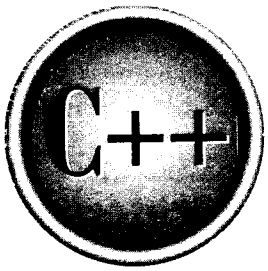
The Complete Reference



Chapter 35

STL Iterators, Allocators, and Function Objects

This chapter describes the classes and functions that support iterators, allocators, and function objects. These components are part of the standard template library. They may also be used for other purposes.

Iterators

While containers and algorithms form the foundation of the standard template library, iterators are the glue that holds it together. An *iterator* is a generalization (or perhaps more precisely, an abstraction) of a pointer. Iterators are handled in your program like pointers, and they implement the standard pointer operators. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array.

Standard C++ defines a set of classes and functions that support iterators. However, for the vast majority of STL-based programming tasks, you will not use these classes directly. Instead, you will use the iterators provided by the various containers in the STL, manipulating them like you would any other pointer. The preceding notwithstanding, it is still valuable to have a general understanding of the iterator classes and their contents. For example, it is possible to create your own iterators that accommodate special situations. Also, developers of third-party libraries will find the iterator classes useful.

Iterators use the header **<iterator>**.

The Basic Iterator Types

There are five types of iterators:

Iterator	Access Allowed
Random Access	Store and retrieve values. Elements may be accessed randomly.
Bidirectional	Store and retrieve values. Forward and backward moving.
Forward	Store and retrieve values. Forward moving only.
Input	Retrieve but not store values. Forward moving only.
Output	Store but not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator.

The STL also supports *reverse iterators*. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus, if

a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point one element before the end.

Stream-based iterators are available that allow you to operate on streams through iterators. Finally, insert iterator classes are provided that simplify the insertion of elements into a container.

All iterators must support the pointer operations allowed by their type. For example, an input iterator class must support ->, ++, *, ==, and !=. Further, the * operator cannot be used to assign a value. By contrast, a random-access iterator must support ->, +, ++, -,,- -, *, <, >, <=, >=, -=, +=, ==, !=, and [ ]. Also, the * must allow assignment.

The Low-Level Iterator Classes

The **<iterator>** header defines several classes that provide support for and aid in the implementation of iterators. As explained in Chapter 24, each of the STL containers defines its own iterator type, which is **typedef**ed as **iterator**. Thus, when using the standard STL containers, you will not usually interact directly with the low-level iterator classes themselves. But you can use the classes described here to derive your own iterators.

Several of the iterator classes make use of the **ptrdiff_t** type. This type is capable of representing the difference between two pointers.

iterator

The **iterator** class is a base for iterators. It is shown here:

```
template <class Cat, class T, class Dist = ptrdiff_t,
  class Pointer = T *, class Ref = T &>
struct iterator {
  typedef T value_type;
  typedef Dist difference_type;
  typedef Pointer pointer;
  typedef Ref reference;
  typedef Cat iterator_category;
};
```

Here, **difference_type** is a type that can hold the difference between two addresses, **value_type** is the type of value operated upon, **pointer** is the type of a pointer to a value, **reference** is the type of a reference to a value, and **iterator_category** describes the type of the iterator (such as input, random-access, etc.).

The following category classes are provided.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
```

```
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public
         bidirectional_iterator_tag {};
```

iterator_traits

The class **iterator_traits** provides a convenient means of exposing the various types defined by an iterator. It is defined like this:

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

The Predefined Iterators

The **<iterator>** header contains several predefined iterators that may be used directly by your program or to help create other iterators. These iterators are shown in Table 35-1. Notice that there are four iterators that operate on streams. The main purpose for the stream iterators is to allow streams to be manipulated by algorithms. Also notice the insert iterators. When these iterators are used in an assignment statement, they insert elements into a sequence rather than overwriting existing elements.

Each of the predefined iterators is examined here.

insert_iterator

The **insert_iterator** class supports output iterators that insert objects into a container. Its template definition is shown here:

```
template <class Cont> class insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Here, **Cont** is the type of container that the iterator operates upon. **insert_iterator** has the following constructor:

```
insert_iterator(Cont &cnt, typename Cont::iterator itr);
```

Here, *cnt* is the container being operated upon and *itr* is an iterator into the container that will be used to initialize the **insert_iterator**.

Class	Description
insert_iterator	An output iterator that inserts anywhere in the container.
back_insert_iterator	An output iterator that inserts at the end of a container.
front_insert_iterator	An output iterator that inserts at the front of a container.
reverse_iterator	A reverse, bidirectional, or random-access iterator.
istream_iterator	An input stream iterator.
istreambuf_iterator	An input streambuf iterator.
ostream_iterator	An output stream iterator.
ostreambuf_iterator	An output streambuf iterator.

Table 35-1. *The Predefined Iterator Classes*

insert_iterator defines the following operators: =, *, ++. A pointer to the container is stored in a protected variable called **container**. The container's iterator is stored in a protected variable called **iter**.

Also defined is the function **inserter( )**, which creates an **insert_iterator**. It is shown here:

```
template <class Cont, class Iterator> insert_iterator<Cont>
  inserter(Cont &cnt, Iterator itr);
```

Insert iterators insert into, rather than overwrite, the contents of a container. To fully understand the effects of an insert iterator, consider the following program. It first creates a small vector of integers, and then uses an **insert_iterator** to insert new elements into the vector rather than overwriting existing elements.

```
// Demonstrate insert_iterator.
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main()
{
  vector<int> v;
```

```
  vector<int>::iterator itr;
  int i;

  for(i=0; i<5; i++)
    v.push_back(i);

  cout << "Original array: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";
  cout << endl;

  itr = v.begin();
  itr += 2; // point to element 2

  // create insert_iterator to element 2
  insert_iterator<vector<int> > i_itr(v, itr);

  // insert rather than overwrite
  *i_itr++ = 100;
  *i_itr++ = 200;

  cout << "Array after insertion: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";

  return 0;
}
```

The output from the program is shown here:

```
Original array: 0 1 2 3 4
Array after insertion: 0 1 100 200 2 3 4
```

In the program, had the assignments of 100 and 200 been done using a standard iterator, the original elements in the array would have been overwritten. The same basic process applies to **back_insert_iterator** and **front_insert_iterator** as well.

back_insert_iterator

The **back_insert_iterator** class supports output iterators that insert objects on the end of a container using **push_back( )**. Its template definition is shown here:

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

Here, **Cont** is the type of container that the iterator operates upon. **back_insert_iterator** has the following constructor:

```
explicit back_insert_iterator(Cont &cnt);
```

Here, *cnt* is the container being operated upon. All insertions will occur at the end.

back_insert_iterator defines the following operators: =, *, ++. A pointer to the container is stored in a protected variable called **container**.

Also defined is the function **back_inserter( )**, which creates a **back_insert_iterator**. It is shown here:

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont &cnt);
```

front_insert_iterator

The **front_insert_iterator** class supports output iterators that insert objects on the front of a container using **push_front( )**. Its template definition is shown here:

```
template <class Cont> class front_insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Here, **Cont** is the type of container that the iterator operates upon. **front_insert_iterator** has the following constructor:

```
explicit front_insert_iterator(Cont &cnt);
```

Here, *cnt* is the container being operated upon. All insertions will occur at the front.

front_insert_iterator defines the following operators: =, *, ++. A pointer to the container is stored in a protected variable called **container**.

Also defined is the function **front_inserter( )**, which creates a **front_insert_iterator**. It is shown here:

```
template <class Cont> front_insert_iterator<Cont> inserter(Cont &cnt);
```

reverse_iterator

The **reverse_iterator** class supports reverse iterator operations. A reverse iterator operates the opposite of a normal iterator. For example, ++ causes a reverse iterator to back up. Its template definition is shown here:

```
template <class Iter> class reverse_iterator:
  public iterator<iterator_traits<Iter>::iterator_category,
                  iterator_traits<Iter>::value_type,
                  iterator_traits<Iter>::difference_type,
                  iterator_traits<Iter>::pointer,
                  iterator_traits<Iter>::reference>
```

Here, **Iter** is either a random-access iterator or a bidirectional iterator. **reverse_iterator** has the following constructors:

```
reverse_iterator( );
explicit reverse_iterator(Iter itr);
```

Here, *itr* is an iterator that specifies the starting location.

If **Iter** is a random-access iterator, then the following operators are available: ->, +, ++, -, - -, *, <, >, <=, >=, -=, +=, ==, !=, and [ ] . If **Iter** is a bidirectional iterator, then only ->, ++, - -, *, ==, and != are available.

The **reverse_iterator** class defines a protected member called **current**, which is an iterator to the current location.

The function **base( )** is also defined by **reverse_iterator**. Its prototype is shown here:

```
Iter base( ) const;
```

It returns an iterator to the current location.

istream_iterator

The **istream_iterator** class supports input iterator operations on a stream. Its template definition is shown here:

```
template <class T, class CharType, class Attr = char_traits<CharType>,
          class Dist = ptrdiff_t> class istream_iterator:
  public iterator<input_iterator_tag, T, Dist, const T *, const T &>
```

Here, **T** is the type of data being transferred, and **CharType** is the character type (**char** or **wchar_t**) that the stream is operating upon. **Dist** is a type capable of holding the difference between two addresses. **istream_iterator** has the following constructors:

```
istream_iterator( );
istream_iterator(istream_type &stream);
istream_iterator(const istream_iterator<T, CharType, Attr, Dist> &ob);
```

The first constructor creates an iterator to an empty stream. The second creates an iterator to the stream specified by *stream*. The type **istream_type** is a **typedef** that specifies the type of the input stream. The third form creates a copy of an **istream_iterator** object.

The **istream_iterator** class defines the following operators: ->, *, ++. The operators == and != are also defined for objects of type **istream_iterator**.

Here is a short program that demonstrates **istream_iterator**. It reads and displays characters from **cin** until a period is received.

```
// Use istream_iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  istream_iterator<char> in_it(cin);

  do {
    cout << *in_it++;
  } while (*in_it != '.');

  return 0;
}
```

istreambuf_iterator

The **istreambuf_iterator** class supports character input iterator operations on a stream. Its template definition is shown here:

```
template <class CharType, class Attr = char_traits<CharType> >
class istreambuf_iterator:
  public iterator<input_iterator_tag, CharType, typename Attr::off_type,
                  CharType *, CharType &>
```

Here, **CharType** is the character type (**char** or **wchar_t**) that the stream is operating upon. **istreambuf_iterator** has the following constructors:

```
istreambuf_iterator( ) throw( );
istreambuf_iterator(istream_type &stream) throw( );
istreambuf_iterator(streambuf_type *streambuf) throw( );
```

The first constructor creates an iterator to an empty stream. The second creates an iterator to the stream specified by *stream*. The type **istream_type** is a **typedef** that specifies the type of the input stream. The third form creates an iterator using the stream buffer specified by *streambuf*.

The **istreambuf_iterator** class defines the following operators: *, ++. The operators == and != are also defined for objects of type **istreambuf_iterator**.

istreambuf_iterator defines the member function **equal( )**, which is shown here:

```
bool equal(istreambuf_iterator<CharType, Attr> &ob);
```

Its operation is a bit counterintuitive. It returns **true** if the invoking iterator and *ob* both point to the end of the stream. It also returns **true** if both iterators do not point to the end of the stream. There is no requirement that what they point to be the same. It returns **false** otherwise. The == and != operators work in the same fashion.

ostream_iterator

The **ostream_iterator** class supports output iterator operations on a stream. Its template definition is shown here:

```
template <class T, class CharType, class Attr = char_traits<CharType> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Here, **T** is the type of data being transferred, **CharType** is the character type (**char** or **wchar_t**) that the stream is operating upon. **ostream_iterator** has the following constructors:

```
ostream_iterator(ostream_type &stream);
ostream_iterator(ostream_type &stream, const CharType *delim);
ostream_iterator(const ostream_iterator<T, CharType, Attr> &ob);
```

The first creates an iterator to the stream specified by *stream*. The type **ostream_type** is a **typedef** that specifies the type of the output stream. The second form creates an iterator to the stream specified by *stream* and uses the delimiters specified by *delim*. The delimiters are written to the stream after every output operation. The third form creates a copy of an **ostream_iterator** object.

The **ostream_iterator** class defines the following operators: =, *, ++.

Here is a short program that demonstrates **ostream_iterator**.

```
// Use ostream_iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  ostream_iterator<char> out_it(cout);

  *out_it = 'X';
  out_it++;
  *out_it = 'Y';
  out_it++;
  *out_it = ' ';

  char str[] = "C++ Iterators are powerful.\n";
  char *p = str;

  while(*p) *out_it++ = *p++;

  ostream_iterator<double> out_double_it(cout);
  *out_double_it = 187.23;
  out_double_it++;
  *out_double_it = -102.7;

  return 0;
}
```

The output from this program is shown here:

```
XY C++ Iterators are powerful.
187.23-102.7
```

ostreambuf_iterator

The **ostreambuf_iterator** class supports character output iterator operations on a stream. Its template definition is shown here:

```
template <class CharType, class Attr = char_traits<CharType> >
class ostreambuf_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Here, **CharType** is the character type (**char** or **wchar_t**) that the stream is operating upon. **ostreambuf_iterator** has the following constructors:

```
ostreambuf_iterator(ostream_type &stream) throw( );
ostreambuf_iterator(streambuf_type *streambuf) throw( );
```

The first creates an iterator to the stream specified by *stream*. The type **ostream_type** is a **typedef** that specifies the type of the input stream. The second form creates an iterator using the stream buffer specified by *streambuf*. The type **streambuf_type** is a **typedef** that specifies the type of the stream buffer.

The **ostreambuf_iterator** class defines the following operators: =, *, ++. The member function **failed( )** is also defined as shown here:

```
bool failed( ) const throw( );
```

It returns **false** if no failure has occurred and **true** otherwise.

Two Iterator Functions

There are two special functions defined for iterators: **advance( )** and **distance( )**. They are shown here:

```
template <class InIter, class Dist> void advance(InIter &itr, Dist d);
template <class InIter> ptrdiff_t distance(InIter start, InIter end);
```

The **advance( )** function increments *itr* by the amount specified by *d*. The **distance( )** function returns the number of elements between *start* and *end*.

The reason for these two functions is that only random-access iterators allow a value to be added to or subtracted from an iterator. The **advance( )** and **distance( )** functions overcome this restriction. It must be noted, however, that some iterators will not be able to implement these functions efficiently.

Function Objects

Function objects are classes that define **operator( )**. The STL defines several built-in function objects that your programs may use. You can also define your own function objects. Support for function objects is in the **<functional>** header. Also defined in **<functional>** are several entities that support function objects. These are binders, negators, and adaptors. Each is described here.

Note *Refer to Chapter 24 for an overview of function objects.*

Function Objects

Function objects come in two varieties: binary and unary. The built-in binary function objects are shown here:

plus	minus	multiplies	divides	modulus
equal_to	not_equal_to	greater	greater_equal	less
less_equal	logical_and	logical_or		

Here are the built-in unary function objects.

logical_not	negate

The general form for invoking a function object is shown here:

func_ob<type>( )

For example,

```
less<int>()
```

invokes **less( )** relative to operands of type **int**.

A base class for all binary function objects is **binary_function**, shown here:

```
template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

The base class for all unary functions is **unary_function**, shown here:

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};
```

These template classes provide concrete type names for the generic data types used by the function object. Although they are technically a convenience, they are almost always used when creating function objects.

The template specifications for all binary function objects are similar, and the template specifications for all unary function objects are similar. Here are examples of each:

```
template <class T> struct plus : binary_function<T, T, T>
{
  T operator() (const T &arg1, const T &arg2) const;
};

template <class T> struct negate : unary_function<T, T>
{
  T operator() (const T &arg) const;
};
```

Each **operator( )** function returns the specified result.

Binders

Binders bind a value to an argument of a binary function object, producing a unary function object. There are two binders: **bind2nd( )** and **bind1st( )**. Here is how they are defined:

```
template <class BinFunc, class T>
        binder1st<BinFunc> bind1st(const BinFunc &op, const T &value);
template <class BinFunc, class T>
        binder2nd<BinFunc> bind2nd(const BinFunc &op, const T &value);
```

Here, *op* is a binary function object, such as **less( )** or **greater( )**, that provides the desired operation, and *value* is the value being bound. **bind1st( )** returns a unary function object that has *op*'s left-hand operand bound to *value*. **bind2nd( )** returns a unary function object that has *op*'s right-hand operand bound to *value*. The **bind2nd( )** binder is by far the most commonly used. In either case, the outcome of a binder is a unary function object that is bound to the value specified.

The **binder1st** and **binder2nd** classes are shown here:

```
template <class BinFunc> class binder1st:
  public unary_function(typename BinFunc::second_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::first_argument_type value;
public:
```

```
  binder1st(const BinFunc &op,
            const typename BinFunc::first_argument_type &v);
  result_type operator()(const argument_type &v) const;
 };

template <class BinFunc> class binder2nd:
  public unary_function(typename BinFunc::first_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::second_argument_type value;
public:
  binder2nd(const BinFunc &op,
            const typename BinFunc::second_argument_type &v);
  result_type operator()(const argument_type &v) const;
 };
```

Here, *BinFunc* is the type of a binary function object. Notice that both classes inherit **unary_function**. This is why the resulting object of **bind1st( )** and **bind2nd( )** can be used anywhere that a unary function can be.

Negators

Negators return predicates that yield the opposite of whatever predicate they modify. The negators are **not1( )** and **not2( )**. They are defined like this:

```
template <class UnPred> unary_negate<UnPred> not1(const UnPred &pred);
template <class BinPred> binary_negate<BinPred> not2(const BinPred &pred);
```

The classes are shown here:

```
template <class UnPred> class unary_negate:
  public unary_function<typename UnPred::argument_type, bool>
{
public:
  explicit unary_negate(const UnPred &pred);
  bool operator()(const argument_type &v) const;
};

template <class BinPred> class binary_negate:
```

```
  public binary_function<typename BinPred::first_argument_type,
                         typename BinPred::second_argument_type,
                         bool>
{
public:
  explicit binary_negate(const BinPred &pred);
  bool operator()(const first_argument_type &v1,
                  const second_argument_type &v2) const;
};
```

In both classes, **operator( )** returns the negation of the predicate specified by *pred*.

Adaptors

The header **<functional>** defines several classes called *adaptors* that allow you to adapt a function pointer to a form that can be used by the STL. For example, you can use an adaptor to allow a function such as **strcmp( )** to be used as a predicate. Adaptors also exist for member pointers.

The Pointer-to-Function Adaptors

The pointer-to-function adaptors are shown here:

```
template <class Argument, class Result>
  pointer_to_unary_function<Argument, Result>
    ptr_fun(Result (*func)(Argument));
template <class Argument1, class Argument2, class Result>
  pointer_to_binary_function<Argument1, Argument2, Result>
    ptr_fun(Result (*func)(Argument1, Argument2));
```

Here, **ptr_fun( )** returns either an object of type **pointer_to_unary_function** or **pointer_to_binary_function**. These classes are shown here:

```
template <class Argument, class Result>
class pointer_to_unary_function:
  public unary_function<Argument, Result>
{
public:
  explicit pointer_to_unary_function(Result (*func)(Argument));
  Result operator()(Argument arg) const;
};
```

```
template <class Argument1, class Argument2, class Result>
class pointer_to_binary_function:
  public binary_function<Argument1, Argument2, Result>
{
public:
  explicit pointer_to_binary_function(
            Result (*func)(Argument1, Argument2));
  Result operator()(Argument1 arg1, Argument2 arg2) const;
};
```

For unary functions, **operator( )** returns

func(*arg*).

For binary functions, **operator( )** returns

func(*arg1*, *arg2*);

The type of the result of the operation is specified by the **Result** generic type.

The Pointer-to-Member Function Adaptors

The pointer-to-member function adaptors are shown here:

```
template<class Result, class T>
    mem_fun_t<Result, T> mem_fun(Result (T::*func)( ));
template<class Result, class T, class Argument>
    mem_fun1_t<Result, T, Argument>
      mem_fun1(Result (T::*func)(Argument));
```

Here, **mem_fun( )** returns an object of type **mem_fun_t**, and **mem_fun1( )** returns an object of type **mem_fun1_t**. These classes are shown here:

```
template <class Result, class T> class mem_fun_t:
  public unary_function<T *, Result> {
public:
  explicit mem_fun_t(Result (T::*func)());
  Result operator() (T *func) const;
```

```
};

template <class Result, class T,
          class Argument> class mem_fun1_t:
  public binary_function<T *, Argument, Result> {
public:
  explicit mem_fun1_t(Result (T::*func)(Argument));
  Result operator() (T *func, Argument arg) const;
};
```

Here, the **mem_fun_t** constructor calls the member function specified as its parameter. The **mem_fun1_t** constructor calls the member function specified as its first parameter, passing a value of type **Argument** as its second parameter.

There are parallel classes and functions for using references to members. The general form of the functions is shown here.

```
template<class Result, class T>
  mem_fun_ref_t<Result, T> mem_fun_ref(Result (T::*func)( ));

template<class Result, class T, class Argument>
  mem_fun1_ref_t<Result, T, Argument>
    mem_fun1_ref(Result (T::*func)(Argument));
```

The classes are shown here:

```
template <class Result, class T> class mem_fun_ref_t:
  public unary_function<T, Result>
{
public:
  explicit mem_fun_ref_t(Result (T::*func)());
  Result operator()(T &func) const;
};

template <class Result, class T, class Argument>
  class mem_fun1_ref_t:
    public binary_function<T, Result, Argument>
{
public:
  explicit mem_fun1_ref_t(Result (T::*func)(Argument));
  Result operator()(T &func, Argument arg) const;
};
```

Allocators

An allocator manages memory allocation for a container. Since the STL defines a default allocator that is automatically used by the containers, most programmers will never need to know the details about allocators or create their own. However, these details are useful if you are creating your own library classes, etc.

All allocators must satisfy several requirements. First, they must define the following types:

const_pointer	A **const** pointer to an object of type **value_type**.
const_reference	A **const** reference to an object of type **value_type**.
difference_type	Can represent the difference between two addresses.
pointer	A pointer to an object of type **value_type**.
reference	A reference to an object of type **value_type**.
size_type	Capable of holding the size of the largest possible object that can be allocated.
value_type	The type of object being allocated.

Second, they must provide the following functions.

address	Returns a pointer given a reference.
allocate	Allocates memory.
deallocate	Frees memory.
max_size	Returns the maximum number of objects that can be allocated.
construct	Constructs an object.
destroy	Destroys an object.

The operations == and != must also be defined.

The default allocator is **allocator**, and it is defined within the header **<memory>**. Its template specification is shown here:

```
template <class T> class allocator
```

Here, **T** is the type of objects that **allocator** will be allocating. **allocator** defines the following constructors.

```
allocator( ) throw( );
allocator(const allocator<T> &ob) throw( );
```

The first creates a new allocator. The second creates a copy of *ob*.

The operators == and != are defined for **allocator**. The member functions defined by **allocator** are shown in Table 35-2.

One last point: A specialization of **allocator** for **void** * pointers is also defined.

Function	Description
pointer address(reference *ob*) const; const_pointer address(const_reference *ob*) const;	Returns the address of *ob*.
pointer allocate(size_type *num*, allocator<void>::const_pointer *h* = 0);	Returns a pointer to allocated memory that is large enough to hold *num* objects of type T. The value of *h* is a hint to the function that can be used to help satisfy the request or ignored.
void construct(pointer *ptr*, const_reference *val*);	Constructs an object of type T with the value specified by *val* at *ptr*.
void deallocate(pointer *ptr*, size_type *num*);	Deallocates *num* objects of type T starting at *ptr*. The value of *ptr* must have been obtained from **allocate( )**.
void destroy(pointer *ptr*);	Destroys the object at *ptr*. Its destructor is automatically called.
size_type max_size( ) const throw( );	Returns the maximum number of objects of type T that can be allocated.

Table 35-2. *Member Functions of* ***allocator***