and the least value greater than 1 that is representable). This function is meaningful for floating point types only.

```
static const bool
has_denorm ;
```

This field is `true` if the type allows denormalized values (variable number of exponent bits). It is meaningful for floating point types only.

```
static const bool
has_infinity ;
```

This field is `true` if the type has a representation for positive infinity. It is meaningful for floating point types only. This field must be `true` for any type claiming conformance to IEC 559.

```
static const bool
has_quiet_NaN ;
```

This field is `true` is the type has a representation for a quiet (non-signaling) "Not a Number". It is meaningful for floating point types only and must be `true` for any type claiming conformance to IEC 559.

```
static const bool
has_signaling_NaN ;
```

This field is `true` if the type has a representation for a signaling "Not a Number". It is meaningful for floating point types only, and must be `true` for any type claiming conformance to IEC 559.

```
static T
infinity ();
```

Returns the representation of positive infinity, if available. This member function is meaningful for only those specializations that declare `has_infinity` to be `true`. Required for any type claiming conformance to IEC 559.

```
static const bool
is_bounded ;
```

This field is `true` if the set of values representable by the type is finite. All built-in C types are bounded; this member would be `false` for arbitrary precision types.

```
static const bool
```
`is_exact ;`

This static member field is `true` if the type uses an exact representation. All integer types are exact, but not vice versa. For example, rational and fixed-exponent representations are exact but not integer. This member is meaningful for all specializations.

```
static const bool
```
`is_iec559 ;`

This member is `true` if and only if the type adheres to the IEC 559 standard. It is meaningful for floating point types only. Must be `true` for any type claiming conformance to IEC 559.

```
static const bool
```
`is_integer ;`

This member is `true` if the type is integer. This member is meaningful for all specializations.

```
static const bool
```
`is_modulo ;`

This field is `true` if the type is modulo. Generally, this is `false` for floating types, `true` for unsigned integers, and `true` for signed integers on most machines. A type is modulo if it is possible to add two positive numbers, and have a result that wraps around to a third number, which is less.

```
static const bool
```
`is_signed ;`

This member is `true` if the type is signed. This member is meaningful for all specializations.

```
static const bool
```
`is_specialized ;`

Indicates whether ***numeric_limits*** has been specialized for type `T`. This flag must be `true` for all specializations of `numeric_limits`. In the default ***numeric_limits<T>*** template, this flag must be `false`.

```
static T
```
`max ();`

Returns the maximum finite value. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static const int
```
`max_exponent ;`

Maximum positive integer such that the radix raised to that power is in range. This field is meaningful for floating point types only.

```
static const int
```
`max_exponent10` `;`

Maximum positive integer such that 10 raised to that power is in range. This field is meaningful for floating point types only.

```
static T
```
`min` `();`

Returns the minimum finite value. For floating point types with denormalization, `min()` must return the minimum normalized value. The minimum denormalized value is provided by `denorm_min()`. This function is meaningful for all specializations that declare `is_bounded` to be `true`.

```
static const int
```
`min_exponent` `;`

Minimum negative integer such that the radix raised to that power is in range. This field is meaningful for floating point types only.

```
static const int
```
`min_exponent10` `;`

Minimum negative integer such that 10 raised to that power is in range. This field is meaningful for floating point types only.

```
static T
```
`quiet_NaN ()``;`

Returns the representation of a quiet "Not a Number", if available. This function is meaningful only for those specializations that declare `has_quiet_NaN` to be true. This field is required for any type claiming conformance to IEC 559.

```
static const int
```
`radix` `;`

For floating types, specifies the base or radix of the exponent representation (often 2). For integer types, this member must specify the base of the representation. This field is meaningful for all specializations.

```
static T
```
`round_error ()``;`

Returns the measure of the maximum rounding error. This function is meaningful for floating point types only.

```
static const float_round_style
```
`round_style` `;`

The rounding style for the type. Specializations for integer types must return `round_toward_zero`. This is meaningful for all floating point types.

`static T`
`signaling_NaN();`

Returns the representation of a signaling "Not a Number", if available. This function is meaningful for only those specializations that declare `has_signaling_NaN` to be `true`. This function must be meaningful for any type claiming conformance to IEC 559.

`static const bool`
`tinyness_before ;`

This member is `true` if `tinyness` is detected before rounding. It is meaningful for floating point types only.

`static const bool`
`traps ;`

This field is `true` if trapping is implemented for this type. The `traps` field is meaningful for all specializations.

Example

```
//
// limits.cpp
//
 #include <limits>


 int main()
 {
    numeric_limits<float> float_info;
    if (float_info.is_specialized && float_info.has_infinity)
    {
      // get value of infinity
      float finfinity=float_info.infinity();
    }
    return 0;
 }
```

Warning

The specializations for `wide chars` and `bool` will only be available if your compiler has implemented them as real types and not simulated them with typedefs.

See Also

IEEE Standard for Binary Floating-Point Arithmetic, 345 East 47th Street, New York, NY 10017

Language Independent Arithmetic (LIA-1)

operator!=, operator>, operator<=, operator>=

Summary

Operators for the C++ Standard Template Library

Synopsis

```
#include <utility>

namespace rel_ops {

template <class T>
bool operator!= (const T&, const T&);

template <class T>
 bool operator> (const T&, const T&);

template <class T>
 bool operator<= (const T&, const T&);

template <class T>
 bool operator>= (const T&, const T&);
}
```

Description

To avoid redundant definitions of `operator!=` out of `operator==` and of `operators >`, `<=`, and `>=` out of `operator<`, the library provides these definitions:

```
operator!= returns !(x==y),
operator>  returns y<x,
operator<= returns !(y<x), and
operator>= returns !(x<y).
```

To avoid clashes with other global operators these definitions are contained in the namespace `rel_ops`. To use them either scope explicitly or provide a using declaration (e.g. `using_namespace_rel_ops`).

Summary

Stream iterators provide iterator capabilities for ostreams and istreams. They allow generic algorithms to be used directly on streams.

Synopsis

```
#include <ostream>

template <class T, class charT,
          class traits = char_traits<charT> >
class ostream_iterator
 : public iterator<output_iterator_tag,void,void>;
```

Description

Stream iterators provide the standard iterator interface for input and output streams.

The class ***ostream_iterator*** writes elements to an output stream. If you use the constructor that has a second, `char *` argument, then that string will be written after every element . (The string must be null-terminated.) Since an ostream iterator is an output iterator, it is not possible to get an element out of the iterator. You can only assign to it.

Interface

```
template <class T, class charT,
          class traits = char_traits<charT> >
 class ostream_iterator
   : public iterator<output_iterator_tag,void,void>
{
 public:
    typedef T value_type;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT,traits> ostream_type;

    ostream_iterator(ostream&);
    ostream_iterator (ostream&, const char*);
    ostream_iterator (const
            ostream_iterator<T,charT,char_traits<charT> >&);
    ~ostream_itertor ();

    ostream_iterator<T,charT,char_traits<charT> >&
         operator=(const T&);
    ostream_iterator<T,charT,char_traits<charT> >&
         operator* () const;
    ostream_iterator<T,charT,char_traits<charT> >& operator++ ();
    ostream_iterator<T,charT,char_traits<charT> >  operator++ (int);
 };
```

Types

`value_type`;
Type of value to stream in.

char_type;
Type of character the stream is built on.

traits_type;
Traits used to build the stream.

ostream_type;
Type of stream this iterator is constructed on.

Constructors

ostream_iterator (ostream& s);
Construct an ***ostream_iterator*** on the given stream.

ostream_iterator (ostream& s, const char* delimiter);
Construct an ***ostream_iterator*** on the given stream. The null terminated string delimitor is written to the stream after every element.

ostream_iterator (const ostream_iterator<T>& x);
Copy constructor.

Destructor

~ostream_iterator ();
Destructor

Operators

const T&
operator= (const T& value);
Shift the value T onto the output stream.

const T& ostream_iterator<T>&
operator* ();

ostream_iterator<T>&
operator++();

ostream_iterator<T>
operator++ (int);
These operators all do nothing. They simply allow the iterator to be used in common constructs.

Example

```
#include <iterator>
#include <numeric>
#include <deque>
#include <iostream.h>

int main ()
{
  //
  // Initialize a vector using an array.
  //
  int arr[4] = { 3,4,7,8 };
  int total=0;
  deque<int> d(arr+0, arr+4);
  //
  // stream the whole vector and a sum to cout
  //
```

```
    copy(d.begin(),d.end()-1,
         ostream_iterator<int,char>(cout," + "));
    cout << *(d.end()-1) << " = " <<
           accumulate(d.begin(),d.end(),total) << endl;
    return 0;
  }
```

Warning

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

`deque<int, allocator<int> >`

instead of :

`deque<int>`

See Also

istream_iterator*, *iterators

Summary

A write-only, forward moving iterator.

Description

For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Output iterators are write-only, forward moving iterators that satisfy the requirements listed below. Note that unlike other iterators used with the standard library, output iterators cannot be constant.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Output Iterators

The following expressions must be valid for output iterators:

`X(a)`	copy constructor, `a == X(a)`.
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`*a = t`	result is not used
`++r`	returns `X&`
`r++`	return value convertible to `const X&`
`*r++ = t`	result is not used

The only valid use for the `operator *` is on the left hand side of the assignment statement.

Algorithms using output iterators should be single pass algorithms. That is, they should not pass through the same iterator twice.

See Also ***Iterators*, *Input Iterators***

pair

Utility Class

Summary A template for heterogeneous pairs of values.

Synopsis

```
#include <utility>

template <class T1, class T2>
struct pair ;
```

Description The ***pair*** class provides a template for encapsulating pairs of values that may be of different types.

Interface

```
template <class T1, class T2>
 struct pair {
        T1 first;
        T2 second;
        pair();
        pair (const T1&, const T2&);
        ~pair();
};

template <class T1, class T2>
 bool operator== (const pair<T1, T2>&,
                  const pair T1, T2>&);

template <class T1, class T2>
 bool operator!= (const pair<T1, T2>&,
                  const pair T1, T2>&);

template <class T1, class T2>
 bool operator< (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator> (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator<= (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 bool operator>= (const pair<T1, T2>&,
                 const pair T1, T2>&);

template <class T1, class T2>
 pair<T1,T2> make_pair (const T1&, const T2&);
```

Constructors and Destructors

pair `();`

Default constructor. Initializes `first` and `second` using their default constructors.

pair `(const T1& x, const T2& y);`

The constructor creates a pair of types `T1` and `T2`, making the necessary conversions in `x` and `y`.

~pair `();`

Destructor.

Non-member Operators

```
template <class T1, class T2>
 bool operator== (const pair<T1, T2>& x,
                  const pair T1, T2>& y);
```

Returns `true` if `(x.first == y.first && x.second == y.second)` is `true`. Otherwise it returns `false`.

```
template <class T1, class T2>
 bool operator!= (const pair<T1, T2>& x,
                  const pair T1, T2>& y);
```

Returns `!(x==y)`.

```
template <class T1, class T2>
bool operator< (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `true` if `(x.first < y.first || (!(y.first < x.first) && x.second < y.second))` is `true`. Otherwise it returns `false`.

```
template <class T1, class T2>
bool operator> (const pair<T1, T2>& x,
                const pair T1, T2>& y);
```

Returns `y < x`.

```
template <class T1, class T2>
bool operator<= (const pair<T1, T2>& x,
                 const pair T1, T2>& y);
```

Returns `!(y < x)`.

```
template <class T1, class T2>
bool operator>= (const pair<T1, T2>& x,
                 const pair T1, T2>& y);
```

Returns `!(x < y)`.

Non-member Functions

```
template <class T1, class T2>
pair<T1,T2>
make_pair(x,y);
```

`make_pair(x,y)` creates a pair by deducing and returning the types of `x` and `y`.

partial_sort

Algorithm

Summary Templated algorithm for sorting collections of entities.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
 void partial_sort (RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
 void partial_sort (RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
```

Description The partial_sort algorithm takes the range `[first,last)` and places the first `middle - first` values into sorted order. The result is that the range `[first, middle)`is sorted like it would be if the entire range `[first,last)` were sorted. The remaining elements in the range (those in `[middle, last)`) are not in any defined order. The first version of the algorithm uses less than (`operator<`) as the comparison operator for the sort. The second version uses the comparison function `comp.`

Complexity ***partial_sort*** does approximately `(last - first) * log(middle-first)` comparisons.

Example

```
//
// partsort.cpp
//
#include <vector>
#include <algorithm>
#include <iostream.h>

int main()
 {
   int d1[20] = {17, 3,  5,  -4, 1, 12, -10, -1, 14, 7,
                 -6, 8, 15, -11, 2, -2,  18,  4, -3, 0};
   //
   // Set up a vector.
   //
   vector<int> v1(d1+0, d1+20);
   //
   // Output original vector.
   //
   cout << "For the vector: ";
   copy(v1.begin(), v1.end(),
        ostream_iterator<int,char>(cout," "));
   //
```

```
    // Partial sort the first seven elements.
    //
    partial_sort(v1.begin(), v1.begin()+7, v1.end());
    //
    // Output result.
    //
    cout << endl << endl << "A partial_sort of seven elements
                             gives: "
         << endl << "     ";
    copy(v1.begin(), v1.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl;
    //
    // A vector of ten elements.
    //
    vector<int> v2(10, 0);
    //
    // Sort the last ten elements in v1 into v2.
    //
    partial_sort_copy(v1.begin()+10, v1.end(), v2.begin(),
                      v2.end());
    //
    // Output result.
    //
    cout << endl << "A partial_sort_copy of the last ten elements
                     gives: "
         << endl << "     ";
    copy(v2.begin(), v2.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl;

    return 0;
  }

Output :
For the vector: 17 3 5 -4 1 12 -10 -1 14 7 -6 8 15 -11 2 -2 18 4 -
3 0
A partial_sort of seven elements gives:
     -11 -10 -6 -4 -3 -2 -1 17 14 12 7 8 15 5 3 2 18 4 1 0
A partial_sort_copy of the last ten elements gives:
     0 1 2 3 4 5 7 8 15 18
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

sort, ***stable_sort***, ***partial_sort_copy***

Summary

Templated algorithm for sorting collections of entities.

Synopsis

```
#include <algorithm>

template <class InputIterator,
          class RandomAccessIterator>
 void partial_sort_copy (InputIterator first,
                         InputIterator last,
                         RandomAccessIterator result_first,
                         RandomAccessIterator result_last);

template <class InputIterator,
          class RandomAccessIterator,
          class Compare>
 void partial_sort_copy (InputIterator first,
                         InputIterator last,
                         RandomAccessIterator result_first,
                         RandomAccessIterator result_last,
                         Compare comp);
```

Description

The ***partial_sort_copy*** algorithm places the smaller of `last - first` and `result_last - result_first` sorted elements from the range `[first, last)` into the range beginning at `result_first`. (i.e., the range: `[result_first, result_first+min(last - first, result_last - result_first))`. Basically, the effect is as if the range `[first,last)` were placed in a temporary buffer, sorted and then as many elements as possible were copied into the range `[result_first, result_last)`.

The first version of the algorithm uses less than (`operator<`) as the comparison operator for the sort. The second version uses the comparison function `comp.`

Complexity

partial_sort_copy does approximately `(last-first) * log(min(last-first, result_last-result_first))` comparisons.

Example

```
//
// partsort.cpp
// #include <vector>
 #include <algorithm>
 #include <iostream.h>

 int main()
 {
   int d1[20] = {17, 3,  5,  -4, 1, 12, -10, -1, 14, 7,
                 -6, 8, 15, -11, 2, -2,  18,  4, -3, 0};
   //
   // Set up a vector.
   //
```

```
    vector<int> v1(d1+0, d1+20);
    //
    // Output original vector.
    //
    cout << "For the vector: ";
    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout," "));
    //
    // Partial sort the first seven elements.
    //
    partial_sort(v1.begin(), v1.begin()+7, v1.end());
    //
    // Output result.
    //
    cout << endl << endl << "A partial_sort of 7 elements gives: "
         << endl << "     ";
    copy(v1.begin(), v1.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl;
    //
    // A vector of ten elements.
    //
    vector<int> v2(10, 0);
    //
    // Sort the last ten elements in v1 into v2.
    //
    partial_sort_copy(v1.begin()+10, v1.end(), v2.begin(),
                      v2.end());
    //
    // Output result.
    //
    cout << endl << "A partial_sort_copy of the last ten elements
                     gives: " << endl << "     ";
    copy(v2.begin(), v2.end(),
         ostream_iterator<int,char>(cout," "));
    cout << endl;
    return 0;
  }
Output :
For the vector: 17 3 5 -4 1 12 -10 -1 14 7 -6 8 15 -11 2 -2 18 4 -
3 0
A partial_sort of seven elements gives:
     -11 -10 -6 -4 -3 -2 -1 17 14 12 7 8 15 5 3 2 18 4 1 0
A partial_sort_copy of the last ten elements gives:
     0 1 2 3 4 5 7 8 15 18
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

sort, ***stable_sort***, ***partial_sort***

Summary

Calculates successive partial sums of a range of values.

Synopsis

```
#include <numeric>

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

template <class InputIterator,
          class OutputIterator,
          class BinaryOperation>
OutputIterator partial_sum (InputIterator first,
                            InputIterator last,
                            OutputIterator result,
                            BinaryOperation binary_op);
```

Description

The ***partial_sum*** algorithm creates a new sequence in which every element is formed by adding all the values of the previous elements, or, in the second form of the algorithm, applying the operation `binary_op` successively on every previous element. That is, ***partial_sum*** assigns to every iterator `i` in the range `[result, result + (last - first))` a value equal to:

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

or, in the second version of the algorithm:

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

For instance, applying ***partial_sum*** to (1,2,3,4,) will yield (1,3,6,10).

The ***partial_sum*** algorithm returns `result + (last - first)`.

If `result` is equal to `first`, the elements of the new sequence successively replace the elements in the original sequence, effectively turning ***partial_sum*** into an inplace transformation.

Complexity

Exactly `(last - first) - 1` applications of the default `+` operator or `binary_op` are performed.

Example

```
//
// partsum.cpp
//
 #include <numeric>   //for accumulate
 #include <vector>    //for vector
```

```
#include <functional> //for times
#include <iostream.h>

int main()
{
  //Initialize a vector using an array of ints
  int d1[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(d1, d1+10);

  //Create an empty vectors to store results
  vector<int> sums((size_t)10), prods((size_t)10);

  //Compute partial_sums and partial_products
  partial_sum(v.begin(), v.end(), sums.begin());
  partial_sum(v.begin(), v.end(), prods.begin(), times<int>());

  //Output the results
  cout << "For the series: " << endl << "     ";
  copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  cout << "The partial sums: " << endl << "     " ;
  copy(sums.begin(),sums.end(),
       ostream_iterator<int,char>(cout," "));
  cout <<" should each equal (N*N + N)/2" << endl << endl;

  cout << "The partial products: " << endl << "     ";
  copy(prods.begin(),prods.end(),
       ostream_iterator<int,char>(cout," "));
  cout << " should each equal N!" << endl;

  return 0;
}
```

```
Output :
For the series:
  1 2 3 4 5 6 7 8 9 10

The partial sums:
  1 3 6 10 15 21 28 36 45 55  should each equal (N*N + N)/2
The partial products:
  1 2 6 24 120 720 5040 40320 362880 3628800  should each equal N!
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

partition

Algorithm

Summary Places all of the entities that satisfy the given predicate before all of the entities that do not.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator, class Predicate>
BidirectionalIterator
partition (BidirectionalIterator first,
           BidirectionalIterator last,
           Predicate pred);
```

Description The ***partition*** algorithm places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy `pred`. It returns an iterator that is one past the end of the group of elements that satisfy `pred`. In other words, ***partition*** returns `i` such that for any iterator `j` in the range`[first, i)`, `pred(*j) == true`, and, for any iterator `k` in the range `[i, last)`, `pred(*j) == false`.

Note that ***partition*** does not necessarily maintain the relative order of the elements that match and elements that do not match the predicate. Use the algorithm ***stable_partition*** if relative order is important.

Complexity The ***partition*** algorithm does at most `(last - first)/2` swaps, and applies the predicate exactly `last - first` times.

Example

```
//
// prtition.cpp
//
 #include <functional>
 #include <deque>
 #include <algorithm>
 #include <iostream.h>

 //
 // Create a new predicate from unary_function.
 //
 template<class Arg>
 class is_even : public unary_function<Arg, bool>
 {
   public:
   bool operator()(const Arg& arg1) { return (arg1 % 2) == 0; }
 };

 int main ()
 {
```

```
   //
   // Initialize a deque with an array of integers.
   //
   int init[10] = { 1,2,3,4,5,6,7,8,9,10 };
   deque<int> d1(init+0, init+10);
   deque<int> d2(init+0, init+10);
   //
   // Print out the original values.
   //
   cout << "Unpartitioned values: " << "\t\t";
   copy(d1.begin(), d1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   //
   // A partition of the deque according to even/oddness.
   //
   partition(d2.begin(), d2.end(), is_even<int>());
   //
   // Output result of partition.
   //
   cout << "Partitioned values: " << "\t\t";
   copy(d2.begin(), d2.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;
   //
   // A stable partition of the deque according to even/oddness.
   //
   stable_partition(d1.begin(), d1.end(), is_even<int>());
   //
   // Output result of partition.
   //
   cout << "Stable partitioned values: " << "\t";
   copy(d1.begin(), d1.end(),
        ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

```
Output :
Unpartitioned values:         1 2 3 4 5 6 7 8 9 10
Partitioned values:           10 2 8 4 6 5 7 3 9 1
Stable partitioned values:    2 4 6 8 10 1 3 5 7 9
```

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

`deque<int, allocator<int> >`

instead of :

`deque<int>`

See Also ***stable_partition***

permutation

Algorithm

Summary Generate successive permutations of a sequence based on an ordering function.

See the entries for ***next_permutation*** and ***prev_permutation***.

plus

Summary A binary function object that returns the result of adding its first and second arguments.

Synopsis

```
#include <functional>

   template<class T>
   struct plus : public binary_function<T, T, T> ;
```

Description ***plus*** is a binary function object. Its `operator()` returns the result of adding `x` and `y`. You can pass a ***plus*** object to any algorithm that uses a binary function. For example, the ***transform*** algorithm applies a binary operation to corresponding values in two collections and stores the result. ***plus*** would be used in that algorithm in the following manner:

```
  vector<int> vec1;
  vector<int> vec2;
  vector<int> vecResult;
  .
  .
  .
  transform(vec1.begin(), vec1.end(),
            vec2.begin(),
            vecResult.begin(), plus<int>());
```

After this call to ***transform***, `vecResult(n)` will contain `vec1(n)` plus `vec2(n)`.

Interface

```
template<class T>
struct plus : binary_function<T, T, T> {
  typedef typename binary_function<T, T, T>::second_argument_type
                                             second_argument_type;
  typedef typename binary_function<T, T, T>::first_argument_type
                                             first_argument_type;
  typedef typename binary_function<T, T, T>::result_type result_type;
  T operator() (const T&, const T&) const;
};
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***binary_function, function objects***

pointer_to_binary-function

Function Object

Summary A function object which adapts a pointer to a binary function to work where a ***binary_function*** is called for.

Synopsis

```
#include <functional>

template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1, Arg2,
                                                            Result> ;
```

Description

The ***pointer_to_binary_function*** class encapsulates a pointer to a two-argument function. The class provides an `operator()` so that the resulting object serves as a binary function object for that function.

The `ptr_fun` function is overloaded to create instances of a ***pointer_to_binary_function*** when provided with the appropriate pointer to a function.

Interface

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1, Arg2,
                                                            Result> {
 public:
   typedef typename binary_function<Arg1, Arg2,
                                    Result>::second_argument_type
                                             second_argument_type;
   typedef typename binary_function<Arg1, Arg2,
                                    Result>::first_argument_type
                                             first_argument_type;
   typedef typename binary_function<Arg1, Arg2, Result>::result_type
                                                           result_type;
   explicit pointer_to_binary_function (Result (*f)(Arg1, Arg2));
   Result operator() (const Arg1&, const Arg2&) const;
};

template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
  ptr_fun (Result (*x)(Arg1, Arg2));
```

See Also ***binary_function, function_objects, pointer_to_unary_function, ptr_fun***

Summary

A function object class that adapts a *pointer to a function* to work where a ***unary_function*** is called for.

Synopsis

```
#include <functional>

template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result>;
```

Description

The ***pointer_to_unary_function*** class encapsulates a pointer to a single-argument function. The class provides an `operator()` so that the resulting object serves as a function object for that function.

The `ptr_fun` function is overloaded to create instances of ***pointer_to_unary_function*** when provided with the appropriate pointer to a function.

Interface

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {

 public:
   typedef typename unary_function<Arg,Result>::argument_type
                                              argument_type;
   typedef typename unary_function<Arg,Result>::result_type
                                              result_type;
   explicit pointer_to_unary_function (Result (*f)(Arg));
   Result operator() (const Arg&) const;
};

template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
  ptr_fun (Result (*f)(Arg));
```

See Also

function_objects, ***pointer_to_binary_function***, ***ptr_fun, unary_function***

pop_heap

Algorithms

Summary Moves the largest element off the heap.

Synopsis

```
template <class RandomAccessIterator>
  void
  pop_heap(RandomAccessIterator first,
           RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  pop_heap(RandomAccessIterator first,
           RandomAccessIterator last, Compare comp);
```

Description A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** algorithm or a new element added by the ***push_heap*** algorithm, in `O(logN)` time.

These properties make heaps useful as priority queues.

The ***pop_heap*** algorithm uses the less than (`<`) operator as the default comparison. An alternate comparison operator can be specified.

The ***pop_heap*** algorithm can be used as part of an operation to remove the largest element from a heap. It assumes that the range `[first, last)` is a valid heap (i.e., that `first` is the largest element in the heap or the first element based on the alternate comparison operator). It then swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last -1)` back into a heap. You can then access the element in `last` using the vector or deque `back()` member function, or remove the element using the `pop_back` member function. Note that ***pop_heap*** does not actually remove the element from the data structure, you must use another function to do that.

Complexity ***pop_heap*** performs at most `2 * log(last - first)` comparisons.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
```

```
#include <vector>
#include <iostream.h>

int main(void)
{
  int d1[4] = {1,2,3,4};
  int d2[4] = {1,3,2,4};

  // Set up two vectors
  vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

  // Make heaps
  make_heap(v1.begin(),v1.end());
  make_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
  // Note that x, y and z represent the remaining
  // values in the container (other than 4).
  // The definition of the heap and heap operations
  // does not require any particular ordering
  // of these values.

  // Copy both vectors to cout
  ostream_iterator<int,char> out(cout," ");
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // Now let's pop
  pop_heap(v1.begin(),v1.end());
  pop_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (3,x,y,4) and v2 = (3,x,y,4)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // And push
  push_heap(v1.begin(),v1.end());
  push_heap(v2.begin(),v2.end(),less<int>());
  // v1 = (4,x,y,z) and v2 = (4,x,y,z)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
  copy(v2.begin(),v2.end(),out);
  cout << endl;

  // Now sort those heaps
  sort_heap(v1.begin(),v1.end());
  sort_heap(v2.begin(),v2.end(),less<int>());
  // v1 = v2 = (1,2,3,4)

  // Copy both vectors to cout
  copy(v1.begin(),v1.end(),out);
  cout << endl;
```

```
    copy(v2.begin(),v2.end(),out);
    cout << endl;

    return 0;
  }

 Output :
 4 2 3 1
 4 3 2 1
 3 2 1 4
 3 1 2 4
 4 3 1 2
 4 3 2 1
 1 2 3 4
 1 2 3 4
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

make_heap, ***push_heap***, ***sort_heap***

predicate

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

Summary

Generate successive permutations of a sequence based on an ordering function.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator>
bool prev_permutation (BidirectionalIterator first,
                       BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool prev_permutation (BidirectionalIterator first,
                       BidirectionalIterator last, Compare comp);
```

Description

The permutation-generating algorithms (***next_permutation*** and ***prev_permutation***) assume that the set of all permutations of the elements in a sequence is lexicographically sorted with respect to `operator<` or `comp`. So, for example, if a sequence includes the integers 1 2 3, that sequence has six permutations, which, in order from first to last, are: 1 2 3 , 1 3 2, 2 1 3, 2 3 1, 3 1 2, and 3 2 1.

The ***prev_permutation*** algorithm takes a sequence defined by the range `[first, last)` and transforms it into its previous permutation, if possible. If such a permutation does exist, the algorithm completes the transformation and returns `true`. If the permutation does not exist, ***prev_permutation*** returns `false`, and transforms the permutation into its "last" permutation (according to the lexicographical ordering defined by either `operator <`, the default used in the first version of the algorithm, or `comp`, which is user-supplied in the second version of the algorithm.)

For example, if the sequence defined by `[first, last)` contains the integers 1 2 3 (in that order), there is *not* a "previous permutation." Therefore, the algorithm transforms the sequence into its last permutation (3 2 1) and returns `false`.

Complexity

At most `(last - first)/2` swaps are performed.

Example

```
//
// permute.cpp
//
 #include <numeric>    //for accumulate
 #include <vector>        //for vector
 #include <functional> //for less
 #include <iostream.h>
```

```
int main()
{
  //Initialize a vector using an array of ints
  int  a1[] = {0,0,0,0,1,0,0,0,0,0};
  char a2[] = "abcdefghji";

  //Create the initial set and copies for permuting
  vector<int>  m1(a1, a1+10);
  vector<int>  prev_m1((size_t)10), next_m1((size_t)10);
  vector<char> m2(a2, a2+10);
  vector<char> prev_m2((size_t)10), next_m2((size_t)10);

  copy(m1.begin(), m1.end(), prev_m1.begin());
  copy(m1.begin(), m1.end(), next_m1.begin());
  copy(m2.begin(), m2.end(), prev_m2.begin());
  copy(m2.begin(), m2.end(), next_m2.begin());

  //Create permutations
  prev_permutation(prev_m1.begin(),
                   prev_m1.end(),less<int>());
  next_permutation(next_m1.begin(),
                   next_m1.end(),less<int>());
  prev_permutation(prev_m2.begin(),
                   prev_m2.end(),less<int>());
  next_permutation(next_m2.begin(),
                   next_m2.end(),less<int>());
  //Output results
  cout << "Example 1: " << endl << "     ";
  cout << "Original values:      ";
  copy(m1.begin(),m1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << "     ";
  cout << "Previous permutation: ";
  copy(prev_m1.begin(),prev_m1.end(),
       ostream_iterator<int,char>(cout," "));

  cout << endl<< "     ";
  cout << "Next Permutation:     ";
  copy(next_m1.begin(),next_m1.end(),
       ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  cout << "Example 2: " << endl << "     ";
  cout << "Original values: ";
  copy(m2.begin(),m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << "     ";
  cout << "Previous Permutation: ";
  copy(prev_m2.begin(),prev_m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << "     ";

  cout << "Next Permutation:     ";
  copy(next_m2.begin(),next_m2.end(),
       ostream_iterator<char,char>(cout," "));
  cout << endl << endl;
```

```
   return 0;
  }

Output :
Example 1:
     Original values:      0 0 0 0 1 0 0 0 0 0
     Previous permutation: 0 0 0 0 0 1 0 0 0 0
     Next Permutation:     0 0 0 1 0 0 0 0 0 0
Example 2:
     Original values: a b c d e f g h j i
     Previous Permutation: a b c d e f g h i j
     Next Permutation:     a b c d e f g i h j
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

next_permutation

Summary

A container adapter which behaves like a priority queue. Items are popped from the queue are in order with respect to a "priority."

Synopsis

```
#include <queue>

template <class T,
          class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue;
```

Description

priority_queue is a container adaptor which allows a container to act as a priority queue. This means that the item with the highest priority, as determined by either the default comparison operator (`operator <`) or the comparison `Compare`, is brought to the front of the queue whenever anything is pushed onto or popped off the queue.

priority_queue adapts any container that provides `front()`, `push_back()` and `pop_back()`. In particular, ***deque*** and ***vector*** can be used.

Interface

```
template <class T,
          class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
 class priority_queue {
public:

// typedefs
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef typename allocator_type allocator_type;

//  Construct
   explicit priority_queue (const Compare& = Compare(),
                            const allocator_type&=allocator_type());
   template <class InputIterator>
     priority_queue (InputIterator first,
                     InputIterator last,
                     const Compare& = Compare(),
                     const allocator_type& = allocator_type());
   allocator_type get_allocator() const;
   bool empty () const;
   size_type size () const;
   const value_type& top () const;
   void push (const value_type&);
   void pop();
};
```

Constructors

```
explicit priority_queue (const Compare& x = Compare(),
                         const allocator_type& alloc = allocator_type());
```

Default constructor. Constructs a priority queue that uses `Container` for its underlying implementation, `x` as its standard for determining priority, and the allocator `alloc` for all storage management.

```
template <class InputIterator>
priority_queue (InputIterator first, InputIterator last,
                const Compare& x = Compare(),
                const allocator_type& alloc = allocator_type());
```

Constructs a new priority queue and places into it every entity in the range `[first, last)`. The priority_queue will use x for determining the priority, and the allocator alloc for all storage management.

Allocator

```
allocator_type get_allocator () const;
```

Returns a copy of the allocator used by self for storage management.

Member Functions

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

Returns `true` if the priority_queue is empty, fa`l`se otherwise.

```
void
pop();
```

Removes the item with the highest priority from the queue.

```
void
push (const value_type& x);
```

Adds `x` to the queue.

```
size_type
size () const;
```

Returns the number of elements in the priority_queue.

```
const value_type&
top () const;
```

Returns a constant reference to the element in the queue with the highest priority.

Example

```
//
// p_queue.cpp
//
 #include <queue>
 #include <deque>
 #include <vector>
 #include <string>
 #include <iostream.h>

 int main(void)
 {
   // Make a priority queue of int using a vector container
   priority_queue<int, vector<int>, less<int> > pq;
```

```
    // Push a couple of values
    pq.push(1);
    pq.push(2);

    // Pop a couple of values and examine the ends
    cout << pq.top() << endl;
    pq.pop();
    cout << pq.top() << endl;
    pq.pop();

    // Make a priority queue of strings using a deque container
    priority_queue<string, deque<string>, less<string> >
       pqs;

    // Push on a few strings then pop them back off
    int i;
    for (i = 0; i < 10; i++)
    {
      pqs.push(string(i+1,'a'));
      cout << pqs.top() << endl;
    }
    for (i = 0; i < 10; i++)
    {
      cout << pqs.top() << endl;
      pqs.pop();
    }

    // Make a priority queue of strings using a deque
    // container, and greater as the compare operation
    priority_queue<string,deque<string>, greater<string> > pgqs;

    // Push on a few strings then pop them back off
    for (i = 0; i < 10; i++)
    {
      pgqs.push(string(i+1,'a'));
      cout << pgqs.top() << endl;
    }

    for (i = 0; i < 10; i++)
    {
      cout << pgqs.top() << endl;
      pgqs.pop();
    }

    return 0;
  }

Output :
2
1
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
```

```
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
a
a
a
a
a
a
a
a
a
a
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Warning

If your compiler does not support default template parameters, you must always provide a `Container` template parameter, and a `Compare` template parameter when declaring an instance of ***priority_queue***. For example, you would not be able to write,

```
priority_queue<int> var;
```

Instead, you would have to write,

```
priority_queue<int, vector<int>,
  less<typename vector<int>::value_type> > var;
```

See Also

Containers, ***queue***

ptr_fun

Function Adaptor

Summary

A function that is overloaded to adapt a *pointer to a function* to work where a function is called for.

Synopsis

```
#include <functional>

template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
  ptr_fun (Result (*f)(Arg));

template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
  ptr_fun (Result (*x)(Arg1, Arg2));
```

Description

The ***pointer_to_unary_function*** and ***pointer_to_binary_function*** classes encapsulate pointers to functions and provide an `operator()` so that the resulting object serves as a function object for the function.

The `ptr_fun` function is overloaded to create instances of ***pointer_to_unary_function*** or ***pointer_to_binary_function*** when provided with the appropriate pointer to a function.

Example

```
//
// pnt2fnct.cpp
//
 #include <functional>
 #include <deque>
 #include <vector>
 #include <algorithm>
 #include <iostream.h>

 //Create a function
 int factorial(int x)
 {
   int result = 1;
   for(int i = 2; i <= x; i++)
       result *= i;
   return result;
 }

 int main()
 {
   //Initialize a deque with an array of ints
   int init[7] = {1,2,3,4,5,6,7};
   deque<int> d(init, init+7);

   //Create an empty vector to store the factorials
   vector<int> v((size_t)7);

   //Transform the numbers in the deque to their factorials and
```

```
    //store in the vector
    transform(d.begin(), d.end(), v.begin(), ptr_fun(factorial));

    //Print the results
    cout << "The following numbers: " << endl << "     ";
    copy(d.begin(),d.end(),ostream_iterator<int,char>(cout," "));

    cout << endl << endl;
    cout << "Have the factorials: " << endl << "     ";
    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

    return 0;
  }

Output :
The following numbers:
     1 2 3 4 5 6 7
Have the factorials:
     1 2 6 24 120 720 5040
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

function_objects, ***pointer_to_binary_function***, ***pointer_to_unary_function***

Summary Places a new element into a heap.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
  void
  push_heap(RandomAccessIterator first,
            RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  push_heap(RandomAccessIterator first,
            RandomAccessIterator last, Compare comp);
```

Description A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by the ***pop_heap*** algorithm, or a new element added by the ***push_heap*** algorithm, in `O(logN)` time.

These properties make heaps useful as priority queues.

The ***push_heap*** algorithms uses the less than (`<`) operator as the default comparison. As with all of the heap manipulation algorithms, an alternate comparison function can be specified.

The ***push_heap*** algorithm is used to add a new element to the heap. First, a new element for the heap is added to the end of a range. (For example, you can use the vector or deque member function `push_back()`to add the element to the end of either of those containers.) The ***push_heap*** algorithm assumes that the range `[first, last - 1)` is a valid heap. It then properly positions the element in the location `last - 1` into its proper position in the heap, resulting in a heap over the range `[first, last)`.

Note that the ***push_heap*** algorithm does not place an element into the heap's underlying container. You must user another function to add the element to the end of the container before applying ***push_heap***.

Complexity For ***push_heap*** at most `log(last - first)` comparisons are performed.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
   vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

   // Make heaps
   make_heap(v1.begin(),v1.end());
   make_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
   // Note that x, y and z represent the remaining
   // values in the container (other than 4).
   // The definition of the heap and heap operations
   // does not require any particular ordering
   // of these values.

   // Copy both vectors to cout
   ostream_iterator<int,char> out(cout," ");
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now let's pop
   pop_heap(v1.begin(),v1.end());
   pop_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (3,x,y,4) and v2 = (3,x,y,4)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // And push
   push_heap(v1.begin(),v1.end());
   push_heap(v2.begin(),v2.end(),less<int>());
   // v1 = (4,x,y,z) and v2 = (4,x,y,z)

   // Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   // Now sort those heaps
   sort_heap(v1.begin(),v1.end());
   sort_heap(v2.begin(),v2.end(),less<int>());
   // v1 = v2 = (1,2,3,4)
```

```
// Copy both vectors to cout
   copy(v1.begin(),v1.end(),out);
   cout << endl;
   copy(v2.begin(),v2.end(),out);
   cout << endl;

   return 0;
 }

Output :
4 2 3 1
4 3 2 1
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

make_heap, ***pop_heap***, ***sort_heap***

Summary

A container adaptor that behaves like a queue (first in, first out).

Synopsis

```
#include <queue>

template <class T, class Container = deque<T> > class queue ;
```

Description

The ***queue*** container adaptor lets a container function as a queue. In a queue, items are pushed into the back of the container and removed from the front. The first items pushed into the queue are the first items to be popped off of the queue (first in, first out, or "FIFO").

queue can adapt any container that supports the `front()`, `back()`, `push_back()` and `pop_front()` operations. In particular, ***deque*** and ***list*** can be used.

Interface

```
template <class T, class Container = deque<T> >
 class queue {

public:

// typedefs

   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef typename Container::allocator_type allocator_type;

// Construct/Copy/Destroy
   explicit queue (const allocator_type& = allocator_type());
   allocator_type get_allocator () const;

// Accessors

   bool empty () const;
   size_type size () const;
   value_type& front ();
   const value_type& front () const;
   value_type& back ();
   const value_type& back () const;
   void push (const value_type&);
   void pop ();
};


// Non-member Operators

template <class T, class Container>
 bool operator== (const queue<T, Container>&,
                  const queue<T, Container>&);
```

```
template <class T, class Container>
 bool operator!= (const queue<T, Container>&,
                  const queue<T, Container>&);

template <class T, class Container>
 bool operator< (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator> (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator<= (const queue<T, Container>&,
                 const queue<T, Container>&);

template <class T, class Container>
 bool operator>= (const queue<T, Container>&,
                 const queue<T, Container>&);
```

Constructors

`explicit` **`queue`** `(const allocator_type& alloc= allocator_type());`

Creates a queue of zero elements. The queue will use the allocator `alloc` for all storage management.

Allocator

`allocator_type` **`get_allocator`** `() const;`

Returns a copy of the allocator used by self for storage management.

Member Functions

`value_type&`
`back` `();`

Returns a reference to the item at the back of the queue (the last item pushed into the queue).

`const value_type&`
`back``() const;`

Returns a constant reference to the item at the back of the queue as a `const_value_type`.

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

Returns `true` if the queue is empty, otherwise `false`.

`value_type&`
`front` `();`

Returns a reference to the item at the front of the queue. This will be the first item pushed onto the queue unless `pop()` has been called since then.

`const value_type&`
`front` `() const;`

Returns a constant reference to the item at the front of the queue as a `const_value_type`.

```
void
pop ();
```

Removes the item at the front of the queue.

```
void
push (const value_type& x);
```

Pushes `x` onto the back of the queue.

```
size_type
size () const;
```

Returns the number of elements on the queue.

Non-member Operators

```
template <class T, class Container>
  bool operator== (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T, class Container>
  bool operator!= (const queue<T, Container>& x,
                   const queue<T, Container>& y);
```

Inequality operator. Returns `!(x==y)`.

```
template <class T, class Container>
  bool operator< (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `true` if the queue defined by the elements contained in `x` is lexicographically less than the queue defined by the elements contained in `y`.

```
template <class T, class Container>
  bool operator> (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `y < x`.

```
template <class T, class Container>
  bool operator< (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `!(y < x)`.

```
template <class T, class Container>
  bool operator< (const queue<T, Container>& x,
                  const queue<T, Container>& y);
```

Returns `!(x < y)`.

Example

```
//
// queue.cpp
//
 #include <queue>
 #include <string>
 #include <deque>
 #include <list>
 #include <iostream.h>

 int main(void)
```

```
 {
   // Make a queue using a list container
   queue<int, list<int>> q;

   // Push a couple of values on then pop them off
   q.push(1);
   q.push(2);
   cout << q.front() << endl;
   q.pop();
   cout << q.front() << endl;
   q.pop();

   // Make a queue of strings using a deque container
   queue<string,deque<string>> qs;

   // Push on a few strings then pop them back off
   int i;
   for (i = 0; i < 10; i++)
   {
     qs.push(string(i+1,'a'));
     cout << qs.front() << endl;
   }
   for (i = 0; i < 10; i++)
   {
     cout << qs.front() << endl;
     qs.pop();
   }

   return 0;
 }

Output :
1
2
a
a
a
a
a
a
a
a
a
a
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Warnings

If your compiler does not support default template parameters, you must always provide a `Container` template parameter. For example you would not be able to write:

```
queue<int> var;
```

rather, you would have to write,

```
queue<int, deque<int> > var;
```

See Also ***allocator, Containers, priority_queue***

Summary

An iterator that reads and writes, and provides random access to a container.

Description

For a complete discussion of iterators, see the *Iterators* section of this reference.

Iterators are a generalization of pointers that allow a C++ program to uniformly interact with different data structures. Random access iterators can read and write, and provide random access to the containers they serve. These iterators satisfy the requirements listed below.

Key to Iterator Requirements

The following key pertains to the iterator requirements listed below:

`a` and `b`	values of type `X`
`n`	value of `distance` type
`u, Distance, tmp` and `m`	identifiers
`r`	value of type `X&`
`t`	value of type `T`

Requirements for Random Access Iterators

The following expressions must be valid for random access iterators:

`X u`	`u` might have a singular value
`X()`	`X()` might be singular
`X(a)`	copy constructor, `a == X(a)`.
`X u(a)`	copy constructor, `u == a`
`X u = a`	assignment, `u == a`
`a == b, a != b`	return value convertible to `bool`
`*a`	return value convertible to `T&`
`a->m`	equivalent to `(*a).m`
`++r`	returns `X&`
`r++`	return value convertible to const `X&`

`*r++`	returns `T&`
`--r`	returns `X&`
`r--`	return `value` convertible to `const X&`
`*r--`	returns `T&`
`r += n`	Semantics of `--r` or `++r` `n` times depending on the sign of `n`
`a + n, n + a`	returns type `X`
`r -= n`	returns `X&`, behaves as `r += -n`
`a - n`	returns type `X`
`b - a`	returns `Distance`
`a[n]`	`*(a+n)`, return value convertible to `T`
`a < b`	total ordering relation
`a > b`	total ordering relation opposite to `<`
`a <= b`	`!(a < b)`
`a >= b`	`!(a > b)`

Like forward iterators, random access iterators have the condition that `a == b` implies `*a == *b`.

There are no restrictions on the number of passes an algorithm may make through the structure.

All relational operators return a value convertible to `bool`.

See Also ***Iterators***, ***Forward Iterators***, ***Bidirectional Iterators***

random_shuffle

Algorithm

Summary

Randomly shuffles elements of a collection.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
 void random_shuffle (RandomAccessIterator first,
                      RandomAccessIterator last);

template <class RandomAccessIterator,
          class RandomNumberGenerator>
 void random_shuffle (RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator& rand);
```

Description

The ***random_shuffle*** algorithm shuffles the elements in the range `[first, last)` with uniform distribution. ***random_shuffle*** can take a particular random number generating function object `rand` , where `rand` takes a positive argument `n` of distance `type` of the `RandomAccessIterator` and returns a randomly chosen value between `0` and `n - 1`.

Complexity

In the ***random_shuffle*** algorithm, `(last - first) -1` swaps are done.

Example

```
//
// rndshufl.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>
 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);
   //Print out elements in original (sorted) order
   cout << "Elements before random_shuffle: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //Mix them up with random_shuffle
   random_shuffle(v.begin(), v.end());
   //Print out the mixed up elements
   cout << "Elements after random_shuffle: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
```

```
 }

Output :
Elements before random_shuffle:
     1 2 3 4 5 6 7 8 9 10
Elements after random_shuffle:
     7 9 10 3 2 5 4 8 1 6
```

Warning

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

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

instead of :

```
vector<int>
```

Summary

Enables iterator-based algorithms to store results into uninitialized memory.

Synopsis

```
#include <memory>

template <class OutputIterator, class T>
 class raw_storage_iterator : public output_iterator {

public:
   explicit raw_storage_iterator (OutputIterator);
   raw_storage_iterator<OutputIterator, t>& operator*();
   raw_storage_iterator<OutputIterator, T>&
     operator= (const T&);
   raw_storage_iterator<OutputIterator>& operator++();
   raw_storage_iterator<OutputIterator> operator++ (int);
};
```

Description

Class ***raw_storage_iterator*** enables iterator-based algorithms to store their results in uninitialized memory. The template parameter, `OutputIterator` is required to have its `operator*` return an object for which `operator&` is both defined and returns a pointer to `T`.

Constructor

`raw_storage_iterator (OutputIterator x);`

Initializes the iterator to point to the same value that `x` points to.

Member Operators

`raw_storage_iterator <OutputIterator, T>& operator =(const T& element);`

Constructs an instance of `T`, initialized to the value `element` , at the location pointed to by the iterator.

`raw_storage_iterator <OutputIterator, T>& operator++();`

Pre-increment: advances the iterator and returns a reference to the updated iterator.

`raw_storage_iterator<OutputIterator> operator++ (int);`

Post-increment: advances the iterator and returns the old value of the iterator.

Algorithm

Summary

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Synopsis

```
#include <algorithm>

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

Description

The ***remove*** algorithm eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following condition holds: `*i == value`. ***remove*** returns an iterator that designates the end of the resulting range. ***remove*** is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

remove does not actually reduce the size of the sequence. It actually operates by: 1) copying the values that are to be *retained* to the front of the sequence, and 2) returning an iterator that describes where the sequence of retained values ends. Elements that are after this iterator are simply the original sequence values, left unchanged. Here's a simple example:

Say we want to remove all values of "2" from the following sequence:

354621271

Applying the ***remove*** algorithm results in the following sequence:

3546171 | XX

The vertical bar represents the position of the iterator returned by ***remove***. Note that the elements to the left of the vertical bar are the original sequence with the "2's" removed.

If you want to actually delete items from the container, use the following technique:

```
container.erase(remove(first,last,value),container.end());
```

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>
 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg& x){ return 1; }
 };
 int main ()
 {
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove the 7
   vector<int>::iterator result =
           remove(v.begin(), v.end(), 7);
   // delete dangling elements from the vector
   v.erase(result, v.end());
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   // remove everything beyond the fourth element
   result = remove_if(v.begin()+4,
                      v.begin()+8, all_true<int>());
   // delete dangling elements
   v.erase(result, v.end());
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   return 0;
 }
Output :
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

remove_if, ***remove_copy***, ***remove_copy_if***

Summary

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Synopsis

```
#include <algorithm>

template <class InputIterator,
          class OutputIterator,
          class T>
 OutputIterator remove_copy (InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             const T& value);
```

Description

The ***remove_copy*** algorithm copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding condition does *not* hold: `*i == value`. ***remove_copy*** returns the end of the resulting range. ***remove_copy*** is stable, that is, the relative order of the elements in the resulting range is the same as their relative order in the original range. The elements in the original sequence are not altered by ***remove_copy***.

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
 //
 // remove.cpp
 //
  #include <algorithm>
  #include <vector>
  #include <iterator>
  #include <iostream.h>

  template<class Arg>
  struct all_true : public unary_function<Arg, bool>
  {
    bool operator() (const Arg&) { return 1; }
  };

  int main ()
  {
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    vector<int> v(arr+0, arr+10);

    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
    cout << endl << endl;
    //
    // Remove the 7.
    //
```

```
    vector<int>::iterator result = remove(v.begin(), v.end(), 7);
    //
    // Delete dangling elements from the vector.
    //
    v.erase(result, v.end());

    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
    cout << endl << endl;
    //
    // Remove everything beyond the fourth element.
    //
    result = remove_if(v.begin()+4, v.begin()+8, all_true<int>());
    //
    // Delete dangling elements.
    //
    v.erase(result, v.end());

    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
    cout << endl << endl;
    //
    // Now remove all 3s on output.
    //
    remove_copy(v.begin(), v.end(),
                ostream_iterator<int,char>(cout," "), 3);
    cout << endl << endl;
    //
    // Now remove everything satisfying predicate on output.
    // Should yield a NULL vector.
    //
    remove_copy_if(v.begin(), v.end(),
                   ostream_iterator<int,char>(cout," "),
                   all_true<int>());

    return 0;
  }

Output :
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

remove, ***remove_if***, ***remove_copy_if***

remove_copy_if

Algorithm

Summary Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Synopsis

```
#include <algorithm>

template <class InputIterator,
          class OutputIterator,
          class Predicate>
 OutputIterator remove_copy_if (InputIterator first,
                                InputIterator last,
                                OutputIterator result,
                                Predicate pred);
```

Description The ***remove_copy_if*** algorithm copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following condition does *not* hold: `pred(*i)  == true`. ***remove_copy_if*** returns the end of the resulting range. ***remove_copy_if*** is stable, that is, the relative order of the elements in the resulting range is the same as their relative order in the original range.

Complexity Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator() (const Arg&) { return 1; }
 };

 int main ()
 {
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr+0, arr+10);

   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Remove the 7.
   //
   vector<int>::iterator result = remove(v.begin(), v.end(), 7);
```

```
  //
  // Delete dangling elements from the vector.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Remove everything beyond the fourth element.
  //
  result = remove_if(v.begin()+4, v.begin()+8, all_true<int>());
  //
  // Delete dangling elements.
  //
  v.erase(result, v.end());

  copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  //
  // Now remove all 3s on output.
  //
  remove_copy(v.begin(), v.end(),
              ostream_iterator<int>(cout," "), 3);
  cout << endl << endl;
  //
  // Now remove everything satisfying predicate on output.
  // Should yield a NULL vector.
  //
  remove_copy_if(v.begin(), v.end(),
                 ostream_iterator<int,char>(cout," "),
                 all_true<int>());

  return 0;
}
```

```
Output :
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***remove***, ***remove_if***, ***remove_copy***

remove_if

Algorithm

Summary

Move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if (ForwardIterator first,
                           ForwardIterator last,
                           Predicate pred);
```

Description

The ***remove_if*** algorithm eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding condition holds: `pred(*i) == true`. ***remove_if*** returns the end of the resulting range. ***remove_if*** is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

remove_if does not actually reduce the size of the sequence. It actually operates by: 1) copying the values that are to be *retained* to the front of the sequence, and 2) returning an iterator that describes where the sequence of retained values ends. Elements that are after this iterator are simply the original sequence values, left unchanged. Here's a simple example:

Say we want to remove all even numbers from the following sequence:

123456789

Applying the ***remove_if*** algorithm results in the following sequence:

13579 | XXXX

The vertical bar represents the position of the iterator returned by ***remove_if***. Note that the elements to the left of the vertical bar are the original sequence with the even numbers removed. The elements to the right of the bar are simply the untouched original members of the original sequence.

If you want to actually delete items from the container, use the following technique:

```
container.erase(remove(first,last,value),container.end());
```

Complexity

Exactly `last1 - first1` applications of the corresponding predicate are done.

Example

```
//
// remove.cpp
//
```

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

template<class Arg>
struct all_true : public unary_function<Arg, bool>
{
  bool operator()(const Arg& x){ return 1; }
};

int main ()
{
  int arr[10] = {1,2,3,4,5,6,7,8,9,10};
  vector<int> v(arr, arr+10);

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

  // remove the 7
  vector<int>::iterator result =
          remove(v.begin(), v.end(), 7);
  // delete dangling elements from the vector
  v.erase(result, v.end());

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

  // remove everything beyond the fourth element
  result = remove_if(v.begin()+4,
                     v.begin()+8, all_true<int>());
  // delete dangling elements
  v.erase(result, v.end());

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

  return 0;
}
Output :
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 8 9 10
1 2 3 4
1 2 4
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***remove***, ***remove_copy***, ***remove_copy_if***

Summary

Substitutes elements stored in a collection with new values.

Synopsis

```
#include <algorithm>

template <class ForwardIterator, class T>
void replace (ForwardIterator first,
              ForwardIterator last,
              const T& old_value,
              const T& new_value);
```

Description

The ***replace*** algorithm replaces elements referred to by iterator `i` in the range `[first, last)` with `new_value` when the following condition holds: `*i == old_value`

Complexity

Exactly `last - first` comparisons or applications of the corresponding predicate are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg&){ return 1; }
 };

 int main()
 {

   //Initialize a vector with an array of integers
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out original vector
   cout << "The original list: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Replace the number 7 with 11
   replace(v.begin(), v.end(), 7, 11);
```

```
    // Print out vector with 7 replaced,
    // s.b. 1 2 3 4 5 6 11 8 9 10
    cout << "List after replace " << endl << "     ";
    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
    cout << endl << endl;

    //Replace 1 2 3 with 13 13 13
    replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);

    // Print out the remaining vector,
    // s.b. 13 13 13 4 5 6 11 8 9 10
    cout << "List after replace_if " << endl << "     ";
    copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
    cout << endl << endl;

    return 0;
  }

Output :
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

replace_if, ***replace_copy***, ***replace_copy_if***

replace_copy

Summary Substitutes elements stored in a collection with new values.

Synopsis

```
#include <algorithm>

template <class InputIterator,
          class OutputIterator,
          class T>
OutputIterator replace_copy (InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             const T& old_value,
                             const T& new_value);
```

Description The ***replace_copy*** algorithm leaves the original sequence intact and places the revised sequence into `result`. The algorithm compares elements referred to by iterator `i` in the range `[first, last)` with `old_value`. If `*i` does not equal `old_value`, then the ***replace_copy*** copies `*i` to `result+(first-i)`. If `*i==old_value`, then ***replace_copy*** copies `new_value` to `result+(first-i)`. ***replace_copy*** returns `result+(last-first)`.

Complexity Exactly `last - first` comparisons between values are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator() (const Arg&) { return 1; }
 };

 int main ()
 {
   //
   // Initialize a vector with an array of integers.
   //
   int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
   vector<int> v(arr+0, arr+10);
   //
   // Print out original vector.
```

```
   //
   cout << "The original list: " << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace the number 7 with 11.
   //
   replace(v.begin(), v.end(), 7, 11);
   //
   // Print out vector with 7 replaced.
   //
   cout << "List after replace:" << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace 1 2 3 with 13 13 13.
   //
   replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);
   //
   // Print out the remaining vector.
   //
   cout << "List after replace_if:" << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace those 13s with 17s on output.
   //
   cout << "List using replace_copy to cout:" << endl << "     ";
   replace_copy(v.begin(), v.end(),
                ostream_iterator<int,char>(cout, " "), 13, 17);
   cout << endl << endl;
   //
   // A simple example of replace_copy_if.
   //
   cout << "List w/ all elements output as 19s:" << endl << "   ";
   replace_copy_if(v.begin(), v.end(),
                   ostream_iterator<int,char>(cout, " "),
                   all_true<int>(), 19);
   cout << endl;

   return 0;
 }
Output :
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

Warning

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

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

instead of :

```
vector<int>
```

See Also

replace, ***replace_if***, ***replace_copy_if***

replace_copy_if

Summary

Substitutes elements stored in a collection with new values.

Synopsis

```
#include <algorithm>
template <class InputIterator,
          class OutputIterator,
          class Predicate,
          class T>
OutputIterator replace_copy_if (InputIterator first,
                                InputIterator last,
                                OutputIterator result,
                                Predicate pred,
                                const T& new_value);
```

Description

The ***replace_copy_if*** algorithm leaves the original sequence intact and places a revised sequence into `result`. The algorithm compares each element `*i` in the range `[first, last)` with the conditions specified by `pred`. If `pred(*i)==false`, ***replace_copy_if*** copies `*i` to `result+(first-i)`. If `pred(*i)==true`, then ***replace_copy*** copies `new_value` to `result+(first-i)`. ***replace_copy_if*** returns `result+(last-first)`.

Complexity

Exactly `last - first` applications of the predicate are performed.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator() (const Arg&) { return 1; }
 };

 int main ()
 {
   //
   // Initialize a vector with an array of integers.
   //
   int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
   vector<int> v(arr+0, arr+10);
   //
   // Print out original vector.
```

```
   //
   cout << "The original list: " << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace the number 7 with 11.
   //
   replace(v.begin(), v.end(), 7, 11);
   //
   // Print out vector with 7 replaced.
   //
   cout << "List after replace:" << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace 1 2 3 with 13 13 13.
   //
   replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);
   //
   // Print out the remaining vector.
   //
   cout << "List after replace_if:" << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Replace those 13s with 17s on output.
   //
   cout << "List using replace_copy to cout:" << endl << "     ";
   replace_copy(v.begin(), v.end(),
                ostream_iterator<int,char>(cout, " "), 13, 17);
   cout << endl << endl;
   //
   // A simple example of replace_copy_if.
   //
   cout << "List w/ all elements output as 19s:" << endl << "   ";
   replace_copy_if(v.begin(), v.end(),
                   ostream_iterator<int,char>(cout, " "),
                   all_true<int>(), 19);
   cout << endl;

   return 0;
 }
Output :
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

Warning

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

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

instead of :

```
vector<int>
```

See Also

replace, ***replace_if***, ***replace_copy***

replace_if

Algorithm

Summary Substitutes elements stored in a collection with new values.

Synopsis

```
#include <algorithm>

template <class ForwardIterator,
          class Predicate,
          class T>
void replace_if (ForwardIterator first,
                 ForwardIterator last,
                 Predicate pred
                 const T& new_value);
```

Description The ***replace_if*** algorithm replaces element referred to by iterator `i` in the range `[first, last)` with `new_value` when the following condition holds: `pred(*i) == true`.

Complexity Exactly `last - first` applications of the predicate are done.

Example

```
//
// replace.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iterator>
 #include <iostream.h>

 template<class Arg>
 struct all_true : public unary_function<Arg, bool>
 {
   bool operator()(const Arg&){ return 1; }
 };

 int main()
 {

   //Initialize a vector with an array of integers
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out original vector
   cout << "The original list: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Replace the number 7 with 11
   replace(v.begin(), v.end(), 7, 11);
```

```
   // Print out vector with 7 replaced,
   // s.b. 1 2 3 4 5 6 11 8 9 10
   cout << "List after replace " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Replace 1 2 3 with 13 13 13
   replace_if(v.begin(), v.begin()+3, all_true<int>(), 13);

   // Print out the remaining vector,
   // s.b. 13 13 13 4 5 6 11 8 9 10
   cout << "List after replace_if " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   return 0;
 }

Output :
The original list:
     1 2 3 4 5 6 7 8 9 10
List after replace:
     1 2 3 4 5 6 11 8 9 10
List after replace_if:
     13 13 13 4 5 6 11 8 9 10
List using replace_copy to cout:
     17 17 17 4 5 6 11 8 9 10
List with all elements output as 19s:
     19 19 19 19 19 19 19 19 19 19
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***replace***, ***replace_copy***, ***replace_copy_if***

Summary Pointer based primitive for handling memory

Synopsis

```
#include <memory>

template <class T>
void return_temporary_buffer (T* p, T*);
```

Description The ***return_temporary_buffer*** templated function returns a buffer, previously allocated through ***get_temporary_buffer***, to available memory. Parameter p points to the buffer.

See Also ***allocate***, ***deallocate***, ***construct***, ***get_temporary_buffer***

Summary

Reverse the order of elements in a collection.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator>
void reverse (BidirectionalIterator first,
              BidirectionalIterator last);
```

Description

The algorithm ***reverse*** reverses the elements in a sequence so that the last element becomes the new first element, and the first element becomes the new last. For each non-negative integer `i <= (last - first)/2`, ***reverse*** applies ***swap*** to all pairs of iterators `first + I, (last - I) - 1`.

Because the iterators are assumed to be bidirectional, ***reverse*** does not return anything.

Complexity

reverse performs exactly `(last - first)/2` swaps.

Example

```
//
// reverse.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>
 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out elements in original (sorted) order
   cout << "Elements before reverse: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Reverse the ordering
   reverse(v.begin(), v.end());

   //Print out the reversed elements
   cout << "Elements after reverse: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }
```

```
Output :
Elements before reverse:
     1 2 3 4 5 6 7 8 9 10
Elements after reverse:
     10 9 8 7 6 5 4 3 2 1
A reverse_copy to cout:
     1 2 3 4 5 6 7 8 9 10
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

reverse_copy, ***swap***

Summary An iterator that traverses a collection backwards.

Synopsis

```
#include <iterator>

template <class BidirectionalIterator,
          class T,
          class Reference = T&,
          class Pointer = T*
          class Distance = ptrdiff_t>
class reverse_bidirectional_iterator : public
      iterator<bidirectional_iterator_tag,T, Distance> ;

template <class RandomAccessIterator,
          class T,
          class Reference = T&,
          class Pointer = T*,
          class Distance = ptrdiff_t>
class reverse_iterator : public
       iterator<random_access_iterator_tag,T,Distance>;
```

Description

The iterators ***reverse_iterator*** and ***reverse_bidirectional_iterator*** correspond to ***random_access_iterator*** and ***bidirectional_iterator***, except they traverse the collection they point to in the opposite direction. The fundamental relationship between a reverse iterator and its corresponding iterator `i` is established by the identity:

```
&*(reverse_iterator(i)) == &*(i-1);
```

This mapping is dictated by the fact that, while there is always a pointer past the end of a container, there might not be a valid pointer before its beginning.

The following are true for ***reverse_bidirectional_iterators*** :

- These iterators may be instantiated with the default constructor or by a single argument constructor that initializes the new `reverse_bidirectional_iterator` with a `bidirectional_iterator`.
- `operator*` returns a reference to the current value pointed to.
- `operator++` advances the iterator to the previous item (`--current`) and returns a reference to `*this`.
- `operator++(int)` advances the iterator to the previous item (`--current`) and returns the old value of `*this`.

- `operator--` advances the iterator to the following item (`++current`) and returns a reference to `*this`.
- `operator--(int)` Advances the iterator to the following item (`++current)` and returns the old value of `*this.`
- `operator==` This non-member operator returns `true` if the iterators `x` and `y` point to the same item.

The following are true for ***reverse_iterator***s :

- These iterators may be instantiated with the default constructor or by a single argument constructor which initializes the new `reverse_iterator` with a `random_access_iterator`.
- `operator*` returns a reference to the current value pointed to.
- `operator++` advances the iterator to the previous item (`--current`) and returns a reference to `*this`.
- `operator++(int)` advances the iterator to the previous item (`--current`) and returns the old value of `*this`.
- `operator--` advances the iterator to the following item (`++current`) and returns a reference to `*this`.
- `operator--(int)` advances the iterator to the following item (`++current`) and returns the old value of `*this`.
- `operator==` is a non-member operator returns `true` if the iterators `x` and `y` point to the same item.
- `operator!=` is a non-member operator returns `!(x==y).`
- `operator<` is a non-member operator returns `true` if the iterator `x` precedes the iterator `y`.
- `operator>` is a non-member operator returns `y < x`.
- `operator<=` is a non-member operator returns `!(y < x)`.
- `operator>=` is a non-member operator returns `!(x < y)`.
- The remaining operators (`<, +, - , +=, -=`) are redefined to behave exactly as they would in a `random_access_iterator`, except with the sense of direction reversed.

Complexity

All iterator operations are required to take at most amortized constant time.

Interface

```
template <class BidirectionalIterator,
          class T,
          class Reference = T&,
          class Pointer = T*,
          class Distance = ptrdiff_t>
class reverse_bidirectional_iterator
   : public iterator<bidirectional_iterator_tag,T, Distance> {
    typedef reverse_bidirectional_iterator<BidirectionalIterator, T,
                                          Reference,
                                          Pointer, Distance> self;
    friend bool operator== (const self&, const self&);
  public:
    reverse_bidirectional_iterator ();
    explicit reverse_bidirectional_iterator
      (BidirectionalIterator);
    BidirectionalIterator base ();
    Reference operator* ();
    self& operator++ ();
    self operator++ (int);
    self& operator-- ();
    self operator-- (int);
  };

// Non-member Operators

  template <class BidirectionalIterator,
            class T, class Reference,
            class Pointer, class Distance>
  bool operator== (
     const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Pointer,Distance>&,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Pointer,Distance>&);

template <class BidirectionalIterator,
            class T, class Reference,
            class Pointer, class Distance>
  bool operator!= (
     const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Pointer,Distance>&,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Pointer,Distance>&);


template <class RandomAccessIterator,
          class T,
          class Reference = T&,
          class Pointer = T*,
          class Distance = ptrdiff_t>
class reverse_iterator
   : public iterator<random_access_iterator_tag,T,Distance> {
```

```
    typedef reverse_iterator<RandomAccessIterator, T, Reference,
                             Pointer, Distance> self;

    friend bool operator==    (const self&, const self&);
    friend bool operator<     (const self&, const self&);
    friend Distance operator- (const self&, const self&);
    friend self operator+     (Distance, const self&);

public:
    reverse_iterator ();
    explicit reverse_iterator (RandomAccessIterator);
    RandomAccessIterator base ();
    Reference operator* ();
    self& operator++ ();
    self operator++ (int);
    self& operator-- ();
    self operator-- (int);

    self  operator+ (Distance) const;
    self& operator+= (Distance);
    self operator- (Distance) const;
    self& operator-= (Distance);
    Reference operator[] (Distance);
};

// Non-member Operators

    template <class RandomAccessIterator, class T,
              class Reference, class Pointer,
              class Distance> bool operator== (
        const reverse_iterator<RandomAccessIterator, T,
                              Reference, Pointer,
                              Distance>&,
        const reverse_iterator<RandomAccessIterator, T,
                              Reference, Pointer,
                              Distance>&);

template <class RandomAccessIterator, class T,
              class Reference, class Pointer,
              class Distance> bool operator!= (
        const reverse_iterator<RandomAccessIterator, T,
                              Reference, Pointer,
                              Distance>&,
        const reverse_iterator<RandomAccessIterator, T,
                              Reference, Pointer,
                              Distance>&);

     template <class RandomAccessIterator, class T,
               class Reference, class Pointer,
               class Distance> bool operator< (
         const reverse_iterator<RandomAccessIterator, T,
                               Reference, Pointer,
                               Distance>&,
         const reverse_iterator<RandomAccessIterator, T,
                               Reference, Pointer,
                               Distance>&);

template <class RandomAccessIterator, class T,
```

```
                class Reference, class Pointer,
                class Distance> bool operator> (
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&,
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&);

template <class RandomAccessIterator, class T,
                class Reference, class Pointer,
                class Distance> bool operator<= (
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&,
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&);

template <class RandomAccessIterator, class T,
                class Reference, class Pointer,
                class Distance> bool operator>= (
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&,
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&);

     template <class RandomAccessIterator, class T,
                class Reference, class Pointer,
                class Distance> Distance operator- (
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&,
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&);

     template <class RandomAccessIterator, class T,
                class Reference, class Pointer,
                class Distance>
      reverse_iterator<RandomAccessIterator, T,
                        Reference, Pointer,
                        Distance> operator+ (
          Distance,
          const reverse_iterator<RandomAccessIterator, T,
                                 Reference, Pointer,
                                 Distance>&);
```

Example

```
//
// rev_itr.cpp
//
 #include <iterator>
 #include <vector>
 #include <iostream.h>
```

```
 int main()
 {
   //Initialize a vector using an array
   int arr[4] = {3,4,7,8};
   vector<int> v(arr,arr+4);
   //Output the original vector
   cout << "Traversing vector with iterator: " << endl << "     ";
   for(vector<int>::iterator i = v.begin(); i != v.end(); i++)
     cout << *i << " ";
   //Declare the reverse_iterator
   vector<int>::reverse_iterator rev(v.end());
   vector<int>::reverse_iterator rev_end(v.begin());
   //Output the vector backwards
   cout << endl << endl;
   cout << "Same vector, same loop, reverse_itertor: " << endl
        << "     ";
   for(; rev != rev_end; rev++)
     cout << *rev << " ";
   return 0;
 }

Output :
Traversing vector with iterator:
     3 4 7 8
Same vector, same loop, reverse_itertor:
     8 7 4 3
```

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***Iterators***

reverse_copy

Summary

Reverse the order of elements in a collection while copying them to a new collecton.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy (BidirectionalIterator first,
                             BidirectionalIterator last,
                             OutputIterator result);
```

Description

The ***reverse_copy*** algorithm copies the range `[first, last)` to the range `[result, result + (last - first))` such that for any non- negative integer `i < (last - first)`, the following assignment takes place:

```
*(result + (last - first) -i) = *(first + i)
```

reverse_copy returns `result + (last - first)`. The ranges `[first, last)` and `[result, result + (last - first))` must not overlap.

Complexity

reverse_copy performs exactly `(last - first)` assignments.

Example

```
//
// reverse.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main ()
 {
   //
   // Initialize a vector with an array of integers.
   //
   int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
   vector<int> v(arr+0, arr+10);
   //
   // Print out elements in original (sorted) order.
   //
   cout << "Elements before reverse: " << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;
   //
   // Reverse the ordering.
   //
   reverse(v.begin(), v.end());
   //
   // Print out the reversed elements.
```

```
   //
   cout << "Elements after reverse: " << endl << "     ";
   copy(v.begin(), v.end(), ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   cout << "A reverse_copy to cout: " << endl << "     ";
   reverse_copy(v.begin(), v.end(),
                ostream_iterator<int,char>(cout, " "));
   cout << endl;

   return 0;
 }

Output :
Elements before reverse:
     1 2 3 4 5 6 7 8 9 10
Elements after reverse:
     10 9 8 7 6 5 4 3 2 1
A reverse_copy to cout:
     1 2 3 4 5 6 7 8 9 10
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

reverse

reverse_iterator

See the ***reverse_bidirectional_iterator*** section of this reference.

rotate, rotate_copy

Algorithm

Summary

Left rotates the order of items in a collection, placing the first item at the end, second item first, etc., until the item pointed to by a specified iterator is the first item in the collection.

Synopsis

```
#include <algorithm>

template <class ForwardIterator>
void rotate (ForwardIterator first,
             ForwardIterator middle,
             ForwardIterator last);

template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy (ForwardIterator first,
                            ForwardIterator middle,
                            ForwardIterator last,
                            OutputIterator result);
```

Description

The ***rotate*** algorithm takes three iterator arguments, `first`, which defines the start of a sequence, `last`, which defines the end of the sequence, and `middle` which defines a point within the sequence. ***rotate*** "swaps" the segment that contains elements from `first` through `middle-1` with the segment that contains the elements from `middle` through `last`. After ***rotate*** has been applied, the element that was in position `middle`, is in position `first`, and the other elements in that segment are in the same order relative to each other. Similarly, the element that was in position `first` is now in position `last-middle +1`. An example will illustrate how ***rotate*** works:

Say that we have the sequence:

```
2 4 6 8 1 3 5
```

If we call ***rotate*** with `middle = 5`, the two segments are

```
2 4 6  8     and     1 3 5
```

After we apply rotate, the new sequence will be:

```
1 3 5 2 4 6 8
```

Note that the element that was in the fifth position is now in the first position, and the element that was in the first position is in position 4 (`last - first + 1`, or 8 - 5 +1 =4).

The formal description of this algorithms is: for each non-negative integer `i < (last - first)`, ***rotate*** places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`. `[first, middle)` and `[middle, last)` are valid ranges.

rotate_copy rotates the elements as described above, but instead of swapping elements within the same sequence, it copies the result of the rotation to a container specified by `result`. ***rotate_copy*** copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non- negative integer `i < (last - first)` the following assignment takes place:

```
*(result + (i + (last - middle)) % (last -first)) = *(first + i).
```

The ranges `[first,  last)` and `[result, result, + (last - first))` may not overlap.

Complexity

For ***rotate*** at most `last - first` swaps are performed.

For ***rotate_copy*** `last - first` assignments are performed.

Example

```
//
// rotate
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main()
 {
   //Initialize a vector with an array of ints
   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> v(arr, arr+10);

   //Print out elements in original (sorted) order
   cout << "Elements before rotate: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl << endl;

   //Rotate the elements
   rotate(v.begin(), v.begin()+4, v.end());

   //Print out the rotated elements
   cout << "Elements after rotate: " << endl << "     ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));
   cout << endl;

   return 0;
 }

Output :
Elements before rotate:
     1 2 3 4 5 6 7 8 9 10
Elements after rotate:
     5 6 7 8 9 10 1 2 3 4
```

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

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

instead of :

```
vector<int>
```

Summary Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range.

Synopsis

```
#include <algorithm>

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

template <class ForwardIterator1,
          class ForwardIterator2,
          class BinaryPredicate>
 ForwardIterator1 search (ForwardIterator1 first1,
                          ForwardIterator1 last1,
                          ForwardIterator2 first2,
                          ForwardIterator2 last2,
                          BinaryPredicate binary_pred);

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

template <class ForwardIterator,
          class Size,
          class T,
          class BinaryPredicate>
ForwardIterator search_n (ForwardIterator first,
                          ForwardIterator last,
                          Size count, const T& value,
                          BinaryPredicate pred)
```

Description The ***search*** and ***search_n*** are used for searching for a sub-sequence within a sequence. The ***search*** algorithm searches for a sub-sequence `[first2, last2)` within a sequence `[first1, last1)`, and returns the beginning location of the sub-sequence. If it does not find the sub-sequence, ***search*** returns `last1`. The first version of ***search*** uses the equality (`==`) operator as a default, and the second version allows you to specify a binary predicate to perform the comparison.

The ***search_n*** algorithm searches for the sub-sequence composed of `count` occurrences of `value` within a sequence `[first, last)`, and returns `first` if this sub-sequence is found. If it does not find the sub-sequence, ***search_n*** returns `last`. The first version of ***search_n*** uses the equality (`==`) operator as a default, and the second version allows you to specify a binary predicate to perform the comparison.

Complexity

search performs at most `(last1 - first1)*(last2-first2)` applications of the corresponding predicate.

search_n performs at most `(last - first)` applications of the corresponding predicate.

Example

```
//
// search.cpp
//
 #include <algorithm>
 #include <list>
 #include <iostream.h>

 int main()
 {
   // Initialize a list sequence and
   // sub-sequence with characters
   char seq[40] = "Here's a string with a substring in it";
   char subseq[10] = "substring";
   list<char> sequence(seq, seq+39);
   list<char> subseqnc(subseq, subseq+9);

   //Print out the original sequence
   cout << endl << "The sub-sequence, " << subseq
        << ", was found at the ";
   cout << endl << "location identified by a '*'"
        << endl << "     ";

   // Create an iterator to identify the location of
   // sub-sequence within sequence
   list<char>::iterator place;

   //Do search
   place = search(sequence.begin(), sequence.end(),
                  subseqnc.begin(), subseqnc.end());

   //Identify result by marking first character with a '*'
   *place = '*';

   //Output sequence to display result
   for(list<char>::iterator i = sequence.begin();
           i != sequence.end(); i++)
     cout << *i;
   cout << endl;

   return 0;
 }

Output :
The sub-sequence, substring, was found at the
location identified by a '*'
     Here's a string with a *substring in it
```

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

```
list<char, allocator<char> >
```

instead of :

```
list<char>
```

Sequence

Summary

A ***sequence*** is a container that organizes a set of objects, all the same type, into a linear arrangement. ***vector***, ***list***, ***deque***, and ***string*** fall into this category.

Sequences offer different complexity trade-offs. ***vector*** offers fast inserts and deletes from the end of the container. ***deque*** is useful when insertions and deletions will take place at the beginning or end of the sequence. Use ***list*** when there are frequent insertions and deletions from the middle of the sequence.

See Also

For more information about sequences and their requirements, see the ***Containers*** section of this reference guide, or see the section on the specific container.

Summary An associative container that supports unique keys. A ***set*** supports bidirectional iterators.

Synopsis

```
#include <set>

template <class Key, class Compare = less<Key>,
  class Allocator = allocator<Key> >
class set ;
```

Description ***set<Key, Compare, Allocator>*** is an associative container that supports unique keys and provides for fast retrieval of the keys. A ***set*** contains at most one of any key value. The keys are sorted using `Compare`.

Since a ***set*** maintains a total order on its elements, you cannot alter the key values directly. Instead, you must insert new elements with an `insert_iterator`.

Any type used for the template parameter `Key` must provide the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
Copy constructors     T(t) and T(u)
Destructor            t.~T()
Address of            &t and &u yielding T* and
                      const T* respectively
Assignment            t = a where a is a
                     (possibly const) value of T
```

The `type` used for the `Compare` template parameter must satisfy the requirements for binary functions.

Interface

```
template <class Key, class Compare = less<Key>,
         class Allocator = allocator<Key> >
 class set {

public:

 // types

   typedef Key key_type;
   typedef Key value_type;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef Allocator allocator_type;
   typename reference;
   typename const_reference;
```

```
  typename iterator;
  typename const_iterator;
  typename size_type;
  typename difference_type;
  typename reverse_iterator;
  typename const_reverse_iterator;

// Construct/Copy/Destroy

  explicit set (const Compare& = Compare(),
                const Allocator& = Allocator ());
  template <class InputIterator>
   set (InputIterator, InputIterator, const Compare& = Compare(),
        const Allocator& = Allocator ());
  set (const set<Key, Compare, Allocator>&);
  ~set ();
  set<Key, Compare, Allocator>& operator= (const set <Key, Compare,
                                                     Allocator>&);
  allocator_type get_allocator () const;

// Iterators

  iterator begin ();
  const_iterator begin () const;
  iterator end ();
  const_iterator end () const;
  reverse_iterator rbegin ();
  const_reverse_iterator rbegin () const;
  reverse_iterator rend ();
  const_reverse_iterator rend () const;

// Capacity

  bool empty () const;
  size_type size () const;
  size_type max_size () const;

// Modifiers

  pair<iterator, bool> insert (const value_type&);
  iterator insert (iterator, const value_type&);
  template <class InputIterator>
   void insert (InputIterator, InputIterator);
  iterator erase (iterator);
  size_type erase (const key_type&);
  iterator erase (iterator, iterator);
  void swap (set<Key, Compare, Allocator>&);
  void clear ();

// Observers

  key_compare key_comp () const;
  value_compare value_comp () const;

// Set operations

  size_type count (const key_type&) const;
  pair<iterator, iterator> equal_range (const  key_type&) const;
```

```
   iterator find (const key_type&) const;
   iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&) const

};

 // Non-member Operators

template <class Key, class Compare, class Allocator>
 bool operator== (const set<Key, Compare, Allocator>&,
                  const set<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator!= (const set<Key, Compare, Allocator>&,
                  const set<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator< (const set<Key, Compare, Allocator>&,
                 const set<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator> (const set<Key, Compare, Allocator>&,
                 const set<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator<= (const set<Key, Compare, Allocator>&,
                 const set<Key, Compare, Allocator>&);

template <class Key, class Compare, class Allocator>
 bool operator>= (const set<Key, Compare, Allocator>&,
                 const set<Key, Compare, Allocator>&);


// Specialized Algorithms

template <class Key, class Compare, class Allocator>
void swap (set <Key, Compare, Allocator>&,
           set <Key, Compare, Allocator>&);
```

Constructors and Destructors

```
explicit
set(const Compare& comp = Compare(),
     const Allocator& alloc = Allocator());
```

The default constructor. Creates a set of zero elements. If the function object `comp` is supplied, it is used to compare elements of the set. Otherwise, the default function object in the template argument is used. The template argument defaults to `less (<)`. The allocator `alloc` is used for all storage management.

```
template <class InputIterator>
set(InputIterator first, InputIterator last,
     const Compare& comp = Compare()
     const Allocator& alloc = Allocator());
```

Creates a set of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. If the function object `comp` is supplied, it is used to compare elements of the set. Otherwise, the default function object in the template argument is used. The template argument defaults to `less (<)`. Uses the allocator `alloc` for all storage management.

set(const set<Key, Compare, Allocator>& x);
Copy constructor. Creates a copy of `x`.

~set();
The destructor. Releases any allocated memory for self.

Assignment Operator

set<Key, Compare, Allocator>&
operator=(const set<Key, Compare, Allocator>& x);
Assignment operator. Self will share an implementation with `x`. Returns a reference to self.

Allocator

allocator_type
get_allocator() const;
Returns a copy of the allocator used by self for storage management.

Iterators

iterator
begin();
Returns an `iterator` that points to the first element in self.

const_iterator
begin() const;
Returns a `const_iterator` that points to the first element in self.

iterator
end();
Returns an `iterator` that points to the past-the-end value.

const_iterator
end() const;
Returns a `const_iterator` that points to the past-the-end value.

reverse_iterator
rbegin();
Returns a `reverse_iterator` that points to the past-the-end value.

const_reverse_iterator
rbegin() const;
Returns a `const_reverse_iterator` that points to the past-the-end value.

reverse_iterator
rend();
Returns a `reverse_iterator` that points to the first element.

const_reverse_iterator
rend() const;
Returns a `const_reverse_iterator` that points to the first element.

Member Functions

```
void
clear();
```

Erases all elements from the set.

```
size_type
count(const key_type& x) const;
```

Returns the number of elements equal to `x`. Since a set supports unique keys, `count` will always return 1 or 0.

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

Returns `true` if the size is zero.

```
pair<iterator, iterator>
equal_range(const key_type&  x) const;
```

Returns `pair(lower_bound(x),upper_bound(x))`. The `equal_range` function indicates the valid range for insertion of `x` into the set.

```
size_type
erase(const key_type& x);
```

Deletes all the elements matching `x`. Returns the number of elements erased. Since a set supports unique keys, `erase` will always return 1 or 0.

```
iterator
erase(iterator position);
```

Deletes the map element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()` if the deleted item was the last one in this list.

```
iterator
erase(iterator first, iterator last);
```

Deletes the elements in the range (`first, last`). Returns an `iterator` pointing to the element following the last deleted element, or `end()` if there were no elements after the deleted range.

```
iterator
find(const key_value& x) const;
```

Returns an `iterator` that points to the element equal to `x`. If there is no such element, the iterator points to the past-the-end value.

```
pair<iterator, bool>
insert(const value_type& x);
```

Inserts `x` into self according to the comparison function object. The template's default comparison function object is `less (<)`. If the insertion succeeds, it returns a pair composed of the iterator position where the insertion took place, and `true`. Otherwise, the pair contains the end value, and `false`.

```
iterator
insert(iterator position, const value_type& x);
```

x is inserted into the set. A position may be supplied as a hint regarding where to do the insertion. If the insertion may be done right after position then it takes amortized constant time. Otherwise it will take O (log N) time. The return value points to the inserted x.

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

Inserts copies of the elements in the range [first, last].

```
key_compare
key_comp() const;
```

Returns the comparison function object for the set.

```
iterator
lower_bound(const key_type& x) const;
```

Returns an iterator that points to the first element that is greater than or equal to x. If there is no such element, the iterator points to the past-the-end value.

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

Returns size of the largest possible set.

```
size_type
size() const;
```

Returns the number of elements.

```
void
swap(set<Key, Compare, Allocator>& x);
```

Exchanges self with x.

```
iterator
upper_bound(const key_type& x) const
```

Returns an iterator that points to the first element that is greater than or equal to x. If there is no such element, the iterator points to the past-the-end value.

```
value_compare
value_comp() const;
```

Returns the set's comparison object. This is identical to the function key_comp().

Non-member Operators

```
template <class Key, class Compare, class Allocator>
 bool operator==(const set<Key, Compare, Allocator>& x,
                 const set<Key, Compare, Allocator>& y);
```

Equality operator. Returns true if x is the same as y.

```
template <class Key, class Compare, class Allocator>
 bool operator!=(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
```

Inequality operator. Returns `!(x==y)`.

```
template <class Key, class Compare, class Allocator>
bool operator<(const set <Key, Compare, Allocator>& x,
                const set <Key, Compare, Allocator>& y);
```

Returns true if the elements contained in `x` are lexicographically less than the elements contained in `y`.

```
template <class Key, class Compare, class Allocator>
bool operator>(const set <Key, Compare, Allocator>& x,
                const set <Key, Compare, Allocator>& y);
```

Returns `y < x`.

```
template <class Key, class Compare, class Allocator>
bool operator<=(const set <Key, Compare, Allocator>& x,
                const set <Key, Compare, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class Key, class Compare, class Allocator>
bool operator>=(const set <Key, Compare, Allocator>& x,
                const set <Key, Compare, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class Key, class Compare, class Allocator>
void swap(set <Key, Compare, Allocator>& a,
           set <Key, Compare, Allocator>& b);
```

Efficiently swaps the contents of `a` and `b`.

Example

```
//
// set.cpp
//
 #include <set>
 #include <iostream.h>

 typedef set<double, less<double>, allocator<double> > set_type;

 ostream& operator<<(ostream& out, const set_type& s)
 {
   copy(s.begin(), s.end(),
        ostream_iterator<set_type::value_type,char>(cout," "));
   return out;
 }

 int main(void)
 {
   // create a set of doubles
   set_type   sd;
   int         i;

   for(i = 0; i < 10; ++i) {
```

```
      // insert values
      sd.insert(i);
    }

    // print out the set
    cout << sd << endl << endl;

    // now let's erase half of the elements in the set
    int half = sd.size() >> 1;
    set_type::iterator sdi = sd.begin();
    advance(sdi,half);

    sd.erase(sd.begin(),sdi);

    // print it out again
    cout << sd << endl << endl;

    // Make another set and an empty result set
    set_type sd2, sdResult;
    for (i = 1; i < 9; i++)
       sd2.insert(i+5);
    cout << sd2 << endl;

    // Try a couple of set algorithms
    set_union(sd.begin(),sd.end(),sd2.begin(),sd2.end(),
              inserter(sdResult,sdResult.begin()));
    cout << "Union:" << endl << sdResult << endl;

    sdResult.erase(sdResult.begin(),sdResult.end());
    set_intersection(sd.begin(),sd.end(),
                     sd2.begin(),sd2.end(),
                     inserter(sdResult,sdResult.begin()));
    cout << "Intersection:" << endl << sdResult << endl;

    return 0;
  }

Output :

0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
6 7 8 9 10 11 12 13
Union:
5 6 7 8 9 10 11 12 13
Intersection:
6 7 8 9
```

Warnings

Member function templates are used in all containers provided by the Standard Template Library. An example of this feature is the constructor for set `<Key, Compare, Allocator>` that takes two templated iterators:

```
template <class InputIterator>
 set (InputIterator, InputIterator,
      const Compare& = Compare(),
      const Allocator& = Allocator());
```

set also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature, we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a set in the following two ways:

```
int intarray[10];
set<int> first_set(intarray, intarray + 10);
set<int> second_set(first_set.begin(),
                                          first_set.end());
```

but not this way:

```
set<long> long_set(first_set.begin(),
                                          first_set.end());
```

since the `long_set` and `first_set` are not the same type.

Also, many compilers do not support default template arguments. If your compiler is one of these you need to always supply the `Compare` template argument, and the `Allocator` template argument. For instance, you need to write :

`set<int, less<int>, allocator<int> >`

instead of :

`set<int>`

See Also ***allocator***, ***bidirectional_iterator, Container***, ***lexicographical_compare***

Summary Basic set operation for sorted sequences.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator
set_difference (InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator
set_difference (InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result, Compare comp);
```

Description The ***set_difference*** algorithm constructs a sorted difference that includes copies of the elements that are present in the range `[first1, last1)` but are not present in the range `[first2, last2)`. It returns the end of the constructed range.

As an example, assume we have the following two sets:

```
1 2 3 4 5
```

and

```
3 4 5 6 7
```

The result of applying ***set_difference*** is the set:

```
1 2
```

The result of ***set_difference*** is undefined if the result range overlaps with either of the original ranges.

set_difference assumes that the ranges are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Use the ***set_symetric_difference*** algorithm to return a result that contains all elements that are not in common between the two sets.

Complexity At most ((last1 - first1) + (last2 - first2)) * 2 -1 comparisons are performed.

Example

```
//
// set_diff.cpp
//
 #include <algorithm>
 #include <set>
 #include <iostream.h>
 int main()
 {
   //Initialize some sets
   int a1[10] = {1,2,3,4,5,6,7,8,9,10};
   int a2[6]  = {2,4,6,8,10,12};
   set<int, less<int> > all(a1, a1+10), even(a2, a2+6),
                        odd;
   //Create an insert_iterator for odd
   insert_iterator<set<int, less<int> > >
                 odd_ins(odd, odd.begin());

   //Demonstrate set_difference
   cout << "The result of:" << endl << "{";
   copy(all.begin(),all.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} - {";
   copy(even.begin(),even.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} =" << endl << "{";
   set_difference(all.begin(), all.end(),
                  even.begin(), even.end(), odd_ins);
   copy(odd.begin(),odd.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "}" << endl << endl;
   return 0;
 }
Output :
The result of:
{1 2 3 4 5 6 7 8 9 10 } - {2 4 6 8 10 12 } =
{1 3 5 7 9 }
```

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

```
set<int, less<int> allocator<int> >
```

instead of :

```
set<int>
```

See Also ***includes***, ***set***, ***set_union***, ***set_intersection***, ***set_symmetric_difference***

set_intersection

Algorithm

Summary

Basic set operation for sorted sequences.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator
set_intersection (InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator last2,
                  OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator
set_intersection (InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp);
```

Description

The ***set_intersection*** algorithm constructs a sorted intersection of elements from the two ranges. It returns the end of the constructed range. When it finds an element present in both ranges, ***set_intersection*** always copies the element from the first range into `result`. This means that the result of ***set_intersection*** is guaranteed to be stable. The result of ***set_intersection*** is undefined if the result range overlaps with either of the original ranges.

set_intersection assumes that the ranges are sorted using the default comparison operator less than (<), unless an alternative comparison operator (`comp`) is provided.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// set_intr.cpp
//
 #include <algorithm>
 #include <set>
 #include <iostream.h>
 int main()
 {

   //Initialize some sets
   int a1[10] = {1,3,5,7,9,11};
   int a3[4]  = {3,5,7,8};
   set<int, less<int> > odd(a1, a1+6),
                        result, small(a3,a3+4);
```

```
   //Create an insert_iterator for result
   insert_iterator<set<int, less<int> > >
                 res_ins(result, result.begin());

   //Demonstrate set_intersection
   cout << "The result of:" << endl << "{";
   copy(small.begin(),small.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} intersection {";
   copy(odd.begin(),odd.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} =" << endl << "{";
   set_intersection(small.begin(), small.end(),
                    odd.begin(), odd.end(), res_ins);
   copy(result.begin(),result.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "}" << endl << endl;

   return 0;
 }

Output :
The result of:
{3 5 7 8 } intersection {1 3 5 7 9 11 } =
{3 5 7 }
```

Warning

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

`set<int, less<int> allocator<int> >`

instead of :

`set<int>`

See Also

includes, ***set***, ***set_union***, ***set_difference***, ***set_symmetric_difference***

Algorithm

Summary

Basic set operation for sorted sequences.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator
set_symmetric_difference (InputIterator1 first1,
                          InputIterator1 last1,
                          InputIterator2 first2,
                          InputIterator2 last2,
                          OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator
set_symmetric_difference (InputIterator1 first1,
                          InputIterator1 last1,
                          InputIterator2 first2,
                          InputIterator2 last2,
                          OutputIterator result, Compare comp);
```

Description

set_symmetric_difference constructs a sorted symmetric difference of the elements from the two ranges. This means that the constructed range includes copies of the elements that are present in the range `[first1, last1)` but not present in the range `[first2, last2)`*and* copies of the elements that are present in the range `[first2, last2)` but not in the range `[first1, last1)`. It returns the end of the constructed range.

For example, suppose we have two sets:

```
1 2 3 4 5
```

and

```
3 4 5 6 7
```

The ***set_symmetric_difference*** of these two sets is:

```
1 2 6 7
```

The result of ***set_symmetric_difference*** is undefined if the result range overlaps with either of the original ranges.

set_symmetric_difference assumes that the ranges are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Use the ***set_symmetric_difference*** algorithm to return a result that includes elements that are present in the first set and not in the second.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// set_s_di.cpp
//
 #include<algorithm>
 #include<set>
 #include <istream.h>

 int main()
 {

   //Initialize some sets
   int a1[] = {1,3,5,7,9,11};
   int a3[] = {3,5,7,8};
   set<int, less<int> > odd(a1,a1+6), result,
                        small(a3,a3+4);

   //Create an insert_iterator for result
   insert_iterator<set<int, less<int> > >
                 res_ins(result, result.begin());

   //Demonstrate set_symmetric_difference
   cout << "The symmetric difference of:" << endl << "{";
   copy(small.begin(),small.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} with {";
   copy(odd.begin(),odd.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} =" << endl << "{";
   set_symmetric_difference(small.begin(), small.end(),
                       odd.begin(), odd.end(), res_ins);
   copy(result.begin(),result.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "}" << endl << endl;

   return 0;
 }

Output :
The symmetric difference of:
{3 5 7 8 } with {1 3 5 7 9 11 } =
{1 8 9 11 }
```

Warning

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

```
set<int, less<int>, allocator<int> >
```

instead of :

```
set<int>
```

See Also ***includes**, **set, set_union**, **set_intersection**, **set_difference***

Summary

Basic set operation for sorted sequences.

Synopsis

```
#include <algorithm>

template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator
set_union (InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator
set_union (InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result, Compare comp);
```

Description

The ***set_union*** algorithm constructs a sorted union of the elements from the two ranges. It returns the end of the constructed range. ***set_union*** is stable, that is, if an element is present in both ranges, the one from the first range is copied. The result of ***set_union*** is undefined if the result range overlaps with either of the original ranges. Note that ***set_union*** does not merge the two sorted sequences. If an element is present in both sequences, only the element from the first sequence is copied to `result`. (Use the ***merge*** algorithm to create an ordered merge of two sorted sequences that contains all the elements from both sequences.)

set_union assumes that the sequences are sorted using the default comparison operator less than (`<`), unless an alternative comparison operator (`comp`) is provided.

Complexity

At most `((last1 - first1) + (last2 - first2)) * 2 -1` comparisons are performed.

Example

```
//
// set_unin.cpp
//
 #include <algorithm>
 #include <set>
 #include <iostream.h>


 int main()
 {
```

```
   //Initialize some sets
   int a2[6]  = {2,4,6,8,10,12};
   int a3[4]  = {3,5,7,8};
   set<int, less<int> >  even(a2, a2+6),
                         result, small(a3,a3+4);

   //Create an insert_iterator for result
   insert_iterator<set<int, less<int> > >
                  res_ins(result, result.begin());

   //Demonstrate set_union
   cout << "The result of:" << endl << "{";
   copy(small.begin(),small.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} union {";
   copy(even.begin(),even.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "} =" << endl << "{";
   set_union(small.begin(), small.end(),
            even.begin(), even.end(), res_ins);
   copy(result.begin(),result.end(),
        ostream_iterator<int,char>(cout," "));
   cout << "}" << endl << endl;

   return 0;
 }

Output :
The result of:
{3 5 7 8 } union {2 4 6 8 10 12 } =
{2 3 4 5 6 7 8 10 12 }
```

Warning

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

```
set<int, less<int>, allocator<int> >
```

instead of :

```
set<int>
```

See Also

includes, ***set, set_intersection***, ***set_difference***, ***set_symmetric_difference***

Summary

Templated algorithm for sorting collections of entities.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
void sort (RandomAccessIterator first,
           RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last, Compare comp);
```

Description

The ***sort*** algorithm sorts the elements in the range `[first, last)` using either the less than (`<`) operator or the comparison operator `comp`. If the worst case behavior is important ***stable_sort*** or ***partial_sort*** should be used.

Complexity

sort performs approximately `NlogN`, where `N` equals `last - first`, comparisons on the average.

Example

```
//
// sort.cpp
//
 #include <vector>
 #include <algorithm>
 #include <functional>
 #include <iostream.h>

 struct associate
 {
   int num;
   char chr;

   associate(int n, char c) : num(n), chr(c){};
   associate() : num(0), chr('\0'){};
 };

 bool operator<(const associate &x, const associate &y)
 {
   return x.num < y.num;
 }

 ostream& operator<<(ostream &s, const associate &x)
 {
   return s << "<" << x.num << ";" << x.chr << ">";
 }
```

```
int main ()
{
  vector<associate>::iterator i, j, k;

  associate arr[20] =
       {associate(-4, ' '), associate(16, ' '),
        associate(17, ' '), associate(-3, 's'),
        associate(14, ' '), associate(-6, ' '),
        associate(-1, ' '), associate(-3, 't'),
        associate(23, ' '), associate(-3, 'a'),
        associate(-2, ' '), associate(-7, ' '),
        associate(-3, 'b'), associate(-8, ' '),
        associate(11, ' '), associate(-3, 'l'),
        associate(15, ' '), associate(-5, ' '),
        associate(-3, 'e'), associate(15, ' ')};


  // Set up vectors
  vector<associate> v(arr, arr+20), v1((size_t)20),
               v2((size_t)20);

  // Copy original vector to vectors #1 and #2
  copy(v.begin(), v.end(), v1.begin());
  copy(v.begin(), v.end(), v2.begin());

  // Sort vector #1
  sort(v1.begin(), v1.end());

  // Stable sort vector #2
  stable_sort(v2.begin(), v2.end());

  // Display the results
  cout << "Original    sort      stable_sort" << endl;
  for(i = v.begin(), j = v1.begin(), k = v2.begin();
      i != v.end(); i++, j++, k++)
  cout << *i << "     " << *j << "     " << *k << endl;

  return 0;
}

Output :
Original    sort      stable_sort
<-4; >     <-8; >     <-8; >
<16; >     <-7; >     <-7; >
<17; >     <-6; >     <-6; >
<-3;s>     <-5; >     <-5; >
<14; >     <-4; >     <-4; >
<-6; >     <-3;e>     <-3;s>
<-1; >     <-3;s>     <-3;t>
<-3;t>     <-3;l>     <-3;a>
<23; >     <-3;t>     <-3;b>
<-3;a>     <-3;b>     <-3;l>
<-2; >     <-3;a>     <-3;e>
<-7; >     <-2; >     <-2; >
<-3;b>     <-1; >     <-1; >
<-8; >     <11; >     <11; >
<11; >     <14; >     <14; >
<-3;l>     <15; >     <15; >
```

```
<15; >     <15; >     <15; >
<-5; >     <16; >     <16; >
<-3;e>     <17; >     <17; >
<15; >     <23; >     <23; >
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

stable_sort, ***partial_sort***, ***partial_sort_copy***

sort_heap

Algorithm

Summary Converts a heap into a sorted collection.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
  void
  sort_heap(RandomAccessIterator first,
            RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void
  sort_heap(RandomAccessIterator first,
            RandomAccessIterator last, Compare comp);
```

Description

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are:

1. `*a` is the largest element in the range.
2. `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in O(logN) time.

These properties make heaps useful as priority queues.

The ***sort_heap*** algorithm converts a heap into a sorted collection over the range `[first, last)` using either the default operator (`<`) or the comparison function supplied with the algorithm. Note that ***sort_heap*** is not stable, i.e., the elements may not be in the same relative order after ***sort_heap*** is applied.

Complexity

sort_heap performs at most `NlogN` comparisons where `N` is equal to `last - first`.

Example

```
//
// heap_ops.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>

 int main(void)
 {
   int d1[4] = {1,2,3,4};
   int d2[4] = {1,3,2,4};

   // Set up two vectors
```

```
    vector<int> v1(d1,d1 + 4), v2(d2,d2 + 4);

    // Make heaps
    make_heap(v1.begin(),v1.end());
    make_heap(v2.begin(),v2.end(),less<int>());
    // v1 = (4,x,y,z)  and  v2 = (4,x,y,z)
    // Note that x, y and z represent the remaining
    // values in the container (other than 4).
    // The definition of the heap and heap operations
    // does not require any particular ordering
    // of these values.

    // Copy both vectors to cout
    ostream_iterator<int,char> out(cout," ");
    copy(v1.begin(),v1.end(),out);
    cout << endl;
    copy(v2.begin(),v2.end(),out);
    cout << endl;

    // Now let's pop
    pop_heap(v1.begin(),v1.end());
    pop_heap(v2.begin(),v2.end(),less<int>());
    // v1 = (3,x,y,4) and v2 = (3,x,y,4)

    // Copy both vectors to cout
    copy(v1.begin(),v1.end(),out);
    cout << endl;
    copy(v2.begin(),v2.end(),out);
    cout << endl;

    // And push
    push_heap(v1.begin(),v1.end());
    push_heap(v2.begin(),v2.end(),less<int>());
    // v1 = (4,x,y,z) and v2 = (4,x,y,z)

    // Copy both vectors to cout
    copy(v1.begin(),v1.end(),out);
    cout << endl;
    copy(v2.begin(),v2.end(),out);
    cout << endl;

    // Now sort those heaps
    sort_heap(v1.begin(),v1.end());
    sort_heap(v2.begin(),v2.end(),less<int>());
    // v1 = v2 = (1,2,3,4)

    // Copy both vectors to cout
    copy(v1.begin(),v1.end(),out);
    cout << endl;
    copy(v2.begin(),v2.end(),out);
    cout << endl;

    return 0;
  }

Output :
4 2 3 1
4 3 2 1
```

```
3 2 1 4
3 1 2 4
4 3 1 2
4 3 2 1
1 2 3 4
1 2 3 4
```

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

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

instead of :

```
vector<int>
```

See Also ***make_heap, pop_heap, push_heap***

stable_partition

Algorithm

Summary

Places all of the entities that satisfy the given predicate before all of the entities that do not, while maintaining the relative order of elements in each group.

Synopsis

```
#include <algorithm>

template <class BidirectionalIterator, class Predicate>
BidirectionalIterator
stable_partition (BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred);
```

Description

The ***stable_partition*** algorithm places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it. It returns an iterator `i` that is one past the end of the group of elements that satisfy `pred`. In other words ***stable_partition*** returns `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`, `pred(*j) == false`. The relative order of the elements in both groups is preserved.

The ***partition*** algorithm can be used when it is not necessary to maintain the relative order of elements within the groups that do and do not match the predicate.

Complexity

The ***stable_partition*** algorithm does at most `(last - first) * log(last - first)` swaps. and applies the predicate exactly `last - first` times.

Example

```
//
// prtition.cpp
//
 #include <functional>
 #include <deque>
 #include <algorithm>
 #include <iostream.h>

 //Create a new predicate from unary_function
 template<class Arg>
 class is_even : public unary_function<Arg, bool>
 {
   public:
   bool operator()(const Arg& arg1)
   {
      return (arg1 % 2) == 0;
   }
 };
```

```
int main()
{
  //Initialize a deque with an array of ints
  int init[10] = {1,2,3,4,5,6,7,8,9,10};
  deque<int> d(init, init+10);

  //Print out the original values
  cout << "Unpartitioned values: " << endl << "     ";
  copy(d.begin(),d.end(),ostream_iterator<int,char>(cout," "));
  cout << endl << endl;

  //Partition the deque according to even/oddness
  stable_partition(d.begin(), d.end(), is_even<int>());

  //Output result of partition
  cout << "Partitioned values: " << endl << "     ";
  copy(d.begin(),d.end(),ostream_iterator<int,char>(cout," "));

  return 0;
}

Output :
Unpartitioned values:          1 2 3 4 5 6 7 8 9 10
Partitioned values:            10 2 8 4 6 5 7 3 9 1
Stable partitioned values:     2 4 6 8 10 1 3 5 7 9
```

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

`deque<int, allocator<int> >`

instead of :

`deque<int>`

See Also ***partition***

stable_sort

Summary Templated algorithm for sorting collections of entities.

Synopsis

```
#include <algorithm>

template <class RandomAccessIterator>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void stable_sort (RandomAccessIterator first,
                  RandomAccessIterator last,
                  Compare comp);
```

Description The ***stable_sort*** algorithm sorts the elements in the range `[first, last)`. The first version of the algorithm uses less than (`<`) as the comparison operator for the sort. The second version uses the comparison function `comp.`

The ***stable_sort*** algorithm is considered stable because the relative order of the equal elements is preserved.

Complexity ***stable_sort*** does at most `N(logN) **2,` where `N` equals `last -first`, comparisons; if enough extra memory is available, it is `NlogN.`

Example

```
//
// sort.cpp
//
 #include <vector>
 #include <algorithm>
 #include <functional>
 #include <iostream.h>

 struct associate
 {
   int num;
   char chr;

   associate(int n, char c) : num(n), chr(c){};
   associate() : num(0), chr('\0'){};
 };

 bool operator<(const associate &x, const associate &y)
 {
   return x.num < y.num;
 }
```

```
ostream& operator<<(ostream &s, const associate &x)
{
  return s << "<" << x.num << ";" << x.chr << ">";
}

int main ()
{
  vector<associate>::iterator i, j, k;

  associate arr[20] =
       {associate(-4, ' '), associate(16, ' '),
        associate(17, ' '), associate(-3, 's'),
        associate(14, ' '), associate(-6, ' '),
        associate(-1, ' '), associate(-3, 't'),
        associate(23, ' '), associate(-3, 'a'),
        associate(-2, ' '), associate(-7, ' '),
        associate(-3, 'b'), associate(-8, ' '),
        associate(11, ' '), associate(-3, 'l'),
        associate(15, ' '), associate(-5, ' '),
        associate(-3, 'e'), associate(15, ' ')};

  // Set up vectors
  vector<associate> v(arr, arr+20), v1((size_t)20),
                 v2((size_t)20);

  // Copy original vector to vectors #1 and #2
  copy(v.begin(), v.end(), v1.begin());
  copy(v.begin(), v.end(), v2.begin());

  // Sort vector #1
  sort(v1.begin(), v1.end());

  // Stable sort vector #2
  stable_sort(v2.begin(), v2.end());

  // Display the results
  cout << "Original    sort      stable_sort" << endl;
  for(i = v.begin(), j = v1.begin(), k = v2.begin();
      i != v.end(); i++, j++, k++)
  cout << *i << "     " << *j << "     " << *k << endl;

  return 0;
}
```

```
Output :
Original    sort      stable_sort
<-4; >     <-8; >     <-8; >
<16; >     <-7; >     <-7; >
<17; >     <-6; >     <-6; >
<-3;s>     <-5; >     <-5; >
<14; >     <-4; >     <-4; >
<-6; >     <-3;e>     <-3;s>
<-1; >     <-3;s>     <-3;t>
<-3;t>     <-3;l>     <-3;a>
<23; >     <-3;t>     <-3;b>
<-3;a>     <-3;b>     <-3;l>
<-2; >     <-3;a>     <-3;e>
<-7; >     <-2; >     <-2; >
```

```
<-3;b>     <-1; >     <-1; >
<-8; >     <11; >     <11; >
<11; >     <14; >     <14; >
<-3;l>     <15; >     <15; >
<15; >     <15; >     <15; >
<-5; >     <16; >     <16; >
<-3;e>     <17; >     <17; >
<15; >     <23; >     <23; >
```

Warning

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

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

instead of :

```
vector<int>
```

See Also

sort, ***partial_sort***, ***partial_sort_copy***

Summary

A container adapter which behaves like a stack (last in, first out).

Synopsis

```
#include <stack>

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

Description

The ***stack*** container adapter causes a container to behave like a "last in, first out" (LIFO) stack. The last item that was put ("pushed") onto the stack is the first item removed ("popped" off). The stack can adapt to any container that provides the operations, `back()`, `push_back()`, and `pop_back()`. In particular, ***deque*** , ***list*** , and ***vector*** can be used.

Interface

```
template <class T, class Container = deque<T> >
 class stack {

public:

// typedefs

   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef typename Container::allocator_type allocator_type

// Construct

   explicit stack (const allocator_type& = allocator_type());
   allocator_type get_allocator () const;

// Accessors

   bool empty () const;
   size_type size () const;
   value_type& top ();
   const value_type& top () const;
   void push (const value_type&);
   void pop ();
};

// Non-member Operators

template <class T, class Container>
 bool operator== (const stack<T, Container>&,
                  const stack<T, Container>&);

template <class T, class Container>
 bool operator!= (const stack<T, Container>&,
                  const stack<T, Container>&);
```

```
template <class T, class Container>
 bool operator< (const stack<T, Container>&,
                 const stack<T, Container>&);

template <class T, class Container>
 bool operator> (const stack<T, Container>&,
                 const stack<T, Container>&);

template <class T, class Container>
 bool operator<= (const stack<T, Container>&,
                 const stack<T, Container>&);

template <class T, class Container>
 bool operator>= (const stack<T, Container>&,
                 const stack<T, Container>&);
```

Constructor

explicit
stack(const allocator_type& alloc = allocator_taype());

Constructs an empty stack. The stack will use the allocator `alloc` for all storage management.

Allocator

allocator_type
get_allocator() const;

Returns a copy of the allocator used by self for storage management.

Member Functions

bool
empty() const;

Returns `true` if the stack is empty, otherwise `false`.

void
pop();

Removes the item at the top of the stack.

void
push(const value_type& x);

Pushes `x` onto the stack.

size_type
size() const;

Returns the number of elements on the stack.

value_type&
top();

Returns a reference to the item at the top of the stack. This will be the last item pushed onto the stack unless `pop()` has been called since then.

const value_type&
top() const;

Returns a constant reference to the item at the top of the stack as a `const value_type`.

Non-member Operators

```
template <class T, class Container>
  bool operator==(const stack<T, Container>& x,
                  const stack<T, Container>& y);
```

Equality operator. Returns `true` if `x` is the same as `y`.

```
template <class T, class Container>
  bool operator!=(const stack<T, Container>& x,
                  const stack<T, Container>& y);
```

Inequality operator. Returns `!(x==y)`.

```
template <class T, class Container>
  bool operator<(const stack<T, Container>& x,
                 const stack<T, Container>& y);
```

Returns `true` if the stack defined by the elements contained in `x` is lexicographically less than the stack defined by the elements of `y`.

```
template <class T, class Container>
  bool operator>(const stack<T, Container>& x,
                 const stack<T, Container>& y);
```

Returns `y < x`.

```
template <class T, class Container>
  bool operator<=(const stack<T, Container>& x,
                 const stack<T, Container>& y);
```

Returns `!(y < x)`.

```
template <class T, class Container>
  bool operator>=(const stack<T, Container>& x,
                 const stack<T, Container>& y);
```

Returns `!(x < y)`.

Example

```
//
// stack.cpp
//
 #include <stack>
 #include <vector>
 #include <deque>
 #include <string>
 #include <iostream.h>

 int main(void)
 {
   // Make a stack using a vector container
   stack<int,vector<int> > s;

   // Push a couple of values on the stack
   s.push(1);
   s.push(2);
   cout << s.top() << endl;

   // Now pop them off
   s.pop();
   cout << s.top() << endl;
```

```
    s.pop();

    // Make a stack of strings using a deque
    stack<string,deque<string> > ss;

    // Push a bunch of strings on then pop them off
    int i;
    for (i = 0; i < 10; i++)
    {
      ss.push(string(i+1,'a'));
      cout << ss.top() << endl;
    }
    for (i = 0; i < 10; i++)
    {
      cout << ss.top() << endl;
      ss.pop();
    }

    return 0;
  }
```

```
Output :
2
1
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaa
aaaaaaaa
aaaaaaa
aaaaaa
aaaaa
aaaa
aaa
aa
a
```

Warnings

If your compiler does not support template parameter defaults, you are required to supply a template parameter for `Container`. For example:

You would not be able to write,

```
stack<int> var;
```

Instead, you would have to write,

```
stack<int, deque<int> > var;
```

See Also ***allocator, Containers, deque, list, vector***

Stream Iterators

Iterators

Summary Stream iterators provide iterator capabilities for ostreams and istreams. They allow generic algorithms to be used directly on streams.

See the sections ***istream_iterator*** and ***ostream_iterator*** for a description of these iterators.

string

Summary A specialization of the ***basic_string*** class. For more information about strings, see the entry ***basic_string***.

Algorithm

Summary

Exchange values.

Synopsis

```
#include <algorithm>

template <class T>
 void swap (T& a, T& b);
```

Description

The ***swap*** algorithm exchanges the values of a and b. The effect is:

```
T tmp = a
a = b
b = tmp
```

See Also

iter_swap, ***swap_ranges***

Summary Exchange a range of values in one location with those in another

Synopsis

```
#include <algorithm>

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

Description The ***swap_ranges*** algorithm exchanges corresponding values in two ranges, in the following manner:

For each non-negative integer `n < (last - first)` the function exchanges `*(first1 + n)` with `*(first2 + n))`. After completing all exchanges, ***swap_ranges*** returns an iterator that points to the end of the second container, i.e., `first2 + (last1 -first1)`. The result of ***swap_ranges*** is undefined if the two ranges `[first, last)` and `[first2, first2 + (last1 - first1))` overlap.

Example

```
  //
  // swap.cpp
  //
   #include <vector>
   #include <algorithm>

   int main()
   {
     int d1[] = {6, 7, 8, 9, 10, 1, 2, 3, 4, 5};

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

     // Output original vector
     cout << "For the vector: ";
     copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

     // Swap the first five elements with the last five elements
     swap_ranges(v.begin(),v.begin()+5, v.begin()+5);

     // Output result
     cout << endl << endl
          << "Swapping the first five elements "
          << "with the last five gives: "
          << endl << "     ";
     copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

     return 0;
   }
```

```
Output :
For the vector: 6 7 8 9 10 1 2 3 4 5
Swapping the first five elements with the last five gives:
     1 2 3 4 5 6 7 8 9 10
Swapping the first and last elements gives:
     10 2 3 4 5 6 7 8 9 1
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

iter_swap, ***swap***

Summary

Applies an operation to a range of values in a collection and stores the result.

Synopsis

```
#include <algorithm>

template <class InputIterator,
          class OutputIterator,
          class UnaryOperation>
OutputIterator transform (InputIterator first,
                          InputIterator last,
                          OutputIterator result,
                          UnaryOperation op);

template <class InputIterator1,
          class InputIterator2,
          class OutputIterator,
          class BinaryOperation>
OutputIterator transform (InputIterator1 first1,
                          InputIterator1 last1,
                          InputIterator2 first2,
                          OutputIterator result,
                          BinaryOperation binary_op);
```

Description

The ***transform*** algorithm has two forms. The first form applies unary operation `op` to each element of the range `[first, last)`, and sends the result to the output iterator `result`. For example, this version of ***transform***, could be used to square each element in a vector. If the output iterator (`result`) is the same as the input iterator used to traverse the range, ***transform***, performs its transformation in place.

The second form of ***transform*** applies a binary operation, `binary_op`, to corresponding elements in the range `[first1, last1)` and the range that begins at `first2`, and sends the result to `result`. For example, ***transform*** can be used to add corresponding elements in two sequences, and store the set of sums in a third. The algorithm assumes, but does not check, that the second sequence has at least as many elements as the first sequence. Note that the output iterator `result` can be a third sequence, or either of the two input sequences.

Formally, ***transform*** assigns through every iterator `i` in the range `[result, result + (last1 - first1))` a new corresponding value equal to:

```
op(*(first1 + (i - result))
```

or

```
binary_op(*(first1 + (i - result), *(first2 + (i - result)))
```

transform returns `result + (last1 - first1)`. `op` and `binary_op` must not have any side effects. `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

Complexity Exactly `last1 - first1` applications of `op` or `binary_op` are performed.

Example

```
//
// trnsform.cpp
//
 #include <functional>
 #include <deque>
 #include <algorithm>
 #include <iostream.h>
 #include <iomanip.h>
 int main()
 {
   //Initialize a deque with an array of ints
   int arr1[5] = {99, 264, 126, 330, 132};
   int arr2[5] = {280, 105, 220, 84, 210};
   deque<int> d1(arr1, arr1+5), d2(arr2, arr2+5);
   //Print the original values
   cout << "The following pairs of numbers: "
        << endl << "     ";
   deque<int>::iterator i1;
   for(i1 = d1.begin(); i1 != d1.end(); i1++)
      cout << setw(6) << *i1 << " ";
   cout << endl << "     ";
   for(i1 = d2.begin(); i1 != d2.end(); i1++)
      cout << setw(6) << *i1 << " ";
   // Transform the numbers in the deque to their
   // factorials and store in the vector
   transform(d1.begin(), d1.end(), d2.begin(),
             d1.begin(), multiplies<int>());
   //Display the results
   cout << endl << endl;
   cout << "Have the products: " << endl << "     ";
   for(i1 = d1.begin(); i1 != d1.end(); i1++)
     cout << setw(6) << *i1 << " ";
   return 0;
 }

Output :
The following pairs of numbers:
         99    264    126    330    132
        280    105    220     84    210
Have the products:
      27720  27720  27720  27720  27720
```

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

```
deque<int, allocator<int> >
```

instead of:

```
deque<int>
```

Summary

Base class for creating unary function objects.

Synopsis

```
#include <functional>

template <class Arg, class Result>
struct unary_function{
  typedef Arg argument_type;
  typedef Result result_type;
};
```

Description

Function objects are objects with an `operator()` defined. They are important for the effective use of the standard library's generic algorithms, because the interface for each algorithmic template can accept either an object with an `operator()` defined or a pointer to a function. The standard library provides both a standard set of function objects, and a pair of classes that you can use as the base for creating your own function objects.

Function objects that take one argument are called *unary function objects*. Unary function objects are required to provide the typedefs `argument_type` and `result_type`. The ***unary_function*** class makes the task of creating templated unary function objects easier by providing the necessary typedefs for a unary function object. You can create your own unary function objects by inheriting from ***unary_function***.

See Also

function objects, and Function Objects Section in User's Guide.

unary_negate

Function Adapter (Negator)

Summary

Function object that returns the complement of the result of its unary predicate

Synopsis

```
#include<functional>

template <class Predicate>
class unary_negate : public unary_function<typename
                                        Predicate::argument_type,
                                        bool>;
```

Description

unary_negate is a function object class that provides a return type for the function adapter ***not1***. ***not1*** is a function adapter, known as a negator, that takes a unary predicate function object as its argument and returns a unary predicate function object that is the complement of the original.

Note that ***not1*** works only with function objects that are defined as subclasses of the class ***unary_function***.

Interface

```
template <class Predicate>
class unary_negate
  : public unary_function<Predicate::argument_type, bool> {
  typedef typename unary_function<typename
    Predicate::argument_type,bool>::argument_type argument_type;
  typedef typename unary_function<typename
    Predicate::argument_type,bool>::result_type result_type;
public:
   explicit unary_negate (const Predicate&);
   bool operator() (const argument_type&) const;
};

template<class Predicate>
unary_negate <Predicate> not1 (const Predicate&);
```

Constructor

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

Construct a `unary_negate` object from predicate `pred`.

Operator

```
bool
operator()(const argument_type& x) const;
```

Return the result of `pred(x)`

See Also

not1, not2, unary_function, ***binary_negate***

uninitialized_copy

Memory Management

Summary An algorithms that uses ***construct*** to copy values from one range to another location.

Synopsis

```
#include <memory>

template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy (InputIterator first,
                                    InputIterator last,
                                    ForwardIterator result);
```

Description ***uninitialized_copy*** copies all items in the range `[first, last)` into the location beginning at `result` using the ***construct*** algorithm.

See Also ***construct***

Summary

Algorithm that uses the ***construct*** algorithm for setting values in a collection.

Synopsis

```
#include <memory>

template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first,
                        ForwardIterator last,
                        const T& x);
```

Description

uninitialized_fill initializes all of the items in the range `[first, last)` to the value `x`, using the ***construct*** algorithm.

See Also

construct, uninitialized_fill_n

Summary

Algorithm that uses the ***construct*** algorithm for setting values in a collection.

Synopsis

```
#include <memory>

template <class ForwardIterator,
          class Size, class T>
void uninitialized_fill_n (ForwardIterator first,
                           Size n, const T& x);
```

Description

unitialized_fill_n starts at the iterator `first` and initializes the first `n` items to the value `x`, using the ***construct*** algorithm.

See Also

construct, uninitialized_fill

unique, unique_copy

Algorithm

Summary

Removes consecutive duplicates from a range of values and places the resulting unique values into the result.

Synopsis

```
#include <algorithm>

template <class ForwardIterator>
ForwardIterator unique (ForwardIterator first,
                        ForwardIterator last);

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

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

template <class InputIterator,
          class OutputIterator,
          class BinaryPredicate>
OutputIterator unique_copy (InputIterator first,
                            InputIterator last,
                            OutputIterator result,
                            BinaryPredicate binary_pred);
```

Description

The ***unique*** algorithm moves through a sequence and eliminates all but the first element from every consecutive group of equal elements. There are two versions of the algorithm, one tests for equality, and the other tests whether a binary predicate applied to adjacent elements is true. An element is unique if it does not meet the corresponding condition listed here:

```
*i  ==  *(i  -  1)
```

or

```
binary_pred(*i, *(i - 1)) == true.
```

If an element is unique, it is copied to the front of the sequence, overwriting the existing elements. Once all unique elements have been identified. The remainder of the sequence is left unchanged, and ***unique*** returns the end of the resulting range.

The ***unique_copy*** algorithm copies the first element from every consecutive group of equal elements, to an OutputIterator. The ***unique_copy*** algorithm, also has two versions--one that tests for equality and a second that tests adjacent elements against a binary predicate.

unique_copy returns the end of the resulting range.

Complexity

Exactly `(last - first) - 1` applications of the corresponding predicate are performed.

Example

```
//
// unique.cpp
//
 #include <algorithm>
 #include <vector>
 #include <iostream.h>
 int main()
 {
   //Initialize two vectors
   int a1[20] = {4, 5, 5, 9, -1, -1, -1, 3, 7, 5,
                 5, 5, 6, 7, 7, 7, 4, 2, 1, 1};
   vector<int> v(a1, a1+20), result;

   //Create an insert_iterator for results
   insert_iterator<vector<int> > ins(result,
                                result.begin());

   //Demonstrate includes
   cout << "The vector: " << endl << "    ";
   copy(v.begin(),v.end(),ostream_iterator<int,char>(cout," "));

   //Find the unique elements
   unique_copy(v.begin(), v.end(), ins);

   //Display the results
   cout << endl << endl
        << "Has the following unique elements:"
        << endl << "     ";
   copy(result.begin(),result.end(),
        ostream_iterator<int,char>(cout," "));

   return 0;
}

Output :
The vector:
    4 5 5 9 -1 -1 -1 3 7 5 5 5 6 7 7 7 4 2 1 1
Has the following unique elements:
     4 5 9 -1 3 7 5 6 7 4 2 1
```

Warning

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

`vector<int, allocator<int> >`

instead of:

`vector<int>`

upper_bound

Algorithm

Summary Determines the last valid position for a value in a sorted container.

Synopsis

```
#include <algorithm>
template <class ForwardIterator, class T>
   ForwardIterator
   upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
 template <class ForwardIterator, class T, class Compare>
    ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

Description The ***upper_bound*** algorithm is part of a set of binary search algorithms. All of these algorithms perform binary searches on ordered containers. Each algorithm has two versions. The first version uses the less than operator (`operator<`) to perform the comparison, and assumes that the sequence has been sorted using that operator. The second version allows you to include a function object of type `Compare`, and assumes that `Compare` is the function used to sort the sequence. The function object must be a binary predicate.

The ***upper_bound*** algorithm finds the *last* position in a container that `value` can occupy without violating the container's ordering. ***upper_bound***'s return value is the iterator for the first element in the container that is *greater than* `value`, or, when the comparison operator is used, the first element that does *not* satisfy the comparison function. Because the algorithm is restricted to using the less than operator or the user-defined function to perform the search, ***upper_bound*** returns an iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the appropriate version of the following conditions holds:

```
!(value < *j)
```

or

```
comp(value, *j) == false
```

Complexity ***upper_bound*** performs at most `log(last - first) + 1` comparisons.

Example

```
//
// ul_bound.cpp
//
 #include <vector>
```

```
#include <algorithm>
#include <iostream.h>

int main()
{
  typedef vector<int>::iterator iterator;
  int d1[11] = {0,1,2,2,3,4,2,2,2,6,7};

  // Set up a vector
  vector<int> v1(d1,d1 + 11);

  // Try lower_bound variants
  iterator it1 = lower_bound(v1.begin(),v1.end(),3);
  // it1 = v1.begin() + 4

  iterator it2 =
      lower_bound(v1.begin(),v1.end(),2,less<int>());
  // it2 = v1.begin() + 4

  // Try upper_bound variants
  iterator it3 = upper_bound(v1.begin(),v1.end(),3);
  // it3 = vector + 5

  iterator it4 =
     upper_bound(v1.begin(),v1.end(),2,less<int>());
  // it4 = v1.begin() + 5

  cout << endl << endl
       << "The upper and lower bounds of 3: ( "
       << *it1 << " , " << *it3 << " ]" << endl;

  cout << endl << endl
       << "The upper and lower bounds of 2: ( "
       << *it2 << " , " << *it4 << " ]" << endl;

  return 0;
}

Output :
The upper and lower bounds of 3: ( 3 , 4 ]
The upper and lower bounds of 2: ( 2 , 3 ]
```

Warning

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

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also

lower_bound, equal_range

value_type

Type primitive

Summary

Determine the type of value an iterator points to. This function is now obsolete. It is retained in order to provide backward compatibility and support compilers that do not provide partial specialization.

Synopsis

```
#include <iterator>

template <class T, class Distance>
inline T* value_type (const input_iterator<T, Distance>&)

template <class T, class Distance>
inline T* value_type (const forward_iterator<T, Distance>&)

template <class T, class Distance>
inline T* value_type (const bidirectional_iterator<T, Distance>&)

template <class T, class Distance>
inline T* value_type (const random_access_iterator<T, Distance>&)

template <class T>
inline T* value_type (const T*)
```

Description

The ***value_type*** function template returns a pointer to a default value of the type pointed to by an iterator. Five overloaded versions of this function template handle the four basic iterator types and simple arrays. Each of the first four take an iterator of a specific type, and return the value used to instantiate the iterator. The fifth version takes and returns a `T*` in order to handle the case when an iterator is a simple pointer.

This family of function templates can be used to extract a value type from an iterator and subsequently use that type to create a local variable. Typically the ***value_type*** functions are used like this:

```
template <class Iterator>
void foo(Iterator first, Iterator last)
{
  __foo(begin,end,value_type(first));
}

template <class Iterator, class T>
void __foo(Iterator first, Iterator last, T*>
{
  T temp = *first;
    …
}
```

The auxiliary function `__foo` extracts a usable value type from the iterator and then puts the type to work.

See Also

Other iterator primitives: ***distance_type, iterator_category, distance, advance***

Summary

Sequence that supports random access iterators.

Synopsis

```
#include <vector>

template <class T, class Allocator = allocator<T> >
class vector ;
```

Description

vector<T, Allocator> is a type of sequence that supports random access iterators. In addition, it supports amortized constant time insert and erase operations at the end. Insert and erase in the middle take linear time. Storage management is handled automatically. In ***vector***, `iterator` is a random access iterator referring to `T`. `const_iterator` is a constant random access iterator that returns a `const T&` when being dereferenced. A constructor for `iterator` and `const_iterator` is guaranteed. `size_type` is an unsigned integral type. `difference_type` is a signed integral type.

Any type used for the template parameter `T` must provide the following (where `T` is the `type`, `t` is a `value` of `T` and `u` is a `const value` of `T`):

```
 Default constructor   T()
   Copy constructors     T(t) and T(u)
   Destructor            t.~T()
   Address of            &t and &u yielding T* and
                          const T* respectively
   Assignment            t = a where a is a
                         (possibly const) value of T
```

Special Case

Vectors of bit values (boolean 1/0 values) are handled as a special case by the standard library, so that they can be efficiently packed several elements to a word. The operations for a boolean vector, ***vector<bool>***, are a superset of those for an ordinary vector, only the implementation is more efficient.

Two member functions are available to the boolean vector data type. One is `flip()`, which inverts all the bits of the vector. Boolean vectors also return as reference an internal value that also supports the `flip()` member function. The other ***vector<bool>***-specific member function is a second form of the `swap()` function

Interface

```
template <class T, class Allocator = allocator<T> >
 class vector {

public:

 // Types
```

```
  typedef T value_type;
  typedef Allocator allocator_type;
  typename reference;
  typename const_reference;
  typename iterator;
  typename const_iterator;
  typename size_type;
  typename difference_type;
  typename reverse_iterator;
  typename const_reverse_iterator;

// Construct/Copy/Destroy

  explicit vector (const Allocator& = Allocator());
  explicit vector (size_type, const Allocator& = Allocator ());
  vector (size_type, const T&, const Allocator& = Allocator());
  vector (const vector<T, Allocator>&);
  template <class InputIterator>
   vector (InputIterator, InputIterator,
           const Allocator& = Allocator ());
  ~vector ();
  vector<T,Allocator>& operator= (const vector<T, Allocator>&);
  template <class InputIterator>
   void assign (InputIterator first, InputIterator last);
  template <class Size, class TT>
   void assign (Size n);
  template <class Size, class TT>
   void assign (Size n, const TT&);
  allocator_type get_allocator () const;

// Iterators

  iterator begin ();
  const_iterator begin () const;
  iterator end ();
  const_iterator end () const;
  reverse_iterator rbegin ();
  const_reverse_iterator rbegin () const;
  reverse_iterator rend ();
  const_reverse_iterator rend () const;

// Capacity

  size_type size () const;
  size_type max_size () const;
  void resize (size_type);
  void resize (size_type, T);
  size_type capacity () const;
  bool empty () const;
  void reserve (size_type);

// Element Access

  reference operator[] (size_type);
  const_reference operator[] (size_type) const;
  reference at (size_type);
  const_reference at (size_type) const;
```

```
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

 // Modifiers

   void push_back (const T&);
   void pop_back ();
   iterator insert (iterator);
   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
   template <class InputIterator>
    void insert (iterator, InputIterator, InputIterator);
   iterator erase (iterator);
   iterator erase (iterator, iterator);
   void swap (vector<T, Allocator>&);

};

 // Non-member Operators

template <class T>
 bool operator== (const vector<T,Allocator>&,
                  const vector <T,Allocator>&);

template <class T>
 bool operator!= (const vector<T,Allocator>&,
                  const vector <T,Allocator>&);

template <class T>
 bool operator< (const vector<T,Allocator>&,
                 const vector<T,Allocator>&);

template <class T>
 bool operator> (const vector<T,Allocator>&,
                 const vector<T,Allocator>&);

template <class T>
 bool operator<= (const vector<T,Allocator>&,
                 const vector<T,Allocator>&);

template <class T>
 bool operator>= (const vector<T,Allocator>&,
                 const vector<T,Allocator>&);


// Specialized Algorithms

template <class T, class Allocator>
 void swap (const vector<T,Allocator>&, const vector<T,Allocator>&);
```

Constructors and Destructors

`explicit` **`vector(`**`const Allocator& alloc = Allocator());`

The default constructor. Creates a vector of length zero. The vector will use the allocator `alloc` for all storage management.

```
explicit vector(size_type n,
                 const Allocator& alloc = Allocator());
```
Creates a vector of length `n`, containing `n` copies of the default value for type `T`. Requires that `T` have a default constructor. The vector will use the allocator `alloc` for all storage management.

```
vector(size_type n, const T& value,
        const Allocator& alloc = Allocator());
```
Creates a vector of length `n`, containing `n` copies of value. The vector will use the allocator `alloc` for all storage management.

```
vector(const vector<T, Allocator>& x);
```
Creates a copy of `x`.

```
template <class InputIterator>
vector(InputIterator first, InputIterator last,
        const Allocator& alloc = Allocator());
```
Creates a vector of length `last - first`, filled with all values obtained by dereferencing the `InputIterators` on the range `[first, last)`. The vector will use the allocator `alloc` for all storage management.

```
~vector();
```
The destructor. Releases any allocated memory for this vector.

Iterators

```
iterator
begin();
```
Returns a random access `iterator` that points to the first element.

```
const_iterator
begin() const;
```
Returns a random access `const_iterator` that points to the first element.

```
iterator
end();
```
Returns a random access `iterator` that points to the past-the-end value.

```
const_iterator
end() const;
```
Returns a random access `const_iterator` that points to the past-the-end value.

```
reverse_iterator
rbegin();
```
Returns a random access `reverse_iterator` that points to the past-the-end value.

```
const_reverse_iterator
rbegin() const;
```
Returns a random access `const_reverse_iterator` that points to the past-the-end value.

```
reverse_iterator
rend();
```

Returns a random access `reverse_iterator` that points to the first element.

```
const_reverse_iterator
rend() const;
```

Returns a random access `const_reverse_iterator` that points to the first element.

Assignment Operator

```
vector<T, Allocator>&
operator=(const vector<T, Allocator>& x);
```

Erases all elements in self then inserts into self a copy of each element in `x`. Returns a reference to self.

Allocator

```
allocator_type
get_allocator() const;
```

Returns a copy of the allocator used by self for storage management.

Reference Operators

```
reference
operator[](size_type n);
```

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the `size` less one.

```
const_reference
operator[](size_type n) const;
```

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size` less one.

Member Functions

```
template <class InputIterator>
void
assign(InputIterator first, InputIterator last);
```

Erases all elements contained in self, then inserts new elements from the range `[first, last)`.

```
template <class Size, class T>
void
assign(Size n, const T& t);
```

Erases all elements contained in self, then inserts `n` instances of the default value of type `T`.

```
template <class Size, class T>
void
assign(Size n, const T& t);
```

Erases all elements contained in self, then inserts `n` instances of the value of `t`.

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

Returns a reference to element `n` of self. The result can be used as an lvalue. The index `n` must be between 0 and the `size` less one.

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

Returns a constant reference to element `n` of self. The index `n` must be between 0 and the `size` less one.

```
reference
back();
```

Returns a reference to the last element.

```
const_reference
back() const;
```

Returns a constant reference to the last element.

```
size_type
capacity() const;
```

Returns the size of the allocated storage, as the number of elements that can be stored.

```
void
clear() ;
```

Deletes all elements from the vector.

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

Returns `true` if the `size` is zero.

```
iterator
erase(iterator position);
```

Deletes the vector element pointed to by the iterator `position`. Returns an `iterator` pointing to the element following the deleted element, or `end()` if the deleted element was the last one in this vector.

```
iterator
erase(iterator first, iterator last);
```

Deletes the vector elements in the range (first, last). Returns an `iterator` pointing to the element following the last deleted element, or `end()` if there were no elements in the deleted range.

```
void
flip();
```

Flips all the bits in the vector. *This member function is only defined for* ***vector<bool>***.

```
reference
front();
```

Returns a reference to the first element.

```
const_reference
front() const;
```

Returns a constant reference to the first element.

```
iterator
insert(iterator position);
```

Inserts `x` before `position`. The return value points to the inserted `x`.

```
iterator
insert(iterator position, const T& x);
```

Inserts `x` before `position`. The return value points to the inserted `x`.

```
void
insert(iterator position, size_type n, const  T& x);
```

Inserts `n` copies of `x` before `position`.

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

Inserts copies of the elements in the range `[first, last]` before `position`.

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

Returns `size()` of the largest possible vector.

```
void
pop_back();
```

Removes the last element of self.

```
void
push_back(const T& x);
```

Inserts a copy of `x` to the end of self.

```
void
reserve(size_type n);
```

Increases the capacity of self in anticipation of adding new elements. `reserve` itself does not add any new elements. After a call to `reserve,` `capacity()` is greater than or equal to `n` and subsequent insertions will not cause a reallocation until the size of the vector exceeds `n`. Reallocation does not occur if `n` is less than `capacity()`. If reallocation does occur, then all iterators and references pointing to elements in the vector are invalidated. `reserve` takes at most linear time in the size of self.

```
void
resize(size_type sz);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` instances of the default value of type `T` are inserted at the end of the vector. If the new size is smaller than the current `capacity`, then the vector is truncated by erasing `size()-sz` elements off the end. If `sz` is equal to `capacity` then no action is taken.

```
void
resize(size_type sz, T c);
```

Alters the size of self. If the new size (`sz`) is greater than the current size, then `sz-size()` `c`'s are inserted at the end of the vector. If the new size is smaller than the current `capacity`, then the vector is truncated by erasing `size()-sz` elements off the end. If `sz` is equal to `capacity` then no action is taken.

```
size_type
size() const;
```

Returns the number of elements.

```
void
swap(vector<T, Allocator>& x);
```

Exchanges self with `x`, by swapping all elements.

```
void
swap(reference x, reference y);
```

Swaps the values of `x` and `y`. *This is a member function of* ***vector<bool>*** *only*.

Non-member Operators

```
template <class T, class Allocator>
bool operator==(const vector<T, Allocator>& x,
                 const vector<T, Allocator>& y);
```

Returns `true` if `x` is the same as `y`.

```
template <class T, class Allocator>
bool operator!=(const vector<T, Allocator>& x,
                 const vector<T, Allocator>& y);
```

Returns `!(x==y)`.

```
template <class T>
bool operator<(const vector<T, Allocator>& x,
                const vector<T, Allocator>& y);
```

Returns `true` if the elements contained in `x` are lexicographically less than the elements contained in `y`.

```
template <class T>
bool operator>(const vector<T, Allocator>& x,
                const vector<T, Allocator>& y);
```

Returns `y < x`.

```
template <class T>
bool operator<=(const vector<T, Allocator>& x,
                const vector<T, Allocator>& y);
```

Returns `!(y < x)`.

```
template <class T>
bool operator>=(const vector<T, Allocator>& x,
                const vector<T, Allocator>& y);
```

Returns `!(x < y)`.

Specialized Algorithms

```
template <class T, class Allocator>
void swap(vector <T, Allocator>& a, vector <T, Allocator>& b);
```

Efficiently swaps the contents of a and b.

Example

```
//
// vector.cpp
//
 #include <vector>
 #include <iostream.h>
 ostream& operator<< (ostream& out,
                      const vector<int, allocator>& v)
 {
   copy(v.begin(), v.end(), ostream_iterator<int,char>(out," "));
   return out;
 }
 int main(void)
 {
   // create a vector of doubles
   vector<int>         vi;
   int                 i;
   for(i = 0; i < 10; ++i) {
     // insert values before the beginning
     vi.insert(vi.begin(), i);
   }
   // print out the vector
   cout << vi << endl;
   // now let's erase half of the elements
   int half = vi.size() >> 1;
   for(i = 0; i < half; ++i) {
     vi.erase(vi.begin());
   }
   // print ir out again
   cout << vi << endl;
   return 0;
 }
Output :
9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Warnings

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

```
template <class InputIterator>
 vector (InputIterator, InputIterator,
         const Allocator = Allocator());
```

vector also has an insert function of this type. These functions, when not restricted by compiler limitations, allow you to use any type of input iterator as arguments. For compilers that do not support this feature we provide substitute functions that allow you to use an iterator obtained from the same type of container as the one you are constructing (or calling a member function on), or you can use a pointer to the type of element you have in the container.

For example, if your compiler does not support member function templates you can construct a vector in the following two ways:

```
int intarray[10];
vector<int> first_vector(intarray, intarray + 10);
vector<int> second_vector(first_vector.begin(),
                                   first_vector.end());
```

but not this way:

```
vector<long>
long_vector(first_vector.begin(),first_vector.end());
```

since the `long_vector` and `first_vector` are not the same type.

Additionally, if your compiler does not support default template parameters, you will need to supply the `Allocator` template argument. For instance, you will need to write :

`vector<int, allocator<int> >`

instead of :

`vector<int>`

See Also ***allocator, Containers, Iterators, lexicographical_compare***

wstring

Summary A specialization of the ***basic_string*** class. For more information about strings, see the entry ***basic_string***.