第35章　STL迭代器、分配器和函数对象

本章描述支持迭代器、分配器、函数对象的类和函数。这些组件是标准模板库的一部分，也可把它们用做其他目的。

35.1　迭代器

虽然容器和算法形成了标准模板库的基础，但是是迭代器（iterator）把它们组合在一起的。迭代器是指针的归纳（或者更准确地说，是抽象）。迭代器在程序中像指针一样处理，并且它们实现了标准指针运算符。它们提供了一种遍历容器内容的方法，这种方法与使用指针来遍历数组类似。

标准C++定义了一系列支持迭代器的类和函数。然而，对于大多数基于STL的编程任务，不应该直接使用这些类。相反，应该使用STL中各种容器提供的迭代器，就像操作任何其他指针那样操作它们。虽然如此，仍然有必要对迭代器类及其内容有一个一般的理解。例如，你可以创建适合于特定情况的迭代器，还有，第三方库的开发者会发现迭代器类特别有用。迭代器使用头文件<iterator>。

35.1.1　基本迭代器类型

迭代器有五种类型，如下表所示：

迭代器	允许的访问
Random Access（随机访问）	存储和检索值。元素可以被随机访问
Bidirectional（双向）	存储和检索值。可以前向和后向移动
Forward（前向）	存储和检索值。仅可以前向移动
Input（输入）	检索但不存储值。仅可以前向移动
Output（输出）	存储但不检索值。仅可以前向移动

一般来讲，具有较大访问能力的迭代器可以用在具有较小访问能力的迭代器所用的地方。例如，在使用输入迭代器的地方，可以使用前向迭代器。

STL也支持反向迭代器。反向迭代器或者是双向的或者是随机访问的迭代器，反向迭代器以相反方向遍历一个序列。因此，如果反向迭代器指向一个序列的末端，增加迭代器将会导致它指向其末端前面的一个元素。

基于流的迭代器可以允许你通过迭代器对流进行操作，而插入迭代器类可以简化向容器中插入元素的操作。

所有迭代器必须支持其类型所允许的指针操作。例如，输入迭代器类必须支持->, ++, *, ==, 和 != 运算符。此外，不能使用 * 运算符来赋值。相反，随机访问迭代器必须支持 ->, +, ++, -, --, *, <, >, <=, >=, -=, +=, ==, != 和 [ ]运算符。还有，* 必须允许赋值。

35.1.2　低级迭代器类

<iterator>头文件定义了一些类，这些类提供了对迭代器实现的支持和帮助。正如在第24章中所讲述的，每一 STL 容器定义它自己的迭代器类型：即类型定义为 iterator。因此，当使用标准STL容器时，通常不直接与低级迭代器类交互，但是可以使用下面描述的类来导出自己的迭代器。有些迭代器类使用 ptrdiff_t 类型，这种类型能够描述两个指针之间的不同。

iterator

iterator 类是迭代器的基础，如下所示：

```
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;
};
```

其中，difference_type 是一种可以保存两个地址间的差值的类型。value_typ 是所操作的类型值，pointer 是指向一个值的指针类型，reference 是值的引用类型，iterator_category 描述了迭代器的类型（比如输入、随机访问等）。

同时也提供了如下所示的类。

```
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

iterator_traits 类提供了一种方便的方法，用来显示迭代器定义的各种类型，它被定义如下：

```
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;
}
```

35.1.3　预定义迭代器

<iterator>头文件包含了一些预定义的迭代器，可以直接为程序所用或帮助创建其他的迭代器。这些迭代器如表35.1所示。注意有四个迭代器用于流的操作。流迭代器的主要用处是让算法来处理流。同时也要注意插入迭代器。当这些迭代器用于赋值语句时，它们把元素插入一个序列，而不是覆盖现存的元素。下面将讨论每一种预定义迭代器。

表 35.1 预定义迭代器类

类	说明
insert_iterator	插入到容器中任何地方的输出迭代器
back_insert_iterator	插入到容器末端的输出迭代器
front_insert_iterator	插入到容器前端的输出迭代器
reverse_iterator	反向、双向或随机访问迭代器
istream_iterator	输入流迭代器
istreambuf_iterator	输入流缓冲迭代器
ostream_iterator	输出流迭代器
ostreambuf_iterator	输出流缓冲迭代器

insert_iterator

insert_iterator类支持向容器中插入对象的输出迭代器，其模板定义如下所示：

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

其中，Cont是迭代器所操作的容器类型，insert_iterator类有下列构造函数：

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

其中，cnt是所操作的容器，itr是容器中的迭代器，用于初始化insert_iterator类。

insert_iterator类定义了下列运算符：=, *, ++。指向容器的指针存储在一个受保护的变量container中，容器的迭代器存储在一个受保护的变量iter中。

另外被定义的有函数inserter( )，它创建了一个insert_iterator类，如下所示：

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

插入迭代器插入而不是覆盖容器的内容。要全面理解插入迭代器的作用，考虑下列程序。该程序首先创建一个小的整型矢量，然后使用insert_iterator向矢量中插入一个新元素，而不是覆盖现有的元素。

```
// 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;
}
```

程序的输出如下所示：

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

在这个程序中，如果使用标准迭代器来赋值 100 和 200，数组中最初的元素会被覆盖。同样的过程适用于 back_insert_iterator 和 front_insert_iterator。

back_insert_iterator

back_insert_iterator 类使用 push_back( )来支持在容器的末端插入对象的输出迭代器，其模板定义如下所示：

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

其中，Cont 是迭代器操作的容器类型，back_insert_iterator 有下列构造函数：

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

其中，cnt 是所操作的容器类型。所有的插入操作发生在容器的末端。back_insert_iterator 定义了下列运算符：=, *, ++。指向容器的指针保存在一个受保护的变量 container 中。

同时定义了函数 back_inserter( )，它创建了一个 back_insert_iterator 类，如下所示：

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

front_insert_iterator

front_insert_iterator 类使用 push_front( )来支持将对象插入容器前端的输出迭代器，其模板定义如下所示：

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

其中，Cont 是迭代器所操作的容器类型，front_insert_iterator 类有如下构造函数：

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

其中，cnt是所操作的容器。所有的插入操作都发生在容器的前端。front_insert_iterator类定义了下列运算符：=, *, ++。容器的指针存储在一个受保护的变量container中。

同时被定义的有函数front_inserter( )，它创建了一个front_insert_iterator类，如下所示：

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

reverse_iterator

reverse_iterator类支持反向迭代器操作，反向迭代器与普通迭代器的操作相反。例如，++运算符会造成反向迭代器回退。其模板定义如下所示：

```
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>
```

其中，Iter是一个随机访问迭代器，或者是一个双向迭代器。reverse_iterator有下列构造函数：

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

其中，itr是一个指定开始位置的迭代器。

如果Iter是一个随机访问迭代器，那么下列运算符可用：->, +,++, -, --, *, <, >, <=, >=, -=, +=, ==, !=, 和[ ]。如果Iter是一个双向迭代器，那么只有->, ++, --, *, ==, 和 != 是可用的。

reverse_iterator类定义了一个受保护的成员current，它是一个到当前位置的迭代器。

函数base( )也由reverse_iterator定义，其原型如下所示：

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

它返回当前位置的迭代器。

istream_iterator

istream_iterator类支持对流进行的输入迭代器操作，它的模板定义如下所示：

```
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 &>
```

其中，T代表要传送的数据类型，CharType是在其上操作的流的字符类型（char或者是wchar_t），Dist是能够保存两种地址之间差值的类型。istream_iterator类有如下构造函数：

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

第一个构造函数创建了一个空的流迭代器。第二个构造函数创建了由stream指定的流迭代器，类型istream_type是指定输入流类型的一个类型定义（typedef）。第三个构造函数创建了一个istream_iterator对象的副本，istream_iterator类定义了下列运算符：->, *, ++。对istream_iterator类型的对象，也定义了运算符 == 和 !=。

这里有一个小程序，演示了istream_iterator。它读取并显示一个来自 cin 的字符，直到接收到一个点号为止。

```
// 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

istreambuf_iterator 类支持对流进行操作的字符输入迭代器，其模板定义如下所示：

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

其中，CharType 是在其上进行操作的字符类型（char 或者 wchar_t）。istreambuf_iterator 有如下构造函数：

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

第一个构造函数创建了一个空的流迭代器。第二个构造函数创建了一个由 stream 指定的流的迭代器，istream_type 类型是指定输入流类型的类型定义（ typedef ）。第三个构造函数创建了一个使用由 streambuf 指定的流缓存的迭代器。

istreambuf_iterator 类定义了如下运算符：*, ++。对 istreambuf_iterator 类型的对象，也定义了运算符 == 和 !=。

istreambuf_iterator 类定义了成员函数 equal( )，如下所示：

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

这个函数的操作与直觉相反。如果调用迭代器和 ob 都指向流的末端，它返回真。如果两个迭代器都不指向流的末端，它也返回真，对它们所指向的内容是否相同并没有要求。否则，返回假。==和 !=运算符的工作方式也是一样。

ostream_iterator

ostream_iterator 类支持对流进行输出迭代器操作，它的模板定义如下所示：

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

其中，T是要被传送的数据类型，CharType是对流进行操作的字符类型(char或者wchar_t)。ostream_iterator类有如下构造函数：

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

第一个构造函数创建了一个由stream指定的流迭代器。类型ostream_type是指定输出流类型的类型定义（typedef）。第二个构造函数创建了一个由stream指定的流的迭代器，并使用由delim指定的分隔符。在每次输出操作之后，分隔符被写入流中。第三个构造函数创建了一个ostream_iterator对象的副本。

ostream_iterator类定义了下列运算符：=, *, ++。

下面是一个演示ostream_iterator类的小程序。

```
// 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;
}
```

这个程序的输出如下所示：

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

ostreambuf_iterator

ostreambuf_iterator类支持对流进行操作的字符输出迭代器，它的模板定义如下所示：

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

其中，CharType 是对流进行操作的字符类型(char 或者 wchar_t)，ostreambuf_iterator 类有下列构造函数：

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

第一个构造函数创建了一个由 stream 指定的流的迭代器。类型 ostream_type 是指定输入流类型的类型定义(typedef)。第二个构造函数使用由 streambuf 指定的流缓存来创建一个迭代器。类型 streambuf_type 是指定流缓存类型的 typedef。

ostreambuf_iterator 类定义下列运算符：=, *, ++。成员函数 failed( )也被定义，如下所示：

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

如果没有错误发生，它返回假；否则，返回真。

35.1.4　两个迭代器函数

有两个为迭代器定义的特殊的函数：advance( )和 distance( )，如下所示：

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

advance( )函数增加 itr，增量由 d 指定。distance( )函数返回在 start 和 end 之间的元素数。定义这两个函数的原因是，只有随机访问迭代器允许从迭代器中加或者减。advance( ) 和 distance( )函数克服了这个限制。然而，必须注意，有些迭代器不能够有效地实现这些函数。

35.2　函数对象

函数对象是定义 operator( )的类。STL 定义了一些应用程序可能用到的内嵌的函数对象。也可以定义自己的函数对象。对函数对象的支持是在<functional>头文件中。支持函数对象的一些实体也定义在<functional>头文件中，它们是绑定器、取反器和适配器。下面分别对它们进行讨论。

注意：关于函数对象的概述，请参见第 24 章。

35.2.1　函数对象

函数对象有两种：二元和一元的。内嵌的二元函数对象如下所示：

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

下面是内嵌的一元函数对象：

logical_not	negate

调用函数对象的一般形式如下所示：

```
func_ob<type>( )
```

例如，

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

调用与整型操作数有关的less( )函数。

所有二元函数对象的基类是binary_function，如下所示：

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

所有一元函数的基类是unary_function，如下所示：

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

这些模板类提供了函数对象所用的通用数据类型的具体类型名。尽管从技术上讲它们很方便，但它们几乎总是在创建函数对象时使用。

所有二元函数对象的模板规范是类似的，所有一元函数对象的模板规范也是类似的。下面是一些例子：

```
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;
};
```

每个operator( )函数都返回指定的结果。

35.2.2 绑定器

绑定器把一个值绑定到一个二元函数对象的变元上，产生一个一元函数对象。有两个绑定器：bind2nd( )和bind1st( )。下面显示了它们是如何定义的：

```
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);
```

其中，op是一个二元函数对象，例如less( )或者greater( )，它提供所要求的操作，value是要被绑定的值。bind1st( )返回一个一元函数对象，其op的左操作数绑定到value上。bind2nd( )返回一个一元函数对象，其op的右操作数绑定到value上。bind2nd( )绑定器是迄今为止最常用的。在任一种情况下，绑定器的结果都是一个一元函数对象，这个一元函数对象绑定到指定的

值上。

bind1st和bind2nd类如下所示：

```
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) ccnst;
 };

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;
 };
```

其中，BinFunc是一个二元函数对象的类型。注意这两个类都是从unary_function继承而来的，这就是bind1st( )和bind2nd( )的结果对象可用在一元函数所用的任何地方的原因。

35.2.3 取反器

取反器返回与它们所修改的谓词相反的值。取反器有not1( )和not2( )，定义如下：

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

这些类显示如下：

```
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;
};
```

在每一种类中，operator( )都返回与 pred 指定的谓词相反的值。

35.2.4 适配器

头文件<functional>定义了一些称为适配器的类，允许你用来把函数指针适配为 STL 可以使用的形式。例如，可以使用适配器来允许一个函数如 strcmp( )作为谓词使用。适配器也可作为成员指针存在。

函数指针适配器

函数指针适配器如下所示：

```
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));
```

其中，ptr_fun( )返回一个 pointer_to_unary_function 或 pointer_to_binary_function 类型的对象。这些类被显示如下：

```
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;
};
```

对于一元函数，operator( )返回 func(arg)。

对于二元函数，operator( )返回 func(arg1, arg2)。

操作结果的类型由 Result 通用类型指定。

成员函数指针适配器

成员函数指针适配器如下所示：

```
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));
```

其中，mem_fun( )返回 mem_fun_t 类型的对象，mem_fun1( )返回一个 mem_fun1_t 类型的对象。这些类显示如下：

```
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;
};
```

其中，mem_fun_t 构造函数调用指定的成员函数作为它的参数。

mem_fun1_t 构造函数调用指定的成员函数作为它的第一个参数，传递一个 Argument 类型的值作为它的第二个参数。

有一些类和函数可以用做成员的引用，这些函数的一般形式如下所示：

```
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));
```

这些类显示如下：

```
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;
};
```

35.3 分配器

分配器管理容器的内存分配。因为STL定义了一个默认的分配器，该分配器为容器自动使用，所以大多数程序员永远不需要知道分配器的细节问题或者自己创建它们。然而，如果要创建自己的库类等，这些细节是很有用处的。

所有的分配器必须满足几个要求。首先，它们必须定义下列类型：

const_pointer	指向 value_type 类型对象的 const 指针
const_reference	value_type 类型对象的 const 引用
difference_type	能够表示两个地址间的差
Pointer	指向 value_type 类型对象的指针
reference	value_type 类型对象的引用
size_type	能够保存可分配的最大可能对象的大小
value_type	要被分配的对象类型

其次，它们必须提供下列函数：

address	返回给定引用的指针
allocate	分配内存
deallocate	释放内存
max_size	返回能够分配的对象的最大数目
construct	构建对象
destroy	销毁对象

也必须定义 == 和 != 操作。

默认分配器是 allocator,它在头文件<memory>中定义。它的模板规范如下所示：

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

其中，T 是 allocator 要分配的对象类型。allocator 定义了下列构造函数：

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

第一个构造函数创建一个新的分配器。第二个创建一个 ob 的副本。对于 allocator，也定义了运算符 == 和 !=。由 allocator 定义的成员函数如表 35.2 所示。

最后，也定义了 void* 指针的 allocator 规范。

表 35.2 分配器的成员函数

函数	说明
pointer address(reference *ob*) const; const_pointer address(const_reference *ob*) const;	返回 ob 的地址
pointer allocate(size_type *num*, allocator<void>::const_pointer *h* = 0);	返回一个指向所分配内存的指针，这个内存足够大，可以保存 num 个 T 类型的对象。h 的值说明可使用函数来满足请求或者忽略掉请求

（续表）

函数	说明
void construct(pointer *ptr*, const_reference *val*);	构造一个 T 类型的对象，其值由在 ptr 处的 val 指定
void deallocate(pointer *ptr*, size_type *num*);	释放 num 个以 ptr 开始的 T 类型的对象。ptr 的值必须从 allocate( )中获得
void destroy(pointer *ptr*);	在 ptr 处销毁对象，其构造函数被自动调用
size_type max_size( ) const throw( );	返回可被分配的最大数量的 T 类型对象