附　录

泛型算法（按字母排序）

在本附录中，我们将依次介绍每个单独的算法。我们选择以字母顺序给出这些算法（除了少数例外），以便可以很容易地查阅它们。本附录给出这些算法的一般形式是，①列出函数原型；②提供一两段说明性的文字，指出某些不直观的行为或可能性；以及最重要的③提供一个程序例子来说明怎样使用该算法。

所有泛型算法的前两个实参都是一对 iterator（迭代器），通常称为 first 和 last，标记出内置数组或容器中要操作的元素的范围。元素范围的表示法（有时也被称为**左包含区间**）通常被写作：

```
// to be read as : includes first and
// each element up to but not including last
[ first, last ]
```

表示该范围从 first 开始，直到 last 结束，但是不包括 last。当表示为以下形式时：

```
first == last
```

该范围被称为是空的。

对于 iterator 对的要求是，它必须能够从 first 开始，通过反复应用递增操作符可以到达 last。但是，编译器自己不能保证这一点。不能满足这个要求将导致未定义的运行时刻行为——通常是程序的核心转储。

每个算法的声明都指出了其 iterator 必须支持的最小分类（关于五个 iterator 分类的简要讨论见 12.4 节）。例如，find()实现了对一个容器的单遍只读遍历，它至少需要一个 InputIterator。它也可以被传递一个 ForwardIterator、BidirectionalIterator 或 RamdomAccessIterator。然而，如果向它传递一个 OutputIterator 就会引起错误。给一个算法传递一个无效的 iterator 类别这样的错误并不一定会在编译时刻被检查出来，因为 iterator 类别不是实际的类型，而是被传递给函数模板的类型参数。

有些算法支持多个版本，一个版本利用内置操作符，而另一个版本接受一个函数对象或指向函数的指针，以便提供该操作符的替代实现。例如，缺省的 unique()利用容器的底层元素类型的等于操作符，来比较两个相邻的元素。但是，如果底层元素类型没有提供等于操作

符，或者我们希望定义不同的元素相等语义，那么可以传递一个函数的对象或指向函数的指针，然后再由该函数提供期望的语义。然而，另外一些算法被分成两个不同名字的实例，其中，第二个版本的实例都有后缀_if，比如 find_if()。例如，有一个使用内置等于操作符的 replace()实例，和一个带有函数对象或函数指针的 replace_if()实例。

对那些修改所操作容器的算法，一般有两个版本：一个是实地（in-place）版本，改变当前正被应用的容器；而在另一个版本中，将返回容器的一个拷贝，所做的修改被应用在这份拷贝上。例如，有 replace()和 replace_copy()两个算法，拷贝版本在名字中总会有_copy。但是，并不是每一个要改变相关容器的算法都有拷贝版本。例如，sort()算法就没有提供拷贝版本。在这种情况下，如果我们希望该算法在拷贝上进行操作，则需要自己做一份拷贝，并传递给算法。

要使用泛型算法，我们必须包含相关的头文件：

```
#include <algorithm>
```

如果要使用以下四个算术算法：adjacent_difference()、accumulate()、inner_product()以及 partial_sum()，则必须包含：

```
#include <numeric>
```

本附录中，实现算法的代码以及这些算法所操作的容器类型反映了当前可用的标准库实现。iostream 库反映了标准 C++之前的实现版本，例如，包括“使用 iostream.h 头文件”这样的行为。在模板机制中，模板参数不支持缺省实参。为了使程序能在读者当前的系统上运行，或许需要修改某些声明。

在[MUSSER96]中，我们可以找到关于泛型算法更完美、更详细的讨论，虽然对于最终的 C++标准库而言，这些讨论有些过时。

accumulate()

```
template < class InputIterator, class Type >
Type accumulate(
    InputIterator first, InputIterator last,
    Type init );
template < class InputIterator, class Type,
           class BinaryOperation >
Type accumulate(
    InputIterator first, InputIterator last,
    Type init, BinaryOperation op );
```

accumulate()的第一个版本把由“iterator 对[first,last]”标记的序列中的元素之和，加到一个由 init 指定的初始值上。例如，已知序列{1,1,2,3,5,8}和初始值 0，则结果是 20。在第二个版本中，不再是做加法，而是传递进来的二元操作被应用在元素上。例如，如果向 accumulate()传递函数对象 times<int>，则结果是 240，当然，假设初始值是 1，而不是 0。accumulate()是一个算术算法。要使用它，我们必须包含<numeric>头文件。

```
#include <numeric>
#include <list>
```

```
#include <functional>
#include <iostream.h>

/*
 * 输出为:
   accumulate()
          operating on values {1,2,3,4}
          result with default addition: 10
          result with plus<int> function object: 10
 */

int main()
{
   int ia[] = { 1, 2, 3, 4 };
   list<int,allocator> ilist( ia, ia+4 );

   int ia_result = accumulate(&ia[0], &ia[4], 0);
   int ilist_res = accumulate(
       ilist.begin(), ilist.end(), 0, plus<int>() );

   cout << "accumulate()\n\t"
        << "operating on values {1,2,3,4}\n\t"
        << "result with default addition: "
        << ia_result << "\n\t"
        << "result with plus<int> function object: "
        << ilist_res
        << endl;
   }
```

adjacent_difference()

```
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 op );
```

adjacent_differece()的第一个版本创建了一个新的序列，该序列中的每个新值（第一个元素除外）都代表了当前元素与上一个元素的差。例如，已知序列{0,1,1,2,3,5,8}，则新序列的第一个元素只是原来序列第一个元素的拷贝：0。第二个元素是前两个元素的差：1。第三个元素是第二个和第三个元素的差，即 1-1，为 0，等等。新序列是{0,1,0,1,1,2,3}。

第二个版本用指定的二元操作计算相邻元素的差。例如，使用同一个序列，让我们传递 times<int>函数对象。同样，新序列的第一个元素只是原来序列第一个元素的拷贝：0。第二个元素是原来第一个和第二个元素的积，也是 0。第三个元素是第二和第三个元素的积，即

1*1，为 1 等等。新序列是 {0,0,1,2,6,15,40} 。

在两个版本中，OutputIterator 总是指向新序列末元素的下一个位置。adjacent_difference() 是一种算术算法。使用这两个版本都必须包含头文件<numeric>。

```
#include <numeric>
#include <list>
#include <functional>
#include <iterator>
#include <iostream.h>

int main()
{
   int ia[] = { 1, 1, 2, 3, 5, 8 };

   list<int,allocator> ilist(ia, ia+6);
   list<int,allocator> ilist_result(ilist.size());


   adjacent_difference(ilist.begin(), ilist.end(),
                       ilist_result.begin() );

   // 输出为:
   // 1 0 1 1 2 3

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

   adjacent_difference( ilist.begin(), ilist.end(),
                        ilist_result.begin(), times<int>() );

   // 输出为:
   // 1 1 2 6 15 40
   copy( ilist_result.begin(), ilist_result.end(),
         ostream_iterator<int>(cout," "));
   cout << endl;
}
```

adjacent_find()

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

adjacent_find()在由[first,last]标记的元素范围内，查找第一对相邻的重复元素。如果找到，则返回一个 ForwardIterator，并指向这对元素的第一个元素；否则返回 last。例如，已知序列 {0,1,1,2,2,4}，元素对{1,1}被找到，函数返回指向第一个 1 的 iterator:

```
#include <algorithm>
#include <vector>
```

```
#include <iostream.h>
#include <assert.h>

class TwiceOver {
public:
    bool operator() ( int val1, int val2 )
         { return val1 == val2/2 ? true : false; }
};

int main()
{
    int ia[] = { 1, 4, 4, 8 };
    vector< int, allocator > vec( ia, ia+4 );

    int *piter;
    vector< int, allocator >::iterator iter;

    // piter 指向 ia[1]
    piter = adjacent_find( ia, ia+4 );
    assert( *piter == ia[ 1 ] );

    // iter 指向 vec[2]
    iter = adjacent_find( vec.begin(), vec.end(), TwiceOver() );
    assert( *iter == vec[ 2 ] );

    // 到达这里表示一切顺利
    cout << "ok: adjacent-find() succeeded!\n";
}
```

binary_search()

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

binary_search()在由[first,last]标记的有序序列中查找 value。如果找到，则返回 true。否则，返回 false。第一个版本假设该容器是用底层类型的小于操作符排序的。在第二个版本中，我们指出了该容器是用指定的函数对象进行排序的：

```
#include <algorithm>
#include <vector>
#include <assert.h>

int main()
{
  int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};

  sort( &ia[0], &ia[12] );
  bool found_it = binary_search( &ia[0], &ia[12], 18 );
  assert( found_it == false );
```

```
    vector< int > vec( ia, ia+12 );
    sort( vec.begin(), vec.end(), greater<int>() );
    found_it = binary_search( vec.begin(), vec.end(),
                              26, greater<int>() );
    assert( found_it == true );
}
```

copy()

```
template < class InputIterator, class OutputIterator >
OutputIterator
copy( InputIterator first1, InputIterator last,
      OutputIterator first2 );
```

copy()把由[first,last)标记的序列中的元素，拷贝到由 first2 标记为开始的地方。它返回 first2，但此时 first2 已经被移动到最后一个插入元素的下一位置。例如，已知序列{0,1,2,3,4,5}，我们可以用下列调用将序列左移 1 位：

```
int ia[] = { 0, 1, 2, 3, 4, 5 };

// 左移1位，结果为{1,2,3,4,5,5}
copy( ia+1, ia+6, ia );
```

copy()从 ia 的第二个元素开始，把 1 拷贝到第一个位置上……，直到所有元素都被拷贝到它左边的位置上：

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

/* 生成:
   0 1 1 3 5 8 13
   将数组序列左移1位:
   1 1 3 5 8 13 13
   将vector序列左移2位:
   1 3 5 8 13 8 13
*/


int main()
{
    int ia[] = { 0, 1, 1, 3, 5, 8, 13 };
    vector< int, allocator > vec( ia, ia+7 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // 左移1位
    copy( ia+1, ia+7, ia );

    cout << "shifting array sequence left by 1:\n";
    copy( ia, ia+7, ofile ); cout << '\n';
```

```
        // 左移2位
        copy( vec.begin()+2, vec.end(), vec.begin() );

        cout << "shifting vector sequence left by 2:\n";
        copy( vec.begin(), vec.end(), ofile ); cout << '\n';
    }
```

copy_backward()

```
template < class BidirectionalIterator1,
           class BidirectionalIterator2 >
BidirectionalIterator2
copy_backward( BidirectionalIterator1 first,
               BidirectionalIterator1 last1,
               BidirectionalIterator2 last2 );
```

copy_backward()除了元素以相反的顺序被拷贝外，其他行为与 copy()相同。也就是说，拷贝操作从 last-1 开始，直到 first。这些元素也被从后向前拷贝到目标容器中，从 last2-1 开始，一直拷贝 last1-first 个元素。

例如，已知序列{0,1,2,3,4,5}，我们可以把最后三个元素（3,4,5）拷贝到前三个（0,1,2）中。做法是，把 first 设为值 0 的地址，last1 设为值 3 的地址，而 last2 设为值 5 的后一个位置。值为 5 的元素被赋给前面值为 2 的元素，而元素 4 被赋给前面值为 1 的元素。最后，元素 3 被赋给前面值为 0 的元素。结果序列是{3,4,5,3,4,5}。

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

class print_elements {
public:
   void operator()( string elem ) {
      cout << elem
           << ( _line_cnt++%8 ? " " : "\n\t" );
   }
   static void reset_line_cnt() { _line_cnt = 1; }

private:
   static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* 生成:
   原字符串为:
   The light untonsured hair grained and hued like
   pale oak
```

```
    copy_backward( begin+1, end-3, end )后的序列为:
    The light untonsured hair light untonsured hair grained
    and hued
*/

int main()
{
    string sa[] = {
        "The", "light", "untonsured", "hair",
        "grained", "and", "hued", "like", "pale", "oak" };

    vector< string, allocator > svec( sa, sa+10 );

    cout << "original list of strings:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n\n";

    copy_backward( svec.begin()+1, svec.end()-3, svec.end() );

    print_elements::reset_line_cnt();

    cout << "sequence after "
         << "copy_backward( begin+1, end-3, end ):\n";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n";
}
```

count()

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

count()利用等于操作符，把[first,last]标记范围内的元素与 value 进行比较。并返回容器中与 value 相等的元素的个数。［注意，标准库的实现支持早期的 count()版本。］

```
#include <algorithm>
#include <string>
#include <list>
#include <iterator>

#include <assert.h>
#include <iostream.h>
#include <fstream.h>

/***************************************************************
* 读入的文本如下：
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
*****************************************************************
   * 程序输出:
   *  count(): fiery occurs 2 times
*****************************************************************
*/

int main()
{
   ifstream infile( "alice_emma" );
   assert ( infile != 0 );

   list<string,allocator> textlines;

   typedef list<string,allocator>::difference_type diff_type;
   istream_iterator< string, diff_type > instream( infile ),
                    eos;

   copy( instream, eos, back_inserter( textlines ));
   string search_item( "fiery" );

/****************************************************************
     * 注意:这是使用count()的标准C++接口
     *      但是目前的RogueWave实现
     *      支持的是早期版本，其中没有开发distance_type
     *      因此 count() 将通过
    *      一个参数返回值
    *
    * 调用方式如下:
    *
    * typedef iterator_traits<InputIterator>::
     * distance_type dis_type;
     *
     * dis_type elem_count;
     * elem_count = count( textlines.begin(), textlines.end(),
    *                  search_item );
***************************************************************/

   int elem_count = 0;
   list<string,allocator>::iterator
      ibegin = textlines.begin(),
      iend   = textlines.end();

   // count()的过时形式
   count( ibegin, iend, search_item, elem_count );

   cout << "count(): " << search_item
        << " occurs "  << elem_count << " times\n";
}
```

count_if()

```
template< class InputIterator, class Predicate >
```

```
iterator_traits<InputIterator>::distance_type
count_if( InputIterator first,
          InputIterator last, Predicate pred );
```

count_if()对于[first,last]标记范围内的每个元素都应用 pred，并返回 pred 计算结果为 true 的次数。

```
#include <algorithm>
#include <list>
#include <iostream.h>

class Even {
public:
   bool operator()( int val )
        { return val%2 ? false : true; }
};

int main()
{
   int ia[] = {0,1,1,2,3,5,8,13,21,34};
   list< int,allocator > ilist( ia, ia+10 );

 /*
  * 目前编译器不支持
  *******************************************************
  typedef
     iterator_traits<InputIterator>::distance_type
     distance_type;

     distance_type ia_count, list_count;

     // 计算偶数元素: 4
     ia_count = count_if( &ia[0], &ia[10], Even() );
     list_count = count_if( ilist.begin(), ilist_end(),
                            bind2nd(less<int>(),10) );
  *******************************************************
  */

    int ia_count = 0;
    count_if( &ia[0], &ia[10], Even(), ia_count );

    // 生成结果为:
    //   count_if(): there are 4 elements that are even.

    cout << "count_if(): there are "
         << ia_count << " elements that are even.\n";

    int list_count = 0;
    count_if( ilist.begin(), ilist.end(),
              bind2nd(less<int>(),10), list_count );

    // 生成结果为:
    // count_if(): there are 7 elements that are less than 10.

    cout << "count_if(): there are "
         << list_count
```

```
             << " elements that are less than 10.\n";
    }
```

equal()

```
    template< class InputIterator1, class InputIterator2 >
    bool
    equal( InputIterator1 first1,
           InputIterator1 last, InputIterator2 first2 );

    template< class InputIterator1, class InputIterator2,
              class BinaryPredicate >
    bool
    equal( InputIterator1 first1, InputIterator1 last,
           InputIterator2 first2, BinaryPredicate pred );
```

如果两个序列在范围[first,last]内包含的元素都相等，则 equal()返回 true。如果第二序列包含更多的元素，则不会考虑这些元素。如果我们希望保证两个序列完全相等，则需要写：

```
    if ( vec1.size() == vec2.size() &&
         equal( vec1.begin(), vec1.end(), vec2.begin() );
```

或使用该容器的等于操作符，比如 vec1==vec2。如果第二个容器比第一个容器的元素少，算法的迭代过程应该超过其末尾，则运行时刻的行为是未定义的。缺省情况下，底层元素类型的等于操作符用来作比较，第二个版本应用 pred。

```
    #include <algorithm>
    #include <list>
    #include <iostream.h>

    class equal_and_odd{
    public:
         bool
         operator()( int val1, int val2 )
         {
            return ( val1 == val2 &&
                   ( val1 == 0 || val1 % 2 ));
          }
    };

    int main()
    {
        int ia[] =  { 0,1,1,2,3,5,8,13 };
        int ia2[] = { 0,1,1,2,3,5,8,13,21,34 };

        bool res;

        // true: 都等于ia.的长度
        // 生成结果为: int ia[7] equal to int ia2[9]? true.

        res = equal( &ia[0], &ia[7], &ia2[0] );
        cout << "int ia[7] equal to int ia2[9]? "
             << ( res ? "true" : "false" ) << ".\n";

        list< int, allocator > ilist(  ia,  ia+7  );
    list< int, allocator > ilist2( ia2, ia2+9 );

    // 生成结果为: list ilist equal to ilist2? true.

    res = equal( ilist.begin(), ilist.end(), ilist2.begin() );
    cout << "list ilist equal to ilist2? "
         << ( res ? "true" : "false" ) << ".\n";

    // false: 0, 2, 8 不相等，也不是奇数
    // 生成结果为: list ilist equal_and_odd() to ilist2? false.

    res = equal( ilist.begin(), ilist.end(),
                 ilist2.begin(), equal_and_odd() );

    cout << "list ilist equal_and_odd() to ilist2? "
         << ( res ? "true" : "false" ) << ".\n";

    return 0;
}
```

equal_range()

```
template< class ForwardIterator, class Type >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

equal_range()返回一对 iterator，第一个 iterator 表示由 lower_bound()返回的 iterator 值，第二个表示由 upper_bound()返回的 iterator 值，它们的语义描述见相应的算法。例如，已知下面的序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 equal_range()，返回一对 iterator，这两个 iterator 都指向值 22。用值 22 调用 equal_range()，返回一对 iterator，其中 first 指向值 22，second 指向值 23。第一个版本使用底层类型的小于操作符，第二个版本则用 comp 对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream.h>

/* 生成结果为:
   array element sequence after sort:
   12 15 17 19 20 22 23 26 29 35 40 51

   equal_range result of search for value 23:
         *ia_iter.first: 23     *ia_iter.second: 26
   equal_range result of search for absent value 21:
           *ia_iter.first: 22     *ia_iter.second: 22

   vector element sequence after sort:
   51 40 35 29 26 23 22 20 19 17 15 12

   equal_range result of search for value 26:
           *ivec_iter.first: 26   *ivec_iter.second: 23

   equal_range result of search for absent value 21:
           *ivec_iter.first: 20   *ivec_iter.second: 20
*/

int main()
{
    int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
    vector< int, allocator > ivec( ia, ia+12 );
    ostream_iterator< int >  ofile( cout, " " );

    sort( &ia[0], &ia[12] );

    cout << "array element sequence after sort:\n";
    copy( ia, ia+12, ofile ); cout << "\n\n";

    pair< int*,int* > ia_iter;
    ia_iter = equal_range( &ia[0], &ia[12], 23 );

    cout << "equal_range result of search for value 23:\n\t"
         << "*ia_iter.first: "  << *ia_iter.first << "\t"
         << "*ia_iter.second: " << *ia_iter.second << "\n\n";

    ia_iter = equal_range( &ia[0], &ia[12], 21 );

    cout << "equal_range result of search for "
         << "absent value 21:\n\t"
         << "*ia_iter.first: "  << *ia_iter.first << "\t"
         << "*ia_iter.second: " << *ia_iter.second << "\n\n";

    sort( ivec.begin(), ivec.end(), greater<int>() );

    cout << "vector element sequence after sort:\n";
    copy( ivec.begin(), ivec.end(), ofile ); cout << "\n\n";

    typedef vector< int, allocator >::iterator iter_ivec;
    pair< iter_ivec, iter_ivec > ivec_iter;

    ivec_iter = equal_range( ivec.begin(), ivec.end(), 26,
                    greater<int>() );
    cout << "equal_range result of search for value 26:\n\t"
         << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
         << "*ivec_iter.second: " << *ivec_iter.second
         << "\n\n";

    ivec_iter = equal_range( ivec.begin(), ivec.end(), 21,
                greater<int>() );

    cout << "equal_range result of search for "
         << "absent value 21:\n\t"
         << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
         << "*ivec_iter.second: " << *ivec_iter.second
         << "\n\n";
}
```

fill()

```
template< class ForwardIterator, class Type >
void
fill( ForwardIterator first,
      ForwardIterator last, const Type& value );
```

fill ()将 value 的拷贝赋给[first,last)范围内的所有元素：

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/* 结果为：
   original array element sequence:
   0 1 1 2 3 5 8

    array after fill(ia+1,ia+6):
    0 9 9 9 9 9 8

   original list element sequence:
   c eiffel java ada perl

   list after fill(++ibegin,--iend):
   c c++ c++ c++ perl
*/

int main()
{
    const int value = 9;
    int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
    ostream_iterator< int > ofile( cout, " " );

    cout << "original array element sequence:\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    fill( ia+1, ia+6, value );
    cout << "array after fill(ia+1,ia+6):\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    string the_lang( "c++" );
    string langs[5] = { "c", "eiffel", "java", "ada", "perl" };

    list< string, allocator > il( langs, langs+5 );
    ostream_iterator< string > sofile( cout, " " );

    cout << "original list element sequence:\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";

    typedef list<string,allocator>::iterator iterator;

    iterator ibegin = il.begin(), iend = il.end();
    fill( ++ibegin, --iend, the_lang );

    cout << "list after fill(++ibegin,--iend):\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";
}
```

fill_n()

```
template< class ForwardIterator, class Size, class Type >
void
fill_n( ForwardIterator first,
        Size n, const Type& value );
```

fill_n()把 value 的拷贝赋给[first,first+count]范围内的 count 个元素:

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

class print_elements {
public:
   void operator()( string elem ) {
      cout << elem
           << ( _line_cnt++%8 ? " " : "\n\t" );
   }
   static void reset_line_cnt() { _line_cnt = 1; }

private:
   static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* 结果为:
original element sequence of array container:
0 1 1 2 3 5 8

array after fill_n( ia+2, 3, 9 ):
```

```
0 1 9 9 9 5 8

原字符串序列为:
      Stephen closed his eyes to hear his boots
      crush crackling wrack and shells
      sequence after fill_n() applied:
      Stephen closed his xxxxx xxxxx xxxxx xxxxx xxxxx
      xxxxx crackling wrack and shells
*/

int main()
{
   int value = 9; int count = 3;
   int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
   ostream_iterator< int > iofile( cout, " " );

   cout << "original element sequence of array container:\n";
   copy( ia, ia+7, iofile ); cout << "\n\n";

   fill_n( ia+2, count, value );

   cout << "array after fill_n( ia+2, 3, 9 ):\n";
   copy( ia, ia+7, iofile ); cout << "\n\n";

   string replacement( "xxxxx" );
   string sa[] = { "Stephen", "closed", "his", "eyes", "to",
        "hear", "his", "boots", "crush", "crackling",
        "wrack", "and", "shells" };

   vector< string, allocator > svec( sa, sa+13 );

   cout << "original sequence of strings:\n\t";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n\n";

   fill_n( svec.begin()+3, count*2, replacement );

   print_elements::reset_line_cnt();

   cout << "sequence after fill_n() applied:\n\t";
   for_each( svec.begin(), svec.end(), print_elements() );
   cout << "\n";
}
```

find()

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

find()利用底层元素类型的等于操作符，对[first,last]范围内的元素与 value 进行比较。当发现匹配时，结束搜索过程，且 find()返回指向该元素的一个 InputIterator。如果没有发现匹配，则返回 last；

```
#include <algorithm>
#include <iostream.h>
#include <list>
#include <string>

int main()
{
    int array[ 17 ] = { 7,3,3,7,6,5,8,7,2,1,3,8,7,3,8,4,3 };

    int elem = array[ 9 ];
    int *found_it;

    found_it = find( &array[0], &array[17], elem );

    // 结果: find the first occurrence of 1 found!
    cout << "find the first occurrence of "
         << elem << "\t"
         << ( found_it ? "found!\n" : "not found!\n" );

    string beethoven[] = {
       "Sonata31", "Sonata32", "Quartet14", "Quartet15",
       "Archduke", "Symphony7" };

    string s_elem( beethoven[ 1 ] );

    list< string, allocator > slist( beethoven, beethoven+6 );
    list< string, allocator >::iterator iter;

    iter = find( slist.begin(), slist.end(), s_elem );

    // 结果: find the first occurrence of Sonata32  found!

    cout << "find the first occurrence of "
         << s_elem << "\t"
         << ( iter != slist.end() ? "found!\n" : "not found!\n" );
}
```

find_if()

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

依次检查[first,last]范围内的元素，并把 **pred** 应用在这些元素上面。如果 **pred** 计算结果为 **true**，则搜索过程结束。**find_if()**返回指向该元素的 **InputIterator**。如果没有找到匹配，则返回 **last**：

```
#include <algorithm>
#include <list>
#include <set>
#include <string>
#include <iostream.h>

// 提供另一种等于操作符
// 如字符串包含在成员对象的
// 友元集中返回true
class OurFriends {
public:
   bool operator()( const string& str ) {
        return ( friendset.count( str ));
   }

   static void
   FriendSet( const string *fs, int count ) {
        copy( fs, fs+count,
             inserter( friendset, friendset.end() ));
   }

private:
   static set< string, less<string>, allocator > friendset;
};

set< string, less<string>, allocator > OurFriends::friendset;

int main()
{
    string Pooh_friends[] = { "Piglet", "Tigger", "Eyeore"  };
    string more_friends[] = { "Quasimodo", "Chip", "Piglet" };
    list<string,allocator> lf( more_friends, more_friends+3 );

    // 生成pooh_friends列表
    OurFriends::FriendSet( Pooh_friends, 3 );

    list<string,allocator>::iterator our_mutual_friend;
    our_mutual_friend =
          find_if( lf.begin(), lf.end(), OurFriends());

    // 结果:
    //   Ah, imagine our friend Piglet is also a friend of Pooh.
    if ( our_mutual_friend != lf.end() )
         cout << "Ah, imagine our friend "
              << *our_mutual_friend
              << " is also a friend of Pooh.\n";

    return 0;
}
```

find_end()

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

在由[first,last]标记的序列中查找“由 iterator 对[first2,last2]标记的第二个序列”的最后一次出现。例如，已知字符序列 mississippi 和第二个序列 ss，则 find_end()返回一个 ForwardIterator，指向第二个 ss 序列的第一个 s。如果在第一个序列中没有找到第二个序列，则返回 last1。在第一个版本中，使用底层的等于操作符。在第二个版本中，使用用户传递进来的二元操作 pred：

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

int main()
{
    int array[ 17 ]   = { 7,3,3,7,6,5,8,7,2,1,3,7,6,3,8,4,3 };
    int subarray[ 3 ] = { 3, 7, 6 };

    int *found_it;

    // 在数组中查找最后一次出现的3,7,6序列
    // 返回首元素的地址...

    found_it = find_end( &array[0], &array[17],
                         &subarray[0], &subarray[3] );

    assert( found_it == &array[10] );

    vector< int, allocator > ivec( array, array+17 );
    vector< int, allocator > subvec( subarray, subarray+3 );

    vector< int, allocator >::iterator found_it2;
    found_it2 = find_end( ivec.begin(), ivec.end(),
                          subvec.begin(), subvec.end(),
                          equal_to<int>() );

    assert( found_it2 == ivec.begin()+10 );

    cout << "ok: find_end correctly returned beginning of "
         << "last matching sequence: 3,7,6!\n";
}
```

find_first_of()

```
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 );
```

由[first2,last2)标记的序列包含了一组元素的集合，find_first_of()将在由[first1,last1)标记的序列中搜索这些元素。例如，假设我们希望在字符序列 synesthesia 中找到第一个元音。为了做到这一点，我们把第二个序列定义为 aeiou。find_first_of()返回一个 ForwardIterator，指向元音序列中的元素的第一个出现，本例中，指向第一个 e。如果第一个序列不含有第二个序列中的任何元素，则返回 last1。在第一个版本中，使用底层元素类型的等于操作符。在第二个版本中，使用二元操作 pred:

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

int main()
{
   string s_array[] = { "Ee", "eE", "ee", "Oo", "oo", "ee" };
   string to_find[] = { "oo", "gg", "ee" };

   // 返回第一次出现的"ee" -- &s_array[2]
   string *found_it =
      find_first_of( s_array, s_array+6,
                     to_find, to_find+3 );

   // 结果:
   // found it: ee
   //         &s_array[2]:    0x7fff2dac
   //         &found_it:      0x7fff2dac

   if ( found_it != &s_array[6] )
        cout << "found it: "      << *found_it   << "\n\t"
             << "&s_array[2]:\t" << &s_array[2] << "\n\t"
             << "&found_it:\t"    << found_it    << "\n\n";

   vector< string, allocator > svec( s_array, s_array+6);
   vector< string, allocator > svec_find( to_find, to_find+3 );

   // 返回找到的"oo" -- svec.end()-2
   vector< string, allocator >::iterator found_it2;

   found_it2 = find_first_of(
               svec.begin(), svec.end(),
               svec_find.begin(), svec_find.end(),
               equal_to<string>() );

   // 结果:
   // found it, too: oo
   //         &svec.end()-2:  0x100067b0
   //         &found_it2:     0x100067b0

   if ( found_it2 != svec.end() )
        cout << "found it, too: "  << *found_it2   << "\n\t"
             << "&svec.end()-2:\t" << svec.end()-2 << "\n\t"
             << "&found_it2:\t"    << found_it2    << "\n";
}
```

for_each()

```
template< class InputIterator, class Function >
Function
for_each( InputIterator first,
          InputIterator last, Function func );
```

for_each()依次对[first,last]范围内的所有元素应用函数 func，func 不能对元素执行写操作（因为前两个参数都是 InputIterator，所以不能保证支持赋值操作）。如果我们希望修改元素，则应该使用 transform()算法。func 可以返回值，但是该值会被忽略：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
   vector< int, allocator > ivec;

   for ( int ix = 0; ix < 10; ix++ )
         ivec.push_back( ix );

   void (*pfi)( int ) = print_elements;
   for_each( ivec.begin(), ivec.end(), pfi );

   return 0;
}
```

generate()

```
template< class ForwardIterator, class Generator >
void
generate( ForwardIterator first,
          ForwardIterator last, Generator gen );
```

generate()通过对 gen 的连续调用，来填充一个序列的[first,last]范围。gen 可以是函数对象或函数指针：

```
#include <algorithm>
#include <list>
#include <iostream.h>

int odd_by_twos() {
   static int seed = -1;
   return seed += 2;
 }

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }
```

```
int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;

    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // 结果:
    // elements within list the first invocation:
    // 1 3 5 7 9 11 13 15 17 19

    cout << "elements within list the first invocation:\n";
    for_each( ilist.begin(), ilist.end(), pfi );

    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // 结果:
    // elements within list the second iteration:
    // 21 23 25 27 29 31 33 35 37 39
    cout << "\n\nelements within list the second iteration:\n";
    for_each( ilist.begin(), ilist.end(), pfi );

    return 0;
}
```

generate_n()

```
template< class ForwardIterator,
          class Size, class Generator >
void
generate_n( OutputIterator first, Size n, Generator gen );
```

generate_n()通过对 gen 的 n 次连续调用，来填充一个序列中从 first 开始的 n 个元素。gen 可以是函数对象或函数指针：

```
#include <algorithm>
#include <iostream.h>
#include <list>

class even_by_twos {
public:
    even_by_twos( int seed = 0 ) : _seed( seed ){}
    int operator()() { return _seed += 2; }
private:
    int _seed;
};

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;
```

```
        generate_n( ilist.begin(), ilist.size(), even_by_twos() );

        // 结果:
        // generate_n with even_by_twos():
        // 2 4 6 8 10 12 14 16 18 20

        cout << "generate_n with even_by_twos():\n";
        for_each( ilist.begin(), ilist.end(), pfi ); cout << "\n";

        generate_n(ilist.begin(),ilist.size(),even_by_twos(100));

        // 结果:
        // generate_n with even_by_twos( 100 ):
        // 102 104 106 108 110 112 114 116 118 120

        cout << "generate_n with even_by_twos( 100 ):\n";
        for_each( ilist.begin(), ilist.end(), pfi );
    }
```

includes()

```
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 );
```

includes()判断[first1,last1)的每一个元素是否被包含在序列[first2,last2)中。第一个版本假设这两个序列是用底层元素类型的小于操作符排序的，第二个版本用 comp 来判定元素顺序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
   int ia1[] = { 13, 1, 21, 2, 0, 34, 5, 1, 8, 3, 21, 34 };
   int ia2[] = { 21, 2, 8, 3, 5, 1 };

   // includes必须传入已排序的容器
   sort( ia1, ia1+12 ); sort( ia2, ia2+6 );

   // 结果: every element of ia2 contained in ia1? true

   bool res = includes( ia1, ia1+12, ia2, ia2+6 );
   cout << "every element of ia2 contained in ia1? "
        << (res ? "true" : "false") << endl;

   vector< int, allocator > ivect1( ia1, ia1+12 );
   vector< int, allocator > ivect2( ia2, ia2+6 );
```

```
    // 按降序排序
    sort( ivect1.begin(), ivect1.end(), greater<int>() );
    sort( ivect2.begin(), ivect2.end(), greater<int>() );

    res = includes( ivect1.begin(), ivect1.end(),
                    ivect2.begin(), ivect2.end(),
                    greater<int>() );

    // 结果:
    // every element of ivect2 contained in ivect1? true

    cout << "every element of ivect2 contained in ivect1? "
         << (res ? "true" : "false") << endl;
}
```

inner_product()

```
template < class InputIterator1, class InputIterator2,
           class Type >
Type
inner_product(
   InputIterator1 first1, InputIterator1 last,
   InputIterator2 first2, Type init );
template < class InputIterator1, class InputIterator2,
           class Type,
           class BinaryOperation1, class BinaryOperation2 >
Type
inner_product(
   InputIterator1 first1, InputIterator1 last,
   InputIterator2 first2, Type init,
   BinaryOperation1 op1, BinaryOperation2 op2 );
```

inner_product()的第一个版本对两个序列做内积（对应的元素相乘，再求和），并将内积加到一个由 init 指定的初始值上。第一个序列由[first1,last)标记，第二个序列由 first2 开始，随着第一个序列而逐渐递增。例如，已知序列{2,3,5,8}和{1,2,3,4}，则下列乘积对的和就是结果：

```
2*1 + 3*2 + 5*3 + 8*4
```

如果提供初始值 0，则结果是 55。

第二个版本用二元操作 op1 代替缺省的加法操作，用二元操作 op2 代替缺省的乘法操作。例如，如果同样用上两个序列，指定 op1 为减法，op2 为加法，则结果是下列加法对的差：

```
(2+1) - (3+2) - (5+3) - (8+4)
```

inner_product()是一个算术算法。要使用它，必须包含头文件<numeric>：

```
#include <numeric>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] =  { 2, 3, 5, 8 };
    int ia2[] = { 1, 2, 3, 4 };

    // 两个数组的元素两两相乘,
    // 并将结果添加到初始值: 0

    int res = inner_product( &ia[0], &ia[4], &ia2[0], 0);

    // 结果: inner product of arrays: 55
    cout << "inner product of arrays: "
         << res << endl;

    vector<int, allocator> vec(  ia,  ia+4 );
    vector<int, allocator> vec2( ia2, ia2+4 );

    // 两个向量中的元素相加
    // 并从初始值中减去和: 0

    res = inner_product( vec.begin(), vec.end(),
                         vec2.begin(), 0,
                         minus<int>(), plus<int>() );

    // 结果: inner product of vectors: -28
    cout << "inner product of vectors: "
         << res << endl;

    return 0;
}
```

inplace_merge()

```
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 );
```

inplace_merge()合并两个排过序的连续序列，分别由[first,middle]和[middle,last]标记。结果序列覆盖了由 first 开始的这两段范围。第一个版本使用底层类型的小于操作符对元素进行排序，第二个版本根据程序员传递的二元比较操作对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

/*
 * 结果:
```

```
ia sorted into two subarrays:
12 15 17 20 23 26 29 35 40 51 10 16 21 41 44 54 62 65 71 74

ia inplace_merge:
10 12 15 16 17 20 21 23 26 29 35 40 41 44 51 54 62 65 71 74

ivec sorted into two subvectors:
51 40 35 29 26 23 20 17 15 12 74 71 65 62 54 44 41 21 16 10

ivec inplace_merge:
74 71 65 62 54 51 44 41 40 35 29 26 23 21 20 17 16 15 12 10
*/

int main()
{
   int ia[] = { 29,23,20,17,15,26,51,12,35,40,
                74,16,54,21,44,62,10,41,65,71 };

   vector< int, allocator > ivec( ia, ia+20 );
   void (*pfi)( int ) = print_elements;

   // 以一定排序排列两上子序列
   sort( &ia[0], &ia[10] );
   sort( &ia[10], &ia[20] );

   cout << "ia sorted into two sub-arrays: \n";
   for_each( ia, ia+20, pfi ); cout << "\n\n";

   inplace_merge( ia, ia+10, ia+20 );

   cout << "ia inplace_merge:\n";
   for_each( ia, ia+20, pfi ); cout << "\n\n";

   sort( ivec.begin(),    ivec.begin()+10, greater<int>() );
   sort( ivec.begin()+10, ivec.end(),      greater<int>() );

   cout << "ivec sorted into two sub-vectors: \n";
   for_each( ivec.begin(), ivec.end(), pfi ); cout << "\n\n";

   inplace_merge( ivec.begin(), ivec.begin()+10,
                  ivec.end(),   greater<int>() );

   cout << "ivec inplace_merge:\n";
   for_each( ivec.begin(), ivec.end(), pfi ); cout << endl;
}
```

iter_swap ()

```
template <class ForwardIterator1, class ForwardIterator2>
void
iter_swap ( ForwardIterator1 a, ForwardIterator2 b );
```

iter_swap()交换由两个 ForwardIterator：a 和 b 所指向的元素中的值。

```
#include <algorithm>
```

```
#include <list>
#include <iostream.h>

int main()
{
    int ia[]  = { 5, 4, 3, 2, 1, 0 };
    list< int,allocator > ilist( ia, ia+6 );

    typedef list< int, allocator >::iterator iterator;
    iterator iter1 = ilist.begin(), iter2,
        iter_end = ilist.end();

    // 对列表进行冒泡排序 ...
    for ( ; iter1 != iter_end; ++iter1 )
            for ( iter2 = iter1; iter2 != iter_end; ++iter2 )
                  if ( *iter2 < *iter1 )
                       iter_swap( iter1, iter2 );

    // 输出结果为:
    // ilist after bubble sort using iter_swap():
    // { 0 1 2 3 4 5 }

    cout << "ilist afer bubble sort using iter_swap(): { ";
    for ( iter1 = ilist.begin(); iter1 != iter_end; ++iter1 )
        cout << *iter1 << " ";
    cout << "}\n";
}
```

lexicographical_compare()

```
template <class InputIterator1, class InputIterator2 >
bool
lexicographical_compare(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2 );
template < class InputIterator1, class InputIterator2,
            class Compare >
bool
lexicographical_compare(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    Compare comp );
```

lexicographical_compare()比较由[first1,last1]和[first2,last2]标识的两个序列的对应元素对。比较操作将一直进行下去，直到某个元素对不匹配，或者到达[last1,last2]对，或者到达last1 或 last2（如果两个序列长度不等）。对于第一个不匹配的元素对，发生以下事情：

- 如果第一个序列的元素小，则返回 true，否则返回 false。
- 如果到达 last1，而 last2 未到，则返回 true。
- 如果到达 last2，而未到达 last1，则返回 false。
- 如果 last1 和 last2 都已到达（所有元素都匹配），则返回 false。即第一个序列在字典序上不小于第二个序列。

例如，已知下列两个序列：

```
string arr1[] = { "Piglet", "Pooh", "Tigger" };
string arr2[] = { "Piglet", "Pooch", "Eeyore" };
```

本算法在第一个元素对上匹配，但是在第二个上不匹配，Pooh 大于 Pooch，因为 c 在字典序上小于 h（想像一下字典中的单词是如何排序的）。算法在这一点上停止（不再比较第三个元素），比较的结果是 false。

算法的第二个版本使用了比较对象，而不再使用底层元素类型的小于操作符。

```
#include <algorithm>
#include <list>
#include <string>
#include <assert.h>
#include <iostream.h>

class size_compare {
public:
   bool operator()( const string &a, const string &b ) {
        return a.length() <= b.length();
   }
};

int main()
{
   string arr1[] = { "Piglet", "Pooh", "Tigger" };
   string arr2[] = { "Piglet", "Pooch", "Eeyore" };

   bool res;

   // 第二个元素值为false
   // Pooch 小于 Pooh
   // 第三个元素值也为false

   res = lexicographical_compare( arr1, arr1+3,
                                  arr2, arr2+3 );

   assert( res == false );

   // 值为true: ilist2每个元素的
   // 长度都小于或等于
   // 对应的ilist1的元素

   list< string, allocator > ilist1( arr1, arr1+3 );
   list< string, allocator > ilist2( arr2, arr2+3 );

  res = lexicographical_compare(
            ilist1.begin(), ilist1.end(),
            ilist2.begin(), ilist2.end(), size_compare() );

   assert( res == true );

   cout << "ok: lexicographical_compare succeeded!\n";
}
```

lower_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );
template< class ForwardIterator, class Type, class Compare >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

lower_bound()返回一个 iterator，它指向在[first,last]标记的有序序列中可以插入 value、而不会破坏容器顺序的第一个位置，而这个位置标记了一个大于等于 value 的值。例如，已知下列序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 lower_bound()，返回一个指向值 22 的 iterator。用值 22 调用 lower_bound()，也返回一个指向值 22 的 iterator。第一个版本使用底层类型的小于操作符，第二个版本根据 comp 对元素进行排序和比较。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
   int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
   sort( &ia[0], &ia[12] );

   int search_value = 18;
   int *ptr = lower_bound( ia, ia+12, search_value );

   // 结果:
   // The first element 18 can be inserted in front of is 19
   // The previous value is 17

   cout << "The first element "
        << search_value
        << " can be inserted in front of is "
        << *ptr << endl
        << "The previous value is "
        << *(ptr-1) << endl;

   vector< int, allocator > ivec( ia, ia+12 );

   // 降序排序...
   sort( ivec.begin(), ivec.end(), greater<int>() );

   search_value = 26;
   vector< int, allocator >::iterator iter;

   // 告诉它这里所用的
```

```
    // 正确的排序关系...

    iter = lower_bound( ivec.begin(), ivec.end(),
                        search_value, greater<int>() );

    // 结果:
    // The first element 26 can be inserted in front of is 26
    // The previous value is 29

    cout << "The first element "
         << search_value
         << " can be inserted in front of is "
         << *iter << endl
         << "The previous value is "
         << *(iter-1) << endl;
}
```

max()

```
template< class Type >
const Type&
max( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
max( const Type &aval, const Type &bval, Compare comp );
```

max()返回 aval 和 bval 两个元素中较大的一个。第一个版本使用与 Type 相关联的大于操作符，第二个版本使用比较操作 comp:

max_element()

```
template< class ForwardIterator >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last, Compare comp );
```

max_element()返回一个 iterator，指向[first,last]序列中值为最大的元素。第一个版本使用底层元素类型的大于操作符，第二个版本使用比较操作 comp:

min()

```
template< class Type >
const Type&
min( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
min( const Type &aval, const Type &bval, Compare comp );
```

min()返回 aval 和 bval 两个元素中较小的一个。第一个版本使用与 Type 相关联的小于操

作符，第二个版本使用比较操作 comp:

min_element()

```
template< class ForwardIterator >
ForwardIterator
min_element( ForwardIterator first,
             ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
min_element( ForwardIterator first,
             ForwardIterator last, Compare comp );
```

min_element()返回一个 iterator，指向[first,last]序列中值为最小的元素。第一个版本使用底层元素类型的小于操作符，第二个版本使用比较操作 comp:

```
// 说明 max(), min(), max_element(), min_element()的用法

#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] = { 7, 5, 2, 4, 3 };
    const vector< int, allocator > ivec( ia, ia+5 );

    int mval = max( max( max( max( ivec[4], ivec[3]),
                                 ivec[2]),ivec[1]),ivec[0]);

    // 输出: the result of nested invocations of max() is: 7
    cout << "the result of nested invocations of max() is: "
         << mval << endl;

    mval = min( min( min( min( ivec[4], ivec[3]),
                             ivec[2]),ivec[1]),ivec[0]);

    // 输出: the result of nested invocations of min() is: 2
    cout << "the result of nested invocations of min() is: "
         << mval << endl;

    vector< int, allocator >::const_iterator iter;
    iter = max_element( ivec.begin(), ivec.end() );

    // 输出: the result of invoking max_element() is also: 7
    cout << "the result of invoking max_element() is also: "
         << *iter << endl;

    iter = min_element( ivec.begin(), ivec.end() );

   // 输出: the result of invoking min_element() is also: 2
    cout << "the result of invoking min_element() is also: "
         << *iter << endl;
}
```

merge()

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
      OutputIterator result );
template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result, Compare comp );
```

merge()把两个分别由[first1,last1]和[first2,last2]标记的有序序列，合并到一个从 result 开始的单个序列中，并返回一个 OutputIterator，指向新序列中最后一个元素的下一位置。第一个版本使用底层类型的小于操作符对元素进行排序，第二个版本根据 comp 对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <list>
#include <deque>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
   int ia[] =  {29,23,20,22,17,15,26,51,19,12,35,40};
   int ia2[] = {74,16,39,54,21,44,62,10,27,41,65,71};

   vector< int, allocator > vec1( ia,  ia +12 ),
                            vec2( ia2, ia2+12 );
   int ia_result[24];
   vector<int,allocator> vec_result(vec1.size()+vec2.size());

   sort( ia,  ia +12 );
   sort( ia2, ia2+12 );

   // 输出:
   // 10 12 15 16 17 19 20 21 22 23 26 27 29 35
   //              39 40 41 44 51 54 62 65 71 74

   merge( ia, ia+12, ia2, ia2+12, ia_result );
   for_each( ia_result, ia_result+24, pfi ); cout << "\n\n";

   sort( vec1.begin(), vec1.end(), greater<int>() );
   sort( vec2.begin(), vec2.end(), greater<int>() );

   merge( vec1.begin(), vec1.end(),
            vec2.begin(), vec2.end(),
            vec_result.begin(), greater<int>() );

     // 输出:
     // 74 71 65 62 54 51 44 41 40 39 35 29 27 26 23 22
     //                          21 20 19 17 16 15 12 10
     for_each( vec_result.begin(), vec_result.end(), pfi );
     cout << "\n\n";
}
```

mismatch()

```
template< class InputIterator1, class InputIterator2 >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1,
          InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1, InputIterator1 last,
          InputIterator2 first2, BinaryPredicate pred );
```

mismatch()并行地比较两个序列，指出第一个“元素不匹配”的位置。它返回一对 iterator，标识出第一个元素不匹配的位置。如果所有的元素都匹配，则返回指向每个容器 last 元素的 iterator。例如，已知序列 meet 和 meat，则两个被返回的 iterator 分别指向第三个元素。缺省情况下，用等于操作符对元素进行比较。第二个版本允许用户指定一个比较操作。如果第二个序列比第一个序列的元素多，这些元素将被忽略。如果第二个序列比第一个序列的元素少，则运行时刻的行为是未定义的：

```
#include <algorithm>
#include <list>
#include <utility>
#include <iostream.h>

class equal_and_odd{
public:
    bool operator()( int ival1, int ival2 )
    {
         // 两个值相等吗，或
         // 都为0或都为奇数
          return ( ival1 == ival2 &&
                 ( ival1 == 0 || ival1%2 ));
    }
};

int main()
{
    int ia[] =  { 0,1,1,2,3,5,8,13 };
    int ia2[] = { 0,1,1,2,4,6,10   };

    pair<int*,int*> pair_ia = mismatch( ia, ia+7, ia2 );
    // 输出: first mismatched pair: ia: 3 and ia2: 4
    cout << "first mismatched pair: ia: "
         << *pair_ia.first << " and ia2: "
         << *pair_ia.second << endl;

    list<int,allocator> ilist(  ia,  ia+7  );
    list<int,allocator> ilist2( ia2, ia2+7 );

    typedef list<int,allocator>::iterator iter;
    pair< iter,iter > pair_ilist =
       mismatch( ilist.begin(),  ilist.end(),
                 ilist2.begin(), equal_and_odd() );

    // 输出:
    // first mismatched pair either not equal or not odd:
    //                  ilist: 2 and ilist2: 2

    cout << "first mismatched pair either not equal "
         << "or not odd: \n\tilist: "
         << *pair_ilist.first << " and ilist2: "
         << *pair_ilist.second << endl;
}
```

next_permutation()

```
template< class BidirectionalIterator >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );
template< class BidirectionalIterator, class Compare >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, Compare comp );
```

next_permutation()取出由[first,last]标记的排列，并将其重新排序为下一个排列（关于怎样确定上一个排列的讨论见 12.5.4 节）。如果不存在下一个排列，则返回 false。否则，返回 true。第一个版本使用底层类型的小于操作符来确定下一个排列，第二个版本根据 comp 对元素进行排序。如果原始字符串是排过序的，则连续调用 next_permutation()会生成整个排列集合。例如，在下列程序中，如果我们不能把 musil 排序成 ilmsu，则不能生成排列的全集：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

void print_char( char elem ) { cout << elem ; }
void (*ppc)( char ) = print_char;

/* 输出:
ilmsu   ilmus   ilsmu   ilsum   ilums   ilusm   imlsu   imlus
imslu   imsul   imuls   imusl   islmu   islum   ismlu   ismul
isulm   isuml   iulms   iulsm   iumls   iumsl   iuslm   iusml
limsu   limus   lismu   lisum   liums   liusm   lmisu   lmius
lmsiu   lmsui   lmuis   lmusi   lsimu   lsium   lsmiu   lsmui
lsuim   lsumi   luims   luism   lumis   lumsi   lusim   lusmi
```

```
    milsu   milus   mislu   misul   miuls   miusl   mlisu   mlius
    mlsiu   mlsui   mluis   mlusi   msilu   msiul   msliu   mslui
    msuil   msuli   muils   muisl   mulis   mulsi   musil   musli
    silmu   silum   simlu   simul   siulm   siuml   slimu   slium
    slmiu   slmui   sluim   slumi   smilu   smiul   smliu   smlui
    smuil   smuli   suilm   suiml   sulim   sulmi   sumil   sumli
    uilms   uilsm   uimls   uimsl   uislm   uisml   ulims   ulism
    ulmis   ulmsi   ulsim   ulsmi   umils   umisl   umlis   umlsi
    umsil   umsli   usilm   usiml   uslim   uslmi   usmil   usmli
    */

    int main()
    {
        vector<char,allocator> vec(5);

        // 字符顺序: musil
        vec[0] = 'm'; vec[1] = 'u'; vec[2] = 's';
        vec[3] = 'i'; vec[4] = 'l';

        int cnt = 2;
        sort( vec.begin(), vec.end() );
        for_each( vec.begin(), vec.end(), ppc ); cout << "\t";

        // 生成"musil"的所有排列组合
        while( next_permutation( vec.begin(), vec.end()))
        {
            for_each( vec.begin(), vec.end(), ppc );
            cout << "\t";

            if ( ! ( cnt++ % 8 )) {
                 cout << "\n";
                 cnt = 1;
            }
        }

        cout << "\n\n";
        return 0;
    }
```

nth_element()

```
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 );
```

nth_element()将[first,last]标记的序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的元素出现在它后面。例如，已知数组：

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

下面的 nth_element()调用使第七个元素为第 n 个（它的值是 26）：

```
nth_element( &ia[0], &ia[6], &ia[12] );
```

产生一个序列，其中小于 26 的七个元素在它的左边，余下大于 26 的四个元素在它的右边：{23,20,22,17,15,19,12,51,35,40,29}，但是，第 n 个元素两边的元素并不保证存在某种特定的顺序。第一个版本使用底层类型的小于操作符。第二个版本根据程序员传递的二元比较操作，对元素调整顺序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
original order of the vector: 29 23 20 22 17 15 26 51 19 12 35 40
sorting vector based on element 26
12 15 17 19 20 22 23 26 51 29 35 40
sorting vector in descending order based on element 23
40 35 29 51 26 23 22 20 19 17 15 12
*/

int main()
{
   int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
   vector< int,allocator > vec( ia, ia+12 );
   ostream_iterator<int> out( cout," " );

   cout << "original order of the vector: ";
   copy( vec.begin(), vec.end(), out ); cout << endl;

   cout << "sorting vector based on element "
        << *( vec.begin()+6 ) << endl;

   nth_element( vec.begin(), vec.begin()+6, vec.end() );
   copy( vec.begin(), vec.end(), out ); cout << endl;

   cout << "sorting vector in descending order "
        << "based on element "
        << *( vec.begin()+6 ) << endl;

   nth_element( vec.begin(), vec.begin()+6,
                vec.end(),   greater<int>() );

   copy( vec.begin(), vec.end(), out ); cout << endl;
}
```

partial_sort()

```
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 );
```

partial_sort()对整个序列作部分排序，被排序元素的个数正好可以被放到 [first,middle] 范围内。在[middle,last]中的元素是未经排序的，它们都落在实际被排序的序列之外。例如，已知数组：

```
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

调用 partial_sort()，使第六个元素为 middle：

```
    stable_sort( &ia[0], &ia[5], &ia[12] );
```

则产生了一个序列，其中五个最小的元素被排序（即 middle-first 个元素）：{12,15,17,19,20,29,23,22,26,51,35,40}。从 middle 到 last-1 的元素并没有按任何特定的顺序，但是它们的值都落在实际被排序的序列之外。第一个版本用底层类型的小于操作符，第二个版本根据 comp 对元素进行排序：

partial_sort_copy()

```
    template< class InputIterator, class RandomAccessIterator >
    RandomAccessIterator
    partial_sort_copy( InputIterator first, InputIterator last,
                       RandomAccessIterator result_first,
                       RandomAccessIterator result_last );
    template< class InputIterator, class RandomAccessIterator,
              class Compare >
    RandomAccessIterator
    partial_sort_copy( InputIterator first, InputIterator last,
                       RandomAccessIterator result_first,
                       RandomAccessIterator result_last,
                       Compare comp );
```

partial_sort_copy()的行为与 partial_sort()相同，只不过它把经过部分排序的序列拷贝到由[result_first,result_last)标记的容器中。因此，如果我们指定了一个独立的容器去接受拷贝，则结果是一个完全排序的序列。例如，已知两个数组：

```
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
    int ia2[5];
```

指定第八个元素为 middle 的 partial_sort_copy()调用：

```
    stable_sort( &ia[0],  &ia[7], &ia[12],
                 &ia2[0], &ia[5] );
```

用五个排过序的元素填充 ia2：{12,15,17,19,20}，而另外两个排过序的元素没有被使用：

```
    #include <algorithm>
    #include <vector>
    #include <iostream.h>
    /*
```

```
 * 输出：
   original order of vector: 69 23 80 42 17 15 26 51 19 12 35 8
   partial sort of vector: seven elements
   8 12 15 17 19 23 26 80 69 51 42 35
   partial_sort_copy() of first seven elements
   of vector in descending order
   26 23 19 17 15 12 8
 */

int main()
{
   int ia[] = {69,23,80,42,17,15,26,51,19,12,35,8 };
   vector< int,allocator > vec( ia, ia+12 );
   ostream_iterator<int> out( cout," " );

   cout << "original order of vector: ";
   copy( vec.begin(), vec.end(), out ); cout << endl;

   cout << "partial sort of vector: seven elements\n";
   partial_sort( vec.begin(), vec.begin()+7, vec.end() );
   copy( vec.begin(), vec.end(), out ); cout << endl;

   vector< int, allocator > res(7);
   cout << "partial_sort_copy() of first seven elements\n\t"
        << "of vector in descending order\n";

   partial_sort_copy( vec.begin(), vec.begin()+7, res.begin(),
                      res.end(), greater<int>() );
   copy( res.begin(), res.end(), out ); cout << endl;
}
```

partial_sum()

```
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 op );
```

partial_sum()的第一个版本创建一个新的元素序列，其中每个新元素的值代表了[first,last]序列中该位置之前（包括该位置）所有元素的和。例如，已知序列{0,1,1,2,3,5,8}，则新序列是{0,1,2,4,7,12,20}。例如，第四个元素是前三个值{0,1,1}的部分和加上它自己（2），产生值 4。

第二个版本使用程序员传递的二元操作。例如，已知序列{1,2,3,4}，我们传递函数对象times<int>。结果序列是{1,2,6,24}。在两个版本中，OutputIterator 指向新序列末元素的下一个位置。

partial_sum()是一个算术算法，我们必须包含标准头文件<numeric>：

```
#include <numeric>
#include <vector>
#include <iostream.h>

/*
 * 输出：
   elements: 1 3 4 5 7 8 9
   partial sum of elements:
   1 4 8 13 20 28 37
   partial sum of elements using times<int>():
   1 3 12 60 420 3360 30240
*/

int main()
{
    const int ia_size = 7;
    int ia[ ia_size ] = { 1, 3, 4, 5, 7, 8, 9 };
    int ia_res[ ia_size ];

    ostream_iterator< int  > outfile( cout, " "  );
    vector< int, allocator > vec( ia, ia+ia_size );
    vector< int, allocator > vec_res( vec.size() );

    cout << "elements: ";
    copy( ia, ia+ia_size, outfile ); cout << endl;

    cout << "partial sum of elements:\n";
    partial_sum( ia, ia+ia_size, ia_res );
    copy( ia_res, ia_res+ia_size, outfile ); cout << endl;

    cout << "partial sum of elements using times<int>():\n";
    partial_sum( vec.begin(), vec.end(), vec_res.begin(),
                 times<int>() );

    copy( vec_res.begin(), vec_res.end(), outfile );
    cout << endl;
}
```

partition()

```
template < class BidirectionalIterator, class UnaryPredicate >
BidirectionalIterator
partition( BidirectionalIterator first,
           BidirectionalIterator last, UnaryPredicate pred );
```

partition()对[first,last]范围内的元素重新排序。当向它传递一个一元谓词操作 pred 时，所有计算结果为 true 的元素都被放在计算结果为 false 的元素前面。例如，已知序列{0,1,2,3,4,5,6}，以及一个“测试元素是否为偶数”的一元谓词操作，则 true 和 false 的元素范围分别是{0,2,4,6}和{1,3,5}。虽然所有的偶元素保证放在奇元素的前面，但是，结果序列并不保证保留元素的相对位置。即，4 可能放在 2 的前面，或者 5 放在 3 之前。后面讨论的 stable_partition()会保证保留容器内元素的相对顺序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

class even_elem {
public:
    bool operator()( int elem )
         { return elem%2 ? false : true; }
};

/*
 * 输出:
  original order of elements:
  29 23 20 22 17 15 26 51 19 12 35 40
  partition based on whether element is even:
  40 12 20 22 26 15 17 51 19 23 35 29
  partition based on whether element is less than 25:
  12 23 20 22 17 15 19 51 26 29 35 40
 */

int main()
{
    const int ia_size = 12;
    int ia[ia_size]   = { 29,23,20,22,17,15,26,51,19,12,35,40 };

    vector< int, allocator > vec( ia, ia+ia_size );
    ostream_iterator< int >  outfile( cout, " "  );

    cout << "original order of elements: \n";
    copy( vec.begin(), vec.end(), outfile ); cout << endl;

    cout << "partition based on whether element is even:\n";
    partition( &ia[0], &ia[ia_size], even_elem() );
    copy( ia, ia+ia_size, outfile ); cout << endl;

   cout << "partition based on whether element "
        << "is less than 25:\n";

    partition( vec.begin(), vec.end(), bind2nd(less<int>(),25) );
    copy( vec.begin(), vec.end(), outfile ); cout << endl;
}
```

prev_permutation()

```
template < class BidirectionalIterator >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );
template < class BidirectionalIterator, class Compare >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, Compare comp );
```

prev_permutation 取出由[first,last]标记的排列，并将它重新排序为上一个排列（关于怎样判断上一个排列的讨论见 12.5.4 节）。如果不存在上一个排列，则返回 **false**；否则，返回 true。

第一个版本使用底层类型的小于操作符，来确定上一个排列，第二个版本根据程序员传递的二元比较操作，对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

// 输出:
//   n d a   n a d   d n a   d a n   a n d   a d n

int main()
{
    vector< char, allocator > vec( 3 );
    ostream_iterator< char > out_stream( cout, " " );

    vec[0] = 'n'; vec[1] = 'd'; vec[2] = 'a';
    copy( vec.begin(), vec.end(), out_stream ); cout << "\t";

    // 生成"dan"的所有排列
    while( prev_permutation( vec.begin(), vec.end() )) {
           copy( vec.begin(), vec.end(), out_stream );
           cout << "\t";
    }

    cout << "\n\n";
}
```

random_shuffle()

```
template< class RandomAccessIterator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last );
template< class RandomAccessIterator,
          class RandomNumberGenerator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last,
                RandomNumberGenerator rand );
```

random_shuffle()对[first,last)范围内的元素随机调整顺序。第二个版本使用一个专门产生随机数的函数对象或函数指针。rand 返回一个 double 类型的、位于区间[0,1]内的值。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    vector< int, allocator > vec;
    for ( int ix = 0; ix < 20; ix++ )
          vec.push_back( ix );

    random_shuffle( vec.begin(), vec.end() );
    // 输出:
    // random_shuffle of sequence of values 1 .. 20:
    // 6 11 9 2 18 12 17 7 0 15 4 8 10 5 1 19 13 3 14 16
    cout << "random_shuffle of sequence of values 1 .. 20:\n";
    copy( vec.begin(), vec.end(),
          ostream_iterator< int >( cout," " ));
}
```

remove()

```
template< class ForwardIterator, class Type >
ForwardIterator
remove( ForwardIterator first,
        ForwardIterator last, const Type &value );
```

remove()删除在[first,last)范围内的所有 value 实例。remove ()以及 remove_if()并不真正地把匹配到的元素从容器中清除（即容器的大小保留不变），而是每个不匹配的元素依次被赋值给从 first 开始的下一个空闲位置上，返回的 ForwardIterator 标记了新的元素范围的下一个位置。例如，考虑序列{0,1,0,2,0,3,0,4,0}。假设我们希望删除所有的 0，则结果序列是{1,2,3,4,0,3,0,4}。1 被拷贝到第一个位置上，2 被拷贝到第二个位置上，3 被拷贝到第三个位置上，4 被拷贝到第四个位置上。返回的 ForwardIterator 指向第五个位置上的 0。典型的做法是，该 iterator 接着被传递给 erase()，以便删除无效的元素。（内置数组不适合于使用 remove()和 remove_if()算法，因为它们不能很容易地被改变大小。由于这个原因，对于数组而言，remove_copy()和 remove_copy_if()是更受欢迎的算法。）

remove_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
remove_copy( InputIterator first, InputIterator last,
             OutputIterator result, const Type &value );
```

remove_copy()把所有不匹配的元素都拷贝到由 result 指定的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，但原始容器没有被改变：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* 输出:
   original vector sequence:
   0 1 0 2 0 3 0 4 0 5
   vector after remove, without applying erase():
   1 2 3 4 5 3 0 4 0 5
   vector after erase():
   1 2 3 4 5
     array after remove_copy():
   1 2 3 4 5
 */
```

```
int main()
{
    int value = 0;
    int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

    vector< int, allocator > vec( ia, ia+10 );
    ostream_iterator< int >  ofile( cout," ");
    vector< int, allocator >::iterator vec_iter;

    cout << "original vector sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vec_iter = remove( vec.begin(), vec.end(), value );

    cout << "vector after remove, without applying erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // 从容器中去除非法元素
    vec.erase( vec_iter, vec.end() );

    cout << "vector after erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    int ia2[5];
    vector< int, allocator > vec2( ia, ia+10 );
    remove_copy( vec2.begin(), vec2.end(), ia2, value );

    cout << "array after remove_copy():\n";
    copy( ia2, ia2+5 ofile ); cout << endl;
}
```

remove_if()

```
template< class ForwardIterator, class Predicate >
ForwardIterator
remove_if( ForwardIterator first,
           ForwardIterator last, Predicate pred );
```

remove_if()删除所有在[first,last]范围内、并且 pred 计算结果为 true 的元素。remove_if()以及 remove()并不真正地把匹配到的元素从容器中清除，而是将每个不匹配的元素依次赋值给从 first 开始的下一个空闲位置上。返回的 ForwardIterator 标记了新的元素范围的下一个位置。一般是将这个 iterator 传递给 erase()，以便真正地删除掉无效的元素。（remove_copy_if()更加适用于内置数组。）

remove_copy_if()

```
template< class InputIterator, class OutputIterator,
          class Predicate >
OutputIterator
remove_copy_if( InputIterator first, InputIterator last,
                OutputIterator result, Predicate pred );
```

remov_copy_if()把所有不匹配的元素拷贝到由 result 指定的容器中。返回的 OutputIterator

标记了被拷贝的末元素的下一个位置，原始容器没有被改变：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* 输出：
   original element sequence:
   0 1 1 2 3 5 8 13 21 34
   sequence after applying remove_if < 10:
   13 21 34
   sequence after applying remove_copy_if even:
   1 1 3 5 13 21
*/

class EvenValue {
public:
    bool operator()( int value ) {
           return value % 2 ? false : true; }
};

int main()
{
    int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

    vector< int, allocator >::iterator iter;
    vector< int, allocator > vec( ia, ia+10 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    iter = remove_if( vec.begin(), vec.end(),
                      bind2nd(less<int>(),10) );
    vec.erase( iter, vec.end() );

    cout << "sequence after applying remove_if < 10:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vector< int, allocator > vec_res( 10 );
    iter = remove_copy_if( ia, ia+10,
                           vec_res.begin(), EvenValue() );

    cout << "sequence after applying remove_copy_if even:\n";
    copy( vec_res.begin(), iter, ofile ); cout << '\n';
}
```

replace()

```
template< class ForwardIterator, class Type >
void
replace( ForwardIterator first, ForwardIterator last,
         const Type& old_value, const Type& new_value );
```

replace()将[first,last)范围内的所有 old_value 实例都用 new_value 替代。

replace_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
replace_copy( InputIterator first, InputIterator last,
              OutputIterator result,
              const Type& old_value, const Type& new_value );
```

replace_copy()的行为与 replace()类似，只不过是把新序列拷贝到由 result 开始的容器内。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，但原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* 输出:
  original element sequence:
  Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
  sequence after applying replace():
  Christopher Robin Pooh Piglet Tigger Eeyore
  sequence after applying replace_copy():
  Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
 */

int main()
{
   string oldval( "Mr. Winnie the Pooh" );
   string newval( "Pooh" );

   ostream_iterator< string >  ofile( cout, " " );
   string sa[] = {
      "Christopher Robin", "Mr. Winnie the Pooh",
      "Piglet", "Tigger", "Eeyore"
   };

   vector< string, allocator > vec( sa, sa+5 );

   cout << "original element sequence:\n";
   copy( vec.begin(), vec.end(), ofile ); cout << '\n';

   replace( vec.begin(), vec.end(), oldval, newval );

   cout << "sequence after applying replace():\n";
   copy( vec.begin(), vec.end(), ofile ); cout << '\n';

   vector< string, allocator > vec2;
   replace_copy( vec.begin(), vec.end(),
                 inserter( vec2, vec2.begin() ),
                 newval, oldval );

   cout << "sequence after applying replace_copy():\n";
   copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
}
```

replace_if()

```
template< class ForwardIterator, class Predicate, class Type >
void
replace_if( ForwardIterator first, ForwardIterator last,
            Predicate pred, const Type& new_value );
```

replace_if()将[frist,last)范围内的、pred 计算结果为 true 的所有元素，都用 new_value 替代。

replace_copy_if()

```
template< class ForwardIterator, class OutputIterator,
          class Predicate, class Type >
OutputIterator
replace_copy_if( ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 Predicate pred, const Type& new_value );
```

replace_copy()的行为与 replace_if()类似，只不过是把新序列拷贝到由 result 开始的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
   original element sequence:
   0 1 1 2 3 5 8 13 21 34
   sequence after applying replace_if < 10 with 0:
   0 0 0 0 0 0 0 13 21 34
   sequence after applying replace_if even with 0:
   0 1 1 0 3 5 0 13 21 0
 */

class EvenValue {
public:
   bool operator()( int value ) {
        return value % 2 ? false : true; }
};

int main()
{
   int new_value = 0;

   int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
   vector< int, allocator > vec( ia, ia+10 );
   ostream_iterator< int >  ofile( cout, " " );

   cout << "original element sequence:\n";
   copy( ia, ia+10, ofile ); cout << '\n';

   replace_if( &ia[0], &ia[10],
                        bind2nd(less<int>(),10), new_value );

            cout << "sequence after applying replace_if < 10 with 0:\n";
            copy( ia, ia+10, ofile ); cout << '\n';

            replace_if( vec.begin(), vec.end(),
                        EvenValue(), new_value );

            cout << "sequence after applying replace_if even with 0:\n";
            copy( vec.begin(), vec.end(), ofile ); cout << '\n';
        }
```

reverse()

```
        template< class BidirectionalIterator >
        void
        reverse( BidirectionalIterator first,
                 BidirectionalIterator last );
```

reverse()对于容器中[first,last)范围内的元素重新按反序排列。例如，已知序列{0,1,1,2,3}，则反序序列是{3,2,1,1,0}。

reverse_copy()

```
        template< class BidirectionalIterator, class OutputIterator >
        OutputIterator
        reverse_copy( BidirectionalIterator first,
                      BidirectionalIterator last, OutputIterator
result );
```

reverse_copy()的行为与 **reverse()**类似，只不过把新序列拷贝到由 **result** 开始的容器中。返回的 **OutputIterator** 指向被拷贝的元素的下一个位置。原始序列没有被改变。

```
        #include <algorithm>
        #include <list>
        #include <string>
        #include <iostream.h>

        /*
         * 输出:
          Original sequence of strings:
               Signature of all things I am here to
               read seaspawn and seawrack that rusty boot

          Sequence after reverse() applied:
               boot rusty that seawrack and seaspawn read to
               here am I things all of Signature
         */

        class print_elements {
        public:
            void operator()( string elem ) {
               cout << elem
                    << ( _line_cnt++%8 ? " " : "\n\t" );
```

```
    }

    static void reset_line_cnt() { _line_cnt = 1; }

private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;

int main()
{
    string sa[] = { "Signature", "of", "all", "things",
         "I", "am", "here", "to", "read",
         "seaspawn", "and", "seawrack", "that",
         "rusty", "boot"
    };

    list< string, allocator > slist( sa, sa+15 );

    cout << "Original sequence of strings:\n\t";
    for_each( slist.begin(), slist.end(), print_elements() );
    cout << "\n\n";

    reverse( slist.begin(), slist.end() );

    print_elements::reset_line_cnt();

    cout << "Sequence after reverse() applied:\n\t";
    for_each( slist.begin(), slist.end(), print_elements() );
    cout << "\n";

    list< string, allocator > slist_copy( slist.size() );
    reverse_copy( slist.begin(), slist.end(),
                  slist_copy.begin() );
}
```

rotate()

```
template< class ForwardIterator >
void
rotate( ForwardIterator first,
        ForwardIterator middle, ForwardIterator last );
```

rotate()把[first,middle]范围内的元素移到容器末尾，由 middle 指向的元素成为容器的第一个元素。例如，已知单词“hissboo”，则以元素“b”为轴的旋转将单词变成“boohiss”。

rotate_copy()

```
template< class ForwardIterator, class OutputIterator >
OutputIterator
rotate_copy( ForwardIterator first, ForwardIterator middle,
             ForwardIterator last,  OutputIterator result );
```

rotate_copy()的行为与 rotate()类似，只不过把旋转后的序列拷贝到由 result 标记的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置。原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
   original element sequence:
   1 3 5 7 9 0 2 4 6 8 10
   rotate on middle element(0) ::
   0 2 4 6 8 10 1 3 5 7 9
   rotate on next to last element(8) ::
   8 10 1 3 5 7 9 0 2 4 6
   rotate_copy on middle element ::
   7 9 0 2 4 6 8 10 1 3 5
 */

int main()
{
    int ia[] = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10 };

    vector< int, allocator > vec( ia, ia+11 );
    ostream_iterator< int > ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    rotate( &ia[0], &ia[5], &ia[11] );

    cout << "rotate on middle element(0) ::\n";
    copy( ia, ia+11, ofile ); cout << '\n';

    rotate( vec.begin(), vec.end()-2, vec.end() );

    cout << "rotate on next to last element(8) ::\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vector< int, allocator > vec_res( vec.size() );

    rotate_copy( vec.begin(), vec.begin()+vec.size()/2,
                 vec.end(), vec_res.begin() );

    cout << "rotate_copy on middle element ::\n";
    copy( vec_res.begin(), vec_res.end(), ofile );
    cout << '\n';
}
```

search()

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

        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred );
```

给出了两个范围，search()返回一个 iterator，指向在[first1,last1]范围内第一次出现子序列[first2,last2]的位置。如果子序列未出现，则返回 last1。例如，在 mississippi 中，子序列 iss 出现两次，则 search()返回一个 iterator，指向第一个实例的起始处。缺省情况下，使用等于操作符进行元素的比较，第二个版本允许用户提供一个比较操作：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
   Expecting to find the substring 'ate': a t e
   Expecting to find the substring 'vat': v a t
 */

int main()
{
   ostream_iterator< char > ofile( cout, " " );

   char str[ 25 ] = "a fine and private place";
   char substr[]  = "ate";

   char *found_str = search(str,str+25,substr,substr+3);

   cout << "Expecting to find the substring 'ate': ";
   copy( found_str, found_str+3, ofile ); cout << '\n';

   vector< char, allocator > vec( str, str+24 );
   vector< char, allocator > subvec(3);

   subvec[0]='v'; subvec[1]='a'; subvec[2]='t';

   vector< char, allocator >::iterator iter;
   iter = search( vec.begin(), vec.end(),
                  subvec.begin(), subvec.end(),
                  equal_to< char >() );

   cout << "Expecting to find the substring 'vat': ";
   copy( iter, iter+3, ofile ); cout << '\n';
}
```

search_n()

```
template< class ForwardIterator, class Size, class Type >
```

```
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value );
template< class ForwardIterator, class Size,
          class Type, class BinaryPredicate >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value, BinaryPredicate pred );
```

search_n()在[first,last]序列中查找“value 出现 count 次”的子序列。如果没有找到“value 的 count 次出现”，则返回 last。例如，为了在序列 mississippi 中找到子序列 ss，value 将被设置为“s”，而 count 为 2。为了找到子串“ssi”的两个实例，value 应该为“ssi”，而 count 仍是 2。search_n()返回一个 iterator，指向被找到的 value 的第一个元素。缺省情况下，使用等于操作符来比较元素，第二版本允许用户提供一个比较操作。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出：
   Expecting to find two instances of 'o': o o
   Expecting to find the substring 'mou':  m o u
 */

int main()
{
   ostream_iterator< char >  ofile( cout, " " );

   const char blank = ' ';
   const char oh    = 'o';

   char str[ 26 ]  = "oh my a mouse ate a moose";
   char *found_str = search_n( str, str+25, 2, oh );

   cout << "Expecting to find two instances of 'o': ";
   copy( found_str, found_str+2, ofile ); cout << '\n';

   vector< char, allocator > vec( str, str+25 );

   // 寻找第一个这样的序列
   // 其中三个字符都不是空格: mouse 中的 mou

   vector< char, allocator >::iterator iter;
   iter = search_n( vec.begin(), vec.end(), 3,
                    blank, not_equal_to< char >() );

   cout << "Expecting to find the substring 'mou':  ";
   copy( iter, iter+3, ofile ); cout << '\n';
}
```

set_difference()

```
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 );
```

set_difference()构造一个排过序的序列，其中的元素出现在第一个序列中（由[first,last]标记），但是不包含在第二个序列中（由[first2,last2]标记）。例如，已知两个序列{0,1,2,3}和{0,2,4,6}，则差集为{1,3}。返回的 OutputIterator 指向被放入 result 所标记的容器中的最后元素的下一个位置。第一个版本假设该序列是用底层元素类型的小于操作符来排序的，第二个版本假设该序列是用 comp 来排序的。

set_intersection()

```
template < class InputIterator1, class InputIterator2,
                class OutputIterator >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result );
template < class InputIterator1, class InputIterator2,
                class OutputIterator, class Compare >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp );
```

set_intersection()构造一个排过序的序列，其中的元素在［first1,last1］和[first2,last2]序列中都存在。例如，已知序列{0,1,2,3}和{0,2,4,6}，则交集为{0,2}。这些元素被从第一个序列中拷贝出来。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

set_symmetric_difference()

```
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 );
```

set_symmetric_difference()构造一个排过序的序列，其中的元素在第一个序列中出现、但不出现在第二个序列中，或者在第二个序列中出现、但不出现在第一个序列中。例如，已知两个序列{0,1,2,3}和{0,2,4,6}，则对称差集是{1,3,4,6}。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

set_union()

```
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 );
```

set_union()构造一个排过序的序列，它包含了[first1,last1)和[first2,last2)这两个范围内的所有元素。例如，已知两个序列{0,1,2,3}和{0,2,4,6}，则并集为{0,1,2,3,4,6}。如果一个元素在两个容器中都存在，比如 0 和 2，则拷贝第一个容器中的元素。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

```
#include <algorithm>
#include <set>
#include <string>
#include <iostream.h>

/*
 * 输出:
   set #1 elements:
       Eeyore Piglet Pooh Tigger

   set #2 elements:
       Heffalump Pooh Woozles

   set_union() elements:
       Eeyore Heffalump Piglet Pooh Tigger Woozles

   set_intersection() elements:
       Pooh

   set_difference() elements:
       Eeyore Piglet Tigger
```

```
  set_symmetric_difference() elements:
      Eeyore Heffalump Piglet Tigger Woozles
*/

int main()
{
    string str1[] = { "Pooh", "Piglet", "Tigger", "Eeyore" };
    string str2[] = { "Pooh", "Heffalump", "Woozles" };
    ostream_iterator< string >  ofile( cout, " " );

    set<string,less<string>,allocator> set1( str1, str1+4 );
    set<string,less<string>,allocator> set2( str2, str2+3 );

    cout << "set #1 elements:\n\t";
    copy( set1.begin(), set1.end(), ofile ); cout << "\n\n";
    cout << "set #2 elements:\n\t";
    copy( set2.begin(), set2.end(), ofile ); cout << "\n\n";

    set<string,less<string>,allocator> res;
    set_union( set1.begin(), set1.end(),
               set2.begin(), set2.end(),
               inserter( res, res.begin() ));

    cout << "set_union() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_intersection( set1.begin(), set1.end(),
                      set2.begin(), set2.end(),
                      inserter( res, res.begin() ));

    cout << "set_intersection() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_difference( set1.begin(), set1.end(),
                    set2.begin(), set2.end(),
                    inserter( res, res.begin() ));

    cout << "set_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_symmetric_difference( set1.begin(), set1.end(),
                              set2.begin(), set2.end(),
                              inserter( res, res.begin() ));

    cout << "set_symmetric_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
}
```

sort()

```
template< class RandomAccessIterator >
```

```
void
sort( RandomAccessIterator first,
      RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
sort( RandomAccessIterator first,
      RandomAccessIterator last, Compare comp );
```

sort()利用底层元素的小于操作符，以升序重新排列[first,last)范围内的元素。第二版本根据 comp 对元素进行排序（为了保留相等元素之间的顺序关系，要使用 stable_sort()，而不是 sort()）。我们不提供专门的程序来说明 sort()的用法，因为它在许多其他的例子中会被用到，比如 binary_search()、equal_range()和 inplace_merge()。

stable_partition()

```
template< class BidirectionalIterator, class Predicate >
BidirectionalIterator
stable_partition( BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred );
```

stable_partition()的行为与 partition()类似，只不过它保证会保留容器中元素的相对顺序。下面是与 partition()的例子相同的一个程序，但是它被修改为调用 stable_partition():

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
  original element sequence:
  29 23 20 22 17 15 26 51 19 12 35 40
  stable_partition on even element:
  20 22 26 12 40 29 23 17 15 51 19
  stable_partition of less than 25:
  23 20 22 17 15 19 12 29 26 51 35 40
 */

class even_elem {
public:
   bool operator()( int elem ) {
        return elem%2 ? false : true;
   }
};

int main()
{
   int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
   vector< int, allocator > vec( ia, ia+12 );
   ostream_iterator< int >  ofile( cout, " " );

   cout << "original element sequence:\n";
   copy( vec.begin(), vec.end(), ofile ); cout << '\n';
```

```
    stable_partition( &ia[0], &ia[12], even_elem() );

    cout << "stable_partition on even element:\n";
    copy( ia, ia+11, ofile ); cout << '\n';

    stable_partition( vec.begin(), vec.end(),
                      bind2nd(less<int>(),25)  );

    cout << "stable_partition of less than 25:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

stable_sort()

```
template< class RandomAccessIterator >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last, Compare comp );
```

stable_sort()利用底层类型的小于操作符，以升序重新排列[first,last)范围内的元素，并且保留相等元素之间的顺序关系。第二版本根据 comp 对元素进行排序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
  original element sequence:
  29 23 20 22 12 17 15 26 51 19 12 23 35 40
  stable sort -- default ascending order:
  12 12 15 17 19 20 22 23 23 26 29 35 40 51
  stable sort: descending order:
  51 40 35 29 26 23 23 22 20 19 17 15 12 12
 */

int main()
{
    int ia[] = { 29,23,20,22,12,17,15,26,51,19,12,23,35,40 };
    vector< int, allocator > vec( ia, ia+14 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    stable_sort( &ia[0], &ia[14] );

    cout << "stable sort -- default ascending order:\n";
    copy( ia, ia+14, ofile ); cout << '\n';

    stable_sort( vec.begin(), vec.end(), greater<int>() );
```

```
        cout << "stable sort: descending order:\n";
        copy( vec.begin(), vec.end(), ofile ); cout << '\n';
    }
```

swap()

```
template< class Type >
void
swap ( Type &ob1, Type &ob2 );
```

swap()交换存贮在对象 ob1 和 ob2 中的值。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
   original element sequence:
   3 4 5 0 1 2
   sequence applying swap() to support bubble sort:
   0 1 2 3 4 5
 */

int main()
{
    int ia[]  = { 3, 4, 5, 0, 1, 2 };
    vector< int, allocator > vec( ia, ia+6 );

    for ( int ix = 0; ix < 6; ++ix )
    for ( int iy = ix; iy < 6; ++iy ) {
    if ( vec[iy] < vec[ ix ] )
    swap( vec[iy], vec[ix] );
    }

    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( ia, ia+6, ofile ); cout << '\n';

  cout << "sequence applying swap() "
       << "to support bubble sort:\n";

    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

swap_range()

```
template <class ForwardIterator1, class ForwardIterator2 >
ForwardIterator2
swap_range( ForwardIterator1 first1, ForwardIterator1 last,
            ForwardIterator2 first2 );
```

swap_range()将[first1,last)标记的元素值与“从 first2 开始、相同个数”的元素值进行交

换。这两个序列可以是同一容器中不相连的序列，也可以位于两个独立的容器中。如果从 first2 开始的序列小于由[first1,last)标记的序列，或者两个序列在同一容器中有重叠，则该算法的运行时刻行为是未定义的。swap_range()返回第二个序列的 iterator，指向最后一个被交换的元素的下一个位置。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * 输出:
  original element sequence of first container:
  0 1 2 3 4 5 6 7 8 9
  original element sequence of second container:
  5 6 7 8 9
  array after swap_ranges() in middle of array:
  5 6 7 8 9 0 1 2 3 4
  first container after swap_ranges() of two vectors:
  5 6 7 8 9 5 6 7 8 9
  second container after swap_ranges() of two vectors:
  0 1 2 3 4
 */

int main()
{
    int ia[]  = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int ia2[] = { 5, 6, 7, 8, 9 };

    vector< int, allocator > vec( ia, ia+10 );
    vector< int, allocator > vec2( ia2, ia2+5 );

    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence of first container:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    cout << "original element sequence of second container:\n";
    copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';

    // 在同一序列中进行交换
    swap_ranges( &ia[0], &ia[5], &ia[5] );

    cout << "array after swap_ranges() in middle of array:\n";
    copy( ia, ia+10, ofile ); cout << '\n';

    // 跨容器交换
    vector< int, allocator >::iterator last =
           find( vec.begin(), vec.end(), 5 );

    swap_ranges( vec.begin(), last, vec2.begin() );

    cout << "first container after "
         << "swap_ranges() of two vectors:\n";
```

```
        copy( vec.begin(), vec.end(), ofile ); cout << '\n';

        cout << "second container after "
             << "swap_ranges() of two vectors:\n";

        copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
    }
```

transform()

```
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 last,
           InputIterator2 first2, OutputIterator result,
           BinaryOperation bop );
```

transform()的第一个版本将 op 作用在[first,last)范围内的每个元素上，从而产生一个新的序列。例如，已知序列{0,1,1,2,3,5}和函数对象 Double（它使每个元素加倍），那么，结果序列是{0,2,2,4,6,10}。

第二个版本将 bop 作用在一对元素上，其中一个元素来自序列[first1,last)，另一个来自由 first2 开始的序列，最终产生一个新的序列。如果第二个序列包含的元素少于第一个序列，则运行时刻行为是未定义的。例如，已知序列{1,3,5,9}和{2,4,6,8}，以及函数对象 AddAndDouble（它把两个元素相加，并将和加倍），则结果序列是{6,14,22,34}。

两个版本的 transform()都把结果序列放在由 result 标记的容器中。result 可以指向两个输入容器之一，则实际达到的效果是，用 transform()返回的元素取代当前的元素。返回的 OutputIterator 指向最后被赋给 result 的元素的下一个位置。

```
#include <algorithm>
#include <vector>
#include <math.h>
#include <iostream.h>

/*
 * 输出:
   original array values: 3 5 8 13 21
   transform each element by doubling: 6 10 16 26 42
   transform each element by difference: 3 5 8 13 21
 */

int double_val( int val ) { return val + val; }
int difference( int val1, int val2 ) {
    return abs( val1 - val2 ); }

int main()
{
    int ia[]  = { 3, 5, 8, 13, 21 };
    vector<int, allocator> vec( 5 );
    ostream_iterator<int> outfile( cout, " " );

    cout << "original array values: ";
    copy( ia, ia+5, outfile ); cout << endl;

    cout << "transform each element by doubling: ";
    transform( ia, ia+5, vec.begin(), double_val );
    copy( vec.begin(), vec.end(), outfile ); cout << endl;

    cout << "transform each element by difference: ";
    transform( ia, ia+5, vec.begin(), outfile, difference );
    cout << endl;
}
```

unique()

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

对于连续的元素，如果它们包含相同的值（使用底层类型的等于操作符来判断），或者把它们传给 pred 的计算结果都为 true，则这些元素被折叠成一个元素。例如，在单词 mississippi 中，语义上的结果是 misisipi。注意，四个 i 不是连续的，所以不会被折叠。类似地，因为两对 s 也是不连续的，所以也没有被折叠成单个实例。为了保证所有重复的实例都被折叠起来，我们必须先对容器进行排序。

实际上，unique()的行为有些不太直观，类似于 remove()算法。在这两种情况下，容器的实际大小并没有变化，每个惟一的元素都被依次拷贝到从 first 开始的下一个空闲位置上。

因此，在我们的例子中，实际的结果是 misisipippi，这里的 ppi 字符序列可以说是算法的残留物。返回的 ForwordIterator 指向残留物的起始处。典型的做法是，这个 iterator 被传递给 erase()，以便删除无效的元素。（由于内置数组不支持 erase()操作，所以 unique()不太适合于数组；unique_copy()对数组更为合适一些。）

unique_copy()

```
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 pred );
```

unique_copy()把每组“含有相同的值（使用底层类型的等于操作符来判断）”或“被传递给 pred 时计算结果为 true（描述见 unique()）”的连续元素，拷贝一个实例。为了保证所有重复的元素都被清除掉，必须先对容器进行排序，返回的 OutputIterator 指向目标容器的尾部。

```
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;
void (*pfs)( string ) = print_elements;

int main()
{
   int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

   vector<int,allocator> vec( ia, ia+10 );
   vector<int,allocator>::iterator vec_iter;

   // 生成不能交换的序列：没有连续的0
   // 结果: 0 1 0 2 0 3 0 4 0 5
   vec_iter = unique( vec.begin(), vec.end() );
   for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

   // 排了序的向量: 0 0 0 0 0 1 2 3 4 5
   // 应用unique()后:
   // 结果: 0 1 2 3 4 5 2 3 4 5

   sort( vec.begin(), vec.end() );
   vec_iter = unique( vec.begin(), vec.end() );
   for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

   // 从容器中删除无效元素
   // 结果: 0 1 2 3 4 5

  vec.erase( vec_iter, vec.end() );
  for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

   string sa[] = { "enough", "is", "enough",
                   "enough", "is", "good"
   };

   vector<string,allocator> svec( sa, sa+6 );
   vector<string,allocator> vec_result( svec.size() );
   vector<string,allocator>::iterator svec_iter;

   sort( svec.begin(), svec.end() );
   svec_iter = unique_copy( svec.begin(), svec.end(),
                            vec_result.begin() );
```

```
        // 结果: enough good is
        for_each( vec_result.begin(), svec_iter, pfs );
        cout << "\n\n";
    }
```

upper_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

upper_bound()返回一个 iterator，它指向在[first,last]标记的有序序列中可以插入 value、而不会破坏容器顺序的最后一个位置。这个位置标记了一个大于 value 的值。例如，已知序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 upper_bound (),返回一个指向值 22 的 iterator。用值 22 调用 upper_bound (),则返回一个指向值 23 的 iterator。第一个版本使用底层类型的小于操作符，而第二个版本根据 comp 对元素进行排序和比较。

```
#include <algorithm>
#include <vector>
#include <assert.h>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    vector<int,allocator> vec(ia,ia+12);

    sort(ia,ia+12);
    int *iter = upper_bound(ia,ia+12,19);
    assert( *iter == 20 );

    sort( vec.begin(), vec.end(), greater<int>() );
    vector<int,allocator>::iterator iter_vec;

    iter_vec = upper_bound( vec.begin(), vec.end(),
                            27, greater<int>() );
```

```
        assert( *iter_vec == 26 );

        // 结果: 51 40 35 29 27 26 23 22 20 19 17 15 12
        vec.insert( iter_vec, 27 );
        for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
    }
```

堆算法

标准库提供的 heap（堆）是一个 max-heap。所谓 max-heap 是一个用数组表示的二叉树，它的每个节点上的键值大于或等于其儿子节点的键值（完整的讨论见[SEDGEWIOK88]）。（另外一种表示是 min-heap，其中每个节点的键值小于或等于其儿子节点的键值。）在标准库的表示中，最大的键值（可以把它想像成树的根）总是在数组的开始处。例如，以下的字母序列满足堆的要求：

```
满足堆要求的字母序列
X T O G S M N A E R A I
```

在这个例子中，X 是根节点，有一个左儿子 T 和右儿子 O。注意，两个儿子之间的顺序是不要求的（即，左儿子不必小于右儿子）。G 和 S 是 T 的儿子，而 M 和 N 是 O 的儿子。类似地，A 和 E 是 G 的儿子，R 和 A 是 S 的儿子，I 是 M 的左儿子，而 N 是叶节点，没有儿子。

四个泛型堆算法：make_heap()、pop_heap()、push_heap()和 sort_heap()为堆的创建和操纵提供了支持。后三个算法假定：由 iterator 对标记的序列代表了一个真正的堆（如果该序列不是一个堆的话，则算法的运行时刻行为是未定义的）。注意，list 容器不能被用作堆，因为它不支持随机访问。内置数组可以被用来支持一个堆，但是 pop_heap()和 push_heap()算法难以与数组一起使用，因为这两个算法要求改变数组的大小。我们先简要介绍这四个算法，然后用一个小程序说明它们的用法。

make_heap()

```
template< class RandomAccessIterator >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

make_heap()把[first,last)范围内的元素做成一个堆。双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

pop_heap()

```
template< class RandomAccessIterator >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last );
```

```
template< class RandomAccessIterator, class Compare >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last, Compare comp );
```

pop_heap()并不真正地把最大元素从堆中弹出，而是重新排序堆。它把 first 和 last-1 交换，然后将[first,last-1]范围的序列重新做成一个堆。之后，我们就可以用容器的成员操作 back()，来访问“被弹出”的元素，或者用 pop_back()将它真正删除掉。双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

push_heap()

```
template< class RandomAccessIterator >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

push_haep()假设由[first,last-1]标记的序列是一个有效的堆，要被加到堆中的新元素在位置 last-1 上，它将[first,last]序列重新做成一个堆。在调用 push_heap()之前，我们必须先把元素插入到容器的后面，或许可以使用 push_back()操作符（这将在下一个程序例子中说明）。双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

sort_heap()

```
template< class RandomAccessIterator >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

sort_heap()对范围[first,last]中的序列进行排序，它假设该序列是一个有效的堆（否则，它的行为是未定义的）。（当然，经过排序之后的堆就不再是一个有效的堆！）双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
   int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

```
    vector< int, allocator > vec( ia, ia+12 );


    // 结果: 51 35 40 23 29 20 26 22 19 12 17 15
    make_heap( &ia[0], &ia[12] );
    void (*pfi)( int ) = print_elements;
    for_each( ia, ia+12, pfi ); cout << "\n\n";

    // 结果: 12 17 15 19 23 20 26 51 22 29 35 40
    // 一个min-heap: 根是最小的元素

    make_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 结果: 12 15 17 19 20 22 23 26 29 35 40 51
    sort_heap( ia, ia+12 );
    for_each(  ia, ia+12, pfi ); cout << "\n\n";

    // 再加一个新的最小的元素:
    vec.push_back( 8 );

    // 结果: 8 17 12 19 23 15 26 51 22 29 35 40 20
    // 将最新最小的元素放在根处

    push_heap( vec.begin(), vec.end(), greater<int>() );
    for_each(  vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 结果: 12 17 15 19 23 20 26 51 22 29 35 40 8
    // 应用次最小的元素替代最小的

    pop_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
}
```

英汉对照索引

凡例

1．本索引共分三级：第一级为加粗显示，第二级和第三级各自缩进一级。其中的意义第二级应加上第一级才完整。第三级则应该加上前两级才完整。

2．每级索引结构相同，分成英文，中文，多个索引项以及参见部分。索引项中，冒号前面是章节号（附录用 A 表示），冒号后的是页码。章节号加粗的表示义项为该章节的主题。

符号

& (ampersand)，
  address-of operator，取地址操作符，2.2:21
    use with function name，用于函数名中，7.9.2:317
  bitwise AND operator，按位与操作符，4.11:136
  reference definition use，用于引用定义中，3.6:86
&= (ampersand equal)，
  compound assignment operator，复合赋值操作符，4.4:126，4.11:136
&& (ampersand-double)，
  logical AND operator，逻辑与操作符，4.2:117，4.3:120–122
< (angle bracket-left，左尖括号)，
  less than operator，小于操作符，2.1:18，4.3:120
  requirement for container element type，容器元素类型必须支持，6.4:220
<= (angle bracket-left equal)，
  less than or equal operator，小于操作符，4.3:120
<< (angle bracket-left-double)，
  bitwise left shift operator，按位左移操作符，4.11:136
  output operator，输出操作符，1.5:15，**20.1**:872–876
    overloading，重载的，**20.4**:891–895
    另参见 iostream
<<= (angle bracket-left-double equal)，
  left shift assign operator，左移赋值操作符，4.4:126
> (angle bracket-right，右尖括号)，
  greater than operator，大于操作符，2.1:18，4.3:120
>= (angle bracket-right equal)，
  greater or equal operator，大于等于操作符，4.3:120
>> (angle bracket-right-double)，
  bitwise shift right operator，按位右移操作符，4.11:136
  input operator，输入操作符，1.5:15，20.2:876–885
    overloading，重载的，**20.5**:895–897
    另参见 iostream
>>= (angle bracket-right-double equal)，
  right shift assign operator，右移赋值操作符，4.4:126
<> (angle bracket，尖括号)，
  explicit template argument specifications，显式指定模板实参，10.4:417
  explicit template specialization，模板显式特化，10.6:424
  include file use，用于包含文件，1.3:10
  template definition use，用于模板定义，10.1:406，16.1:666
*** (asterisk，星号)，**
  defining pointers with，用于定义指针，3.3:72–75
    function pointer，函数指针，7.9.1:316

use in expression，用于表达式中，3.3:72–75，4.1:117
accessing array elements，访问数组元素，3.9.2:97
not required for function invocation，调用函数时不必要，7.9.3:317
multiplication operator，乘法操作符，2.1:18，4.2:118
***= (asterisk equal)，**
multiplication assign operator，乘法赋值操作符，4.4:126
\b (backslash b)，
backspace escape sequence，退格转义序列前缀，3.1:63
\ (backslash，反斜杠)，
as escape sequence operator，用作转义序列操作符，3.1:63，6.9:237
multiline string literal continuation character，多行字符串文字连续符，3.1:63，6.9:237
\a (backslash a)，
alert bell escape sequence，响铃转义序列，3.1:63
\\ (backslash double)，
backslash escape sequence，反斜杠转义序列，3.1:63
\" (backslash double quote)，
double quote escape sequence，双引号转义序列，3.1:63
\f (backslash f)，
formfeed escape sequence，进纸键转义序列，3.1:63
\n (backslash n)，
newline escape sequence，换行符转义序列，3.1:63
\? (backslash question)，
question mark escape sequence，问号转义序列，3.1:63
\r (backslash r)，
carriage return escape sequence，回车键转义序列，3.1:63
\' (backslash single quote)，
single quote escape sequence，单引号转义序列，3.1:63
\v (backslash v)，
vertical tab escape sequence，垂直制表键转义序列，3.1:63
{} (braces，大括号)，
catch clause use，用于catch子句，11.2:454
compound statement delimiters，复合语句边界，5.1:160
linkage directive use，用于链接指示符，7.7:304
multidimensional array initialization use，用于多维数组初始化，3.9.1:96
namespace declaration use，用于名字空间声明，8.5.1:351
try block use，用于try语句块，11.2:452
[，) (bracket-left parenthesis-right)，
containers，left-inclusive interval notation，容器中的左闭合区间记号，12.5:494
[] (bracket，方括号)，
另参见 array，数组
dynamic array allocation with，用于动态数组的分配，8.4.3:345，15.8.1:629
dynamic array deallocation use，用于动态数组的释放，8.4.3:346，15.8.1:630
subscript operator，下标操作符，2.1:19–20，3.9:93–95
bitset use，用于bitset，4.12:140
map use，用于map，6.12: 248
not supported for multisets and multimaps，multiset和multimap不支持，6.15:269
overloaded operator，重载操作符，15.4:618–619
vector use，用于vector，3.10:100–101
: (colon，冒号)，
class derivation designated by，用于指定类的派生，2.4:37，17.1.1:727
member initialization list use，用于成员初始化列表，2.4:38，14.5:588
:: (colon-double，双冒号)，
参见 scope operator，域操作符
，(comma，逗号)，
misuse in array index，在数组索引中的误用，3.9.1:97
operator，操作符，4.10:136
... (ellipsis，省略号)，7.3.6:295–296
catch-all catch clause use，用于catch 的 catch-all 子句，11.3.4:461

function parameter list use，用于函数参数表，7.3.6:295
function pointer definition use，用于指针定义，7.9.1:316
= (equal，等号)，
assignment operator，赋值操作符，3.5:83
lvalue requirement，必须有左值，3.21:67
overloaded operator，重载操作符，14.7:598，15.3:617–618
== (equal-double，双等号)，
equality operator，等于操作符，2.1:18，3.5:83，4.3:120
requirement for container element type，容器元素类型必须支持，6.4:220
! (exclamation，感叹号)，
logical NOT operator，逻辑非操作符，4.3:120–121
!= (exclamation equal)，
inequality operator，不等于操作符，2.1:18，4.3:120
^ (hat，caret，脱字符)，
bitwise XOR operator，按位异或操作符，4.11:136
^= (hat equal)，
compound assignment operator，复合赋值操作符，4.4:126，4.11:136
- (minus，减号)，
subtraction operator，减法操作符，2.1:18，4.2:118
-> (minus angle bracket-right)，
member access operator，成员访问操作符，2.3:25，13.2:511
overloaded operator，重载操作符，**15.6**:620–622
-= (minus equal)，
subtraction assign operator，减法赋值操作符，4.4:126
-- (minus-double，双减号)，
decrement operator，递增操作符，4.5:126
overloaded operator，重载操作符，15.7:622–625
() (parentheses，小括号)，
function call operator，函数调用操作符，7.1:278
overloaded operator，重载操作符，15.5:619–620
overloaded operator for function object，用于函数对象的重载操作符，12.2:472，12.3:481
% (percent，百分号)，
modulus (or remainder) operator，取模（或求余）操作符，4.2:118
%= (percent equal)，
remainder assign operator，求余赋值操作符，4.4:126
. (period，句点)，
member access operator，成员访问操作符，2.3:25，13.2:510
+ (plus，加号)，
addition operator，加法操作符，2.1:18，4.2:118
complex numbers support，复数支持，4.6:128
concatenating strings with，用于字符串连接，3.4.2:80
+= (plus equal)，
addition assignment operator，加法赋值操作符，4.3:121，4.4:126
concatenating strings with，用于字符串连接，3.4.2:80
++ (plus-double，双加号)，
increment operator，递增操作符，4.5:126
overloaded operator，重载操作符，**15.7**:622–625
?: (question colon)，
conditional operator，条件操作符，4.7:131
if-else shorthand use，if-else的简写形式，5.3:169
; (semicolon，分号)，
as definition terminator，用作定义结束符，3.2.3:69
as statement terminator，用作语句结束符，5.1:159
' (single quote，单引号)，
delimiter for literal character constant，字符文字常量的分隔符，3.1:63
/ (slash，斜杠，除号)，
divide operator，除法操作符，2.1:18，4.2:118
/*，*/ (slash asterisk)，
comment pair，注释对，1.4:13
nesting not permitted，不允许嵌套，1.4:14
/= (slash equal)，
division assign operator，除法赋值操作符，4.4:126

// (slash-double，双斜杠)，
comment delimiter，注释分隔符，1.4:14
~ (tilde，颚化符)，
bitwise NOT operator，按位非操作符，4.11:136
destructor identifier，析构函数标识符，2.3:30，14.3:576
| (vertical bar，竖线)，
bitwise OR operator，按位或操作符，4.11:136
|= (vertical bar equal)，
bitwise OR assign operator，按位或赋值操作符，4.4:126，4.11:136
|| (vertical bar-double，双竖线)，
logical OR operator，逻辑或操作符，4.3:120–121
__DATE__，1.3:12
__FILE__，1.3:12
__LINE__，1.3:12
__STDC__，1.3:12
__TIME__，1.3:12

A
abort() function，abort()函数，5.11:190
terminate() 函数的缺省行为，11.3.2:459
access，访问，
base class，基类，18.3:800–806
protected member，保护成员，17.2.1:729
member，成员，**17.3**:736–743
private base class，私有基类，18.3.3:804
class member，类成员，13.1.3:506–508，**13.2.2**:513–514
accumulate() generic algorithm，accumulate() 泛型算法，A:920
adaptor，适配器，
参见 function adapter，函数适配器；function object，函数对象
addition (+) operator，加法操作符，2.1:18，4.2:118
complex number support，复数支持，4.6:128
compound assignment (+=) operator，复合赋值操作符，4.3:121，4.4:126
concatenating strings with，用于字符串连接，3.4.2:80
address，地址，
参见 memory，内存；pointer，指针
address-of (&) operator，取地址操作符，
2.2:21，4.1:117
use with function name，用于函数名，7.9.2:317
use with reference definition，用于引用定义，3.6:86–87
compound assignment (&=) operator，复合赋值操作符，4.4:126，4.11:136
adjacent_difference() generic algorithm，adjacent_difference() 泛型算法，A:921
adjacent_find() generic algorithm，adjacent_find() 泛型算法，A:922
alert bell (\a)，响铃，
as escape sequence，用作转义序列，3.1:63
algorithm header file，algorithm 头文件，2.8:56，12.5:495
algorithm，算法，
参见 generic algorithm，泛型算法
alias，别名，
另参见 reference，引用
namespace，名字空间，2.3:26–32
typedef name，typedef 名字，**3.12**:103–104
allocation，分配，
参见 dynamic memory allocation，动态内存分配
ambiguity，二义性，
function template argument，函数模板实参，10.2:413，10.4:418，10.7:429
overloaded function，重载函数，9.3.3:389–390
overloaded operator，重载操作符，**15.12.3**:661–662
AND (&&) operator，与操作符，4.1:117，4.3:120–123
angle bracket (<>)，尖括号，
参见<>
argc，parameter for main()，main()的参数，**7.8**:306
argument，实参，
另参见 parameter，参数
class template argument，类模板实参，
default，缺省的，16.1:668–669
for non-type parameter，非类型参数的，**16.2.1**:675–678
for type parameter，类型参数的，16.2:671–678
function argument，函数实参，7.1:278

conversion，转换，*参见* conversion，转换
default，缺省的，**7.3.5**:293–295
default and viable function，缺省的和可行的函数，**9.4.4**:403–404
default and virtual function，缺省的和虚拟的函数，**17.5.4**:760–762
pass-by-value，按值传递，7.3:283
passing，传递，7.3:283–296
function template argument，函数模板实参，**10.2**:411–414
deduction of，推演，**10.3**:414–417
explicit specification，显式指定，10.4:417–420
argv array，argv 数组，
command line option access through，命令行选项访问，7.8:306
arithmetic，算术，
另参见 conversion，转换；floating point type，浮点类型；function object，函数对象；integer type，整数类型
conversion，转换，**4.14.2**:148–149
另参见 promotion，提升
bool to int，bool 到 int ，3.7:90
implicit in expression，表达式中的隐式转换，**4.14.1**:147
data type，数据类型，2.1:18
exception，异常，4.2:118
function object，函数对象，**12.3.2**:484
另参见 function object，函数对象
operator，操作符，**4.2**:118–120
complex number support for，复数支持，3.11:103，4.7:131
(table)，(表4.1)，4.2:118
array，数组，3.9:93–99
*另参见*delete expression，delete表达式；dynamic memory allocation，动态内存分配；dynamic memory deallocation，动态内存释放；new expression，new表达式；vector
arrays of references prohibited in，禁止引用数组，3.9:95
assignment with another array prohibited，禁止赋值给另一个数组，3.9:95
auto_ptr prohibited，禁止 auto_ptr ，**8.4.2**:341
built-in abstraction not supported，不支持内置抽象，2.1:20
of class object，类对象的，14.4:581–587
allocated on the heap，在堆中分配，14.4.1:583–585，15.8:626–633
compared with vector，与数组比较，14.4.2:585–587
copying array，复制数组，3.9:95
defining，定义，2.1:19，3.9:93
dynamic allocation and deallocation of，动态分配和释放，**8.4.3**:345–346
of arrays of class，类数组的，14.4.1:583–585，15.8:626–633
function parameter，函数参数，7.3.3:289–291
array-to-pointer conversion，数组到指针的转换，9.3.1:384
as holder of a group of parameter values，用作一组参数值的容器，7.4.1:300
multidimensional，多维的，7.3.3:291
size，not part of parameter type，长度，不是参数类型的一部分，7.3.3:289
of function pointer，函数指针的，7.9.4:318–319
function return type prohibited，禁止函数返回类型，7.2.1:280
generic algorithm support，支持泛型算法，12.1:469
indexing，索引，2.1:19，3.9:93–95
multidimensional，多维的，3.9.1:96–97
range checking not performed for，不进行范围检查，3.9:95
overloaded operator，重载操作符，15.3:618
initialization，初始化，2.1:19，3.9:94–95
for dynamically allocated arrays，动态分配数组的初始化，8.4.3:345–346
for dynamically allocated arrays of class，动态分配的类数组的初始化，14.4:582–585
for multidimensional arrays，多维数组的初始化，3.9.1:96–97
with another array prohibited，禁止用另一个数组进行初始化，3.9:95
iterating through，迭代，遍历
iterator pair use，iterator对的使用 6.5:223–224

through pointer manipulation，通过指针操纵，3.9.2:98
multidimensional，多维的，3.9.1:96–97
pointer type relationship with，与指针类型的关系，3.9.2:97–99
sizeof() use with，使用sizeof()，4.8:132
vector，compared with，与vector的比较，28:54–57，3.10:101
arrow operator (->)，箭头操作符，
参见 member access operator，成员访问操作符
assert() macro，assert()宏，1.3:12
assignment，赋值，
另参见 constructor，构造函数，initialization，初始化
array，with another array prohibited，禁止用另一个数组赋值，3.9:95
auto_ptr behavior，auto_ptr的行为8.4.2:342
class memberwise，按成员赋值，14.7:597–600，17.6:772–776
complex numbers，复数，4.6:128
function pointer，函数指针，7.9.2:317
overloaded function consideration，重载函数的考虑，9.1.7:377
operator，操作符
built-in type，内值类型的，4.4:123–126
compound，复合的，4.4:126
lvalue requirement，必须有右值的要求，3.2.1:67
overloaded，重载的，14.7:597–600，15.3:616–618，17.6:772–776
to a reference，赋值给一个引用，3.6:87
sequence container，序列容器，6.6.2:227
vector，compared with built-in array，向量的赋值，与内置数组的比较，3.10:101
associative container，关联容器，6:209–274
另参见 container type，容器类型
associativity，关联性，
另参见 expression，表达式；precedence，优先级
operator，impact on expression，操作符的，对表达式的影响，4.13:142–146
subexpression evaluation order impact，子表达式计算顺序的影响，4.2:118
atoi() function，函数，7.8:309
auto_ptr class template，auto_ptr 类模板，8.4.2:341–344
initialization，初始化，8.4.2:343
memory header file，memory头文件，8.4.2:341
pitfall，陷阱，8.4.2:344
automatic object，自动对象，8.3.2:336–337
另参见 object，对象
declaration with register，用register声明，8.3.1:335–336
storage property，存储属性，8.3:335

B
back_inserter() function adaptor，函数适配器，
push_back() insert operation use，用于push_back()的插入操作，12.4.1:489
backslash (\)，反斜杠，
参见\
backspace (\b)，退格符，
as escape sequence，用作转义序列，3.1:62
base class，基类，
另参见 class，类；class member，类成员；class scope，类域；constructor，构造函数；derived class，派生类；destructor，析构函数；inheritance，继承
abstract base class，抽象基类17.1.1:724–727，17.5.2:758
access，访问，
protected member，对保护成员的，17.2.1:730
to base class，对基类的，18.3:800–806
to member，对成员的，17.3:736–743
to private base class，对私有基类的，18.3.3:804
to protected base class，对保护基类的，19.3.3:866
assignment，memberwise，按成员赋值，17.6:774–776
constructor，构造函数，17.4:743–750
conversion to base class，转换为基类，17.1.1:724–726
in function overload resolution，在函数重载解析中，19.3.3:864–866
in function template argument deduction，在函数模板实参推演中，10.3:414
defining base class，定义基类

in multiple inheritance，在多继承中，18.2:794–798
in single inheritance，在单继承中，17.2.1:728–732
in virtual inheritance，在虚拟继承中，18.5.1:815–816
destructor，析构函数，**17.4.5**:749–751
virtual destructor，虚拟析构函数，**17.5.5**:763–764
initialization，初始化，
memberwise initialization，按成员初始化，17.6:772–774
multiple inheritance，多继承，18.2:794–795
single inheritance，单继承，17.4:743–751
virtual inheritance，虚拟继承，18.5.2:816–820
member visibility，成员的可见性，
under multiple inheritance，在多继承中，18.4.1:809–817
under single inheritance，在单继承中，18.4:806–809
under virtual inheritance，在虚拟继承中，18.5.4:820–821
virtual base class，虚拟基类，**18.5**:813–823
virtual function，虚拟函数，*参见* virtual function，虚拟函数
best viable function，最佳可行函数，9.2:380，**9.4.3**:399–403
另参见 function overload resolution，函数重载解析
for calls with arguments of class type，15.10.4:648–651
inheritance and，19.3.3:864–866
BidirectionalIterator，12.4.6:493
另参见 iterator，迭代器
binary operator，二元操作符，
参见 operator，操作符
binary_search() generic algorithm，binary_search()泛型算法，A:923
bind1st() function adaptor，bind1st()函数适配器，12.3.5:486
bind2st() function adaptor，bind2st()函数适配器，12.3.5:486
binder，绑定器，
as function adaptor class，用作函数适配器类，12.3.5:486

bit-field，位域，
as a space-saving class member，节省空间的类成员，**13.8**:544–545
bitset class，bitset类，4.11:136
any() function，any()函数，4.12:139
bitset header file，bitset头文件，4.12:139
count() function，count()函数，4.12:139
operation，操作，4.12:139–142
reset()，4.12:139
size() function，size()函数，4.12:139，4.12:140
subscript operator ([])，下标操作符，4.12:140
test() function，test()函数，4.12:139
to_long() function，to_long()函数，4.12:142
to_string() function，to_string()函数，4.12:141
bitset header file，bitset 头文件，4.12:139
bitvector，位向量，4.11:136–137
bitset class compared with，与bitset类的比较，4.11:137
bitwise，按位的，
AND assign (&=) operator，与赋值操作符，4.4:126，4.11:137
AND (&) operator，与操作符，4.11:137
compound assignment operator，复合赋值操作符，4.4:126，4.11:136
NOT (~) operator，非操作符，4.11:137
operator，操作符，4.11:136–138
bitset class support of，bitset 类支持，4.12:141
OR (inclusive or) (|) operator，或操作符，4.11:137
shift operator (<< ，>>)，移位操作符，4.11:137
XOR (exclusive or) (^) operator，异或操作符，4.11:137
block，（语句）块，
comment，注释，1.4:13
function，函数，**7.1:278**
function try block，函数 try 块，11.2:453
constructor and，构造函数和，19.2.7:854–855
statement，语句，5.1:160
try，2.6:46，**11.2**:452–455
bool type，布尔类型，2.1:18，3.7:90–91
operator that evaluate to，计算结果为bool类型的操作符，4.3:120

conversion to, during function overload resolution，在函数重载解析中的转换，9.3.3:388

braces ({ }),
参见{ }

brackets ([ ]),
参见[ ]

break statement，break 语句，5.8:183–184
return statement compared with，与 return 语句的比较，7.4:297
switch statement termination use，用于结束 switch 语句，5.4:172

C

C function，C函数，7.7:304–306
另参见 linkage directive，链接指示符
function pointer to，函数指针，7.9.6:322–323

C-style character string，C风格的字符串，3.4.1:76
access as char*，通过 char*访问，3.4.1:76
C library function，C库函数，3.4.1:76
converting to string object，转换为string对象，3.4.2:97
<cstring>，3.4.1:76
dynamic array allocation for，动态数组分配，8.4.3:345
empty，test for，测试是否为空，3.4.1:77
input/output，输入输出，20.2.1:880
null terminated，以null为终止，3.4.1:77
off-by-one error，一位偏移错误，3.4.1:77
pitfall，易犯的错误，3.4.1:76–78
string type compared with，与string 类型的比较，3.4.1:78
traversal，3.4.1:77

call，调用，
参见 argument，实参；function，函数

candidate function，候选函数，9.4.1:394–397
另参见 function overload resolution，函数重载解析
for calls in class scope，类域中的函数调用的，15.10.3:647–648
for calls to function template instantiations，函数模板实例化调用的，10.8:431–436
for calls with arguments of class type，带有类类型实参调用的，15.10.2:645–647
inheritance and，继承与，19.3.1:859–862
for member function calls，成员函数调用的，15.11.2:653
for overloaded operators，重载操作符的，15.12.1:657–660

capacity for container type，容器类型的容量，6.3: 214–217
initial，relationship to size，与初始长度的关系，6.4:219

carriage return (\r) escape sequence，回车转义序列，3.1:62

case keyword，case 关键字，5.4:170
另参见 switch statement，switch语句

cast，强制转换，4.2:119，4.14.3:149–153
另参见 conversion，转换
const_cast operator，const_cast 操作符，4.14.3:151
dynamic_cast() operator，dynamic_cast() 操作符，**19.1.1:**836–840
forcing exact match in function overload resolution with，在函数重载解析中通过强制转换实现的精确匹配，9.3.1:386
implicit conversion compared with，与隐式转换的比较，4.14.3:151
old-style，旧式的，4.14.4:153
reinterpret_cast operator，reinterpret_cast 操作符，4.14.3:152
selection of function template instantiation，函数模板实例化的选择，10.3:414
static_cast operator，static_cast 操作符，4.14.3:151

catch clause，catch 子句，2.6:46，11.2:11:453–467
另参见 exception handling，异常处理
catch-all handler，catch-all 处理代码，11.3.4:461–462
exception declaration in，其中的异常声明，11.3:455
virtual function and，虚拟函数与，**19.2.4:**849–851
with exception as class hiararchy，类层次形式的异常，19.2.3:847–849

cerr，1.5:15
另参见 iostream
standard error represented by，表示标准错误，20:868

char type，char类型，2.1:18，3.1:61

character type，字符串类型
另参见 C-style character string，C风格字符串；string type，string类型
array of character，字符数组，**3.9**:94–95
character literal，字符文字
notation for，表示法，3.1:62
string literal compared with，与字符串文字的比较，3.1:63
wide-character literal，宽字符文字，3.1:62
null character，null字符，3.1:63
type (char)，char 类型，2.1:18，3.1:61
cin，1.5:15
另参见 iostream
standard input represented by，表示标准输入，**20:**868
class，类，
另参见 base class，基类；class member，类成员；derived class，派生类；inheritance，继承
access，访问，
to base class，对基类的，*参见* base class，基类
to member，对成员的，*参见* class member ，类成员
allocator class，dynamic memory management encapsulation (footnote)，分配器类，动态内存管理的封装（页下注），6.4:218
assignment，赋值，*参见* assignment，赋值
base class，基类，*参见* base class，基类
body，体，13.1: 503–504
(chapter)，**13:**503–564
constructor，构造函数，*参见*constructor，构造函数
declaration，class definition vs.，声明，和类定义，13.1.5:508–509
definition，定义，13.1:503–509
class declaration vs.，和类声明，13.1.5:508–509
derived，派生的，*参见* derived class，派生类
destructor，析构函数，*参见* destructor，析构函数
exception，异常，*参见* exception handling，异常处理
friend，友元，13.1.4:507–508，**15.2**:614
head，类头，13.1:503
hierarchy，层次，*参见* hierarchy，层次
initialization，初始化，*参见* constructor，构造函数
local class，局部类，13.12:562–564
member，成员，*参见* class member，类成员
as namespace member，用作名字空间成员，13.11:559–562
nested class，嵌套类13.10:551–559
object，对象，*参见* object，对象
parameter，参数
as holder of a group of parameter values，作为一组参数值的容器，7.4.1:300
efficiency consideration，效率考虑，7.3.1:285，**14.8**:600–604
return value，返回值，7.4:298–300
as holder of a group of return values，作为一组返回值的容器，7.4.1:301
template，模板，*参见* class template，类模板
union，联合，13.7: 539–543
class keyword，class 关键字，
class definition use，用于类的定义，13.1:503
class template definition use，用于类模板的定义，16.1:664–671
template type parameter use，用于模板类型参数
class template，类模板，16.2:671
function template，函数模板，10.1:407
typename as synonym for，typename 用作class的同义词，10.1:409
class member，类成员，
另参见 base class，基类；class，类；constructor，构造函数；destructor，析构函数；operator，操作符
access，访问，13.1.4:507–508，13.3.2:513–514
accessing private data with，访问私有数据，2.3:25，13.3.3:516
bit-fields，位域，**13.8**:544–545
data，数据，13.1:503–505
mutable，易变的，13.3.6:520–521
protected，17.2.1:728
public vs. private，私有与公有，13.1.3:506–507
static，静态，13.5:526–529
type of，类型，13.6.1:534–535
friend to，类成员的友元，13.1.4:507–508

function，函数，13.1.2:505–506，13.3:511–521
candidate member function，候选成员函数，15.11.2:653
const，13.3.5:517–520
constructor，构造函数，参见 constructor，构造函数
conversion，转换，15.9.1:637–640
destructor，析构函数，参见 destructor，析构函数
exception specification for，异常规范，19.2.6:852–853
inline vs. non-inline，inline和非inline，13.3.1:512–513
member operator function，成员操作符函数，*参见* operator，操作符
overloaded declaration of，重载声明，15.11.1:652–653
overload resolution and，重载解析和，**15.11**:652–656
private vs. public，私有与公有，13.3.3:514–517
special member function，特殊成员函数，13.3.4:517
static，静态，13.5.1: 529–530
type of，类型，13.6.1:534
viable member function，可行的成员函数，15.11.3:654–656
volatile，13.3.5:517–520
member access operator，成员访问操作符，2.3:25，13.2:509–511
static，静态，13.5:525–531
template，模板，16.7:691–695
this pointer，this 指针，13.4:521–525
use in overloaded assignment operator，用于重载赋值操作符中，14.7:598
when to use in member function，何时用于成员函数中，13.4.1:523–525
class template，类模板，
(chapter)，**16**:664–715
compilation model and，编译模式和，**16.8**:695–700
inclusion，包含，16.8.1:696–697
separation，分离，16.8.2:697–699
definition，定义，**16.1**:665–672
name resolution in，其中的名字解析，16.11:705–707
explicit instantiation declaration，显式实例声明，**16.8.3**:699–700
explicit specialization，显式特化，16.9:700–703
friend declaration in，其中的友元声明，**16.4**:682–687
instantiation，**实例化**，16.2: 672–679
member function of，成员函数，**16.3**:679–682
point of instantiation，实例化点，16.11:706–707
member template of，成员模板，**16.7**:691–695
namespace and，名字空间和，**16.12**:707–709
nested type of，嵌套类型，**16.6**:689–691
parameter，参数，16.1:667–669，16.2.1:675–678
non-type parameter，非类型参数，16.2.1:675–678
type parameter，类型参数，16.2:671–675
partial specialization，部分特化，**16.10**:703–705
static class member of，静态类成员，**16.5**:687–689
collection，集合，
参见 container type，容器类型
colon (:) operator，分号操作符
参见:
comma (，)，逗号，
参见，
command line option，命令行选项，7.8:306–315
argc，argv arguments to main()，main()的 argc，argv参数，7.8:306
example，范例，7.8:310–313
comment，注释，1.4:13–14
block，块，1.4:14
line (//)，行，1.4:14
pair (/*，*/)，对，1.4:13
nesting not permitted，不允许嵌套，1.4:14
compilation model，编译模式，
for class template，类模板的，16.8:695–700
inclusion，包含，16.8.1:696–697
separation，分离，16.8.2:697–699
for function template，函数模板的，10.5:421–424
inclusion，包含，10.5.1:421
separation，分离，10.5.2:421–423，16.8.2:697–699
complex header file，complex 头文件，

3.11:103
complex number，复数，1.2:11，**3.11**:103
abs() function，abs()函数，4.6:129
addition，加法，4.6:129
complex header file，complex 头文件，3.11:103
compound assignment，复合赋值，4.6:129
division，除法，4.6:129
initialization，初始化，4.6:128
operation，操作，4.6:128–130
representation，表示，4.6:129
composition，组合，
inheritance vs.，继承与，**18.3.1**:802–803
object，对象，18.3.4:804–806
compound assignment，复合赋值，4.4:126
complex number，复数，4.6:128
compound data type，复合数据类型，2.1:19
compound expression，复合表达式，4.1:117
compound statement，复合语句，5.1:159–160
linkage directive，链接指示符，**7.7**:304
condition state，条件状态，
参见 iostream
conditional (?:) operator，条件操作符，3.15:109，**4.7**:131
function compared with，与函数的比较 7.6:303
if-else shorthand use，if-else 的简写形式，5.3:168–169
conditional preprocessor directive，条件预处理器指示符，1.3:10–13
conditional statement，条件语句，5.1:159
if，5.3:163–170
switch，5.4:170–176
const，3.5:83–85
另参见 volatile
const iterator required for const containers，const 容器必需 const 迭代器，6.5:222–223，12.4:488
dynamic allocation and deallocation of const object，const 对象的动态分配和释放，**8.4.4**:346–347
function overload resolution，函数重载解析，
qualification conversion，限定修饰转换，9.3.1:385
ranking of reference initialization，引用初始化的分级，9.4.3:402
member function，成员函数，13.3.5:517–520
overloaded function declaration and const parameter，重载函数声明和 const 参数，9.1.2:371–372
parameter type，参数类型，
reference to const container type，const 容器类型的引用，7.3.4:292
reference to const type，const 类型的引用，7.3.1:285
array of const element，const 元素数组，7.3.3:289
pointer，指针，3.5:84
reference initialization with object of different type，不同类型对象的引用初始化，3.6:87
string literal as const character，作为 const 字符的字符串文字，3.1:63
transforming object into constant with，将对象转化为常量，3.5:84
volatile compared with，与 volatile 比较，3.13:105
const_cast operator，const_cast 操作符，4.14.3:151
另参见 cast，强制转换；conversion，转换
constant，常量，
expression，表达式，
array dimension required to be，数组维数必须是，3.9:93
sizeof() as，sizeof()被看作，4.8:134
folding，折叠，8.2.3:334
literal，文字，*参见* literal constant，文字常量
preprocessor，预处理器，*参见* preprocessor，预处理器
reference treated like，引用被当作，3.6:86
transforming object into，将对象转化为，3.5:83
constructor，构造函数，14.2:567–576
另参见 class，类；destructor，析构函数；inheritance，继承
for array element，数组元素的，
array initialization list，数组初始化列表，14.4:581–582
dynamic memory allocation，动态内存分配，14.4.1:583–584
for base class，基类的，17.4:743–751

in multiple inheritance，多继承中的，18.2:794–795
in single inheritance，单继承中的，17.4:743–748
in virtual inheritance，虚拟继承中的，18.5.2:816–820
memberwise initialization，按成员初始化，**17.4**:743–745
constraining object creation，限制对象创建，14.2.2:573–574
as conversion function，用作转换函数，15.9.2:640–642
copy constructor，拷贝构造函数，5.11:200，14.2.3:574
memberwise initialization，按成员初始化，14.6:592–597，**17.6**:772–774
use in dynamic vector growth，用于动态向量增长中，**6.3**:214–217
default constructor，缺省构造函数，14.2.1:572–573
used for vector element，用于向量元素，14.4.2:585–587
function try block and，函数 try 块与，19.2.7:854–855
member initialization list，成员初始化列表，14.5:587–592
return type not permitted，禁止指定返回值，14.2:692
virtual function call in，其中的虚拟函数调用，17.5.8:770–772

container type，容器类型，6:209–274
advantage，automatic memory management，优势，自动内存管理，8.4.3:345
associative container type，关联容器类型，6:209–274
capacity，容量，6.3:214
relationship to size，与长度的关系，6.3:214–217
copying container，拷贝容器，
memory allocation issue，内存分配问题，12.4.1:489
random access cost vs. the cost of，随机访问的代价与拷贝的代价的比较，6.2:214
definition，定义，6.4:217–221
deque，双端队列，*参见* deque container type，deque容器类型
initialization，初始化，
copying as initialization operation，通过拷贝进行初始化操作，6.4:219
with pair of iterator，迭代器，6.5:222–223
iterator and，迭代器与，6.5:221–224
list，*参见* list container type，list容器类型
map，*参见* map container type，map容器类型
multimap，*参见* multimap container type，multimap容器类型
multiset，*参见* multiset container type，multiset容器类型
parameter of，参数，7.3.4: 291–292，7.4.1:300
priority queue，优先队列，6.17:271–272
queue，队列，*参见* queue container type ，queue容器类型
requirement for type used to define a container，可定义为容器的类型的必要条件，6.4:220
return value of，返回值，7.4.1:301
sequence container type，序列容器类型，
assignment，赋值，6.6.2:227
criteria for choosing，选择准则，6.2:213
defining，定义，6.4:217–221
deleting element，删除元素，6.6.1:226
generic algorithm，泛型算法，6.6.3:227–228
inserting element in，插入元素，6.6:224–226
operation，操作，6.6:224–227
swapping element，交换元素，6.6.2:227
set，*参见* set container type，set容器类型
size，长度，6.4:218
relationship to capacity，与容量的关系，6.3:214–217
stack，*参见* stack container type，stack容器类型
STL-idiom use，STL习惯用法，**3.10**:101
string，*参见* string type，string类型
vector，*参见* vector container type，vector容器类型

continue statement，continue 语句，5.9:184–185

control flow，控制流，1.2.1:11–13
related concept，相关概念，*参见*
exception handling，异常处理，
loop，循环，

recursion，递归，
statement，语句，*参见*
do while statement，do while 语句
for statement，for 语句
goto statement，goto 语句
switch statement，switch 语句
while statement，while 语句
conversion，转换，
arithmetic，算术，**4.14.2**:148–149
bool to int，bool 到 int，3.7:90
array-to-pointer conversion，数组一指针转换，9.3.1:384
binary to unary function object，Binder function adaptor use for，二元到一元函数对象，使用绑定器适配器，12.3.5:486
conversion function，转换函数，**15.9.1**:637–640
constructor as，构造函数用作，15.9.2:640–642
conversion sequence，转换序列，
standard，标准的，9.4.3:399–403
user-defined，用户定义的，15.10:642–644
explicit type conversion，显式类型转换，*参见* cast，强制转换
exact match，精确匹配，9.3.1:382–386
function-to-pointer conversion，函数到指针转换，9.3.1:384–385
implicit type conversions，隐式类型转换，**4.14.1**:147
in function template argument deduction，函数模板实参推演中的，10.3:414
lvalue transformations，左值转换，9.3.1:386
in function template argument deduction，函数模板实参推演中的，10.3:414
lvalue-to-rvalue conversion，左值-右值转换，9.3.1:383–384
multiple conversion，cast as disambiguation of，多种转换，避免出现歧义，4.14.3:151
narrowing conversion，compiler warning，窄化转换，编译器警告，7.2.3:281
pointer conversion，指针转换，9.3.3:390
to and from void*，与 void*，4.14.3:149
to base class，为基类，17.1.1:724–727
to base class，in function overload resolution，为基类，在函数重载解析中，19.3.3:864–866
to base class，in function template argument deduction，为基类，在函数模板实参推演中，10.3:415
prohibited between function pointer，函数指针之间禁止，9.1.6:377–378
promotion，提升，4.14:146
of enumeration type to arithmetic type，枚举类型到算术类型，**3.8**:93
on argument，实参的，9.3.2:386–388
qualification conversion，限定修饰转换，9.3.1:385
in function template argument deduction，在函数模板实参推演中，10.3:415
ranking in function overload resolution，函数重载解析中的分级，
lvalue transformations，左值转换，9.4.3:400
reference initialization，引用初始化，9.3.4:391–392，9.4.3:402–403
standard conversion sequence，标准转换序列，9.4.3:399–403
user-defined conversion sequence，用户定义转换序列，15.10.4:648–651
user-defined conversion sequence，with inheritance，用户定义转换序列，带继承，19.3.2: 862–863
selection of conversion to and from class type，类类型转换的选择，15.10:642–644，15.10.4:648–651
standard conversion，标准转换，9.3.3:388–391
string object into C-style character string，string对象到C风格的字符串，3.4.2:80
user-defined conversion，用户定义转换，15.10:642–644
with inheritance，带继承，19.3.2:862–863
copy_backwards() generic algorithm，copy_backwards() 泛型算法，A:925
copy() generic algorithm，copy() 泛型算法，A:924
concatenating vector with，合并vector，12.2:471
inserter class use，用于 inserter 类，6.13.1:257
ostream_iterator and istream_iterator use，用于 ostream_iterator 和 istream_iterator，12.4.5:492
copy constructor，拷贝构造函数，

参见 constructor，构造函数
count()，
generic algorithm，泛型算法，A:926
map operation，map操作，6.12.2:251
multiset and multimap operation，multiset 和multimap操作，6.15:267–269
set operation，set操作，6.13.3:257
count_if() generic algorithm，count_if() 泛型算法，A:927
cout，1.5:15
另参见 iostream
standard output represented by，表示标准输出，20:868
ctype header file，ctype 头文件，6.10:239

D
dangling，空悬的，
else statement，else语句，5.3:164
pointer，指针，
to automatic object，自动对象的，8.3.2:336
to dynamically deallocated memory，指向动态分配内存的，2.2:22
data member，数据成员，13.1.1:504–505
另参见 access，访问；class member，类成员
access，访问，
to base class member，对基类成员的，17.3:736–742
to class member，对类成员的，13.1.3:506–507
base class vs. derived class member，基类与派生类成员，17.2:728–734
bit-field，位域，13.8:544–545
mutable，易变的，13.3.6:520–521
protected，17.2:728
public vs. private，公有与私有，13.1.3:506–507
static data member，静态数据成员，13.5:525–529
of class template，类模板的，**16.5**:687–689
this pointer，this 指针，13.4:521–525
use in overloaded assignment operator，用于重载赋值操作符中，14.7:598
when to use in member function，何时用于成员函数中，13.4:523–525
type of，类型，13.6.1:534–535
data type，数据类型，
参见 type，类型
__DATE__，1.3:16
deallocation，释放，
参见 dynamic memory deallocation，动态内存释放
declaration，声明，3.2.1:67
另参见 definition，定义
class declaration vs. class definition，类声明与类定义，13.1.5:508–509
declaration statement，声明语句，**5.2**:160–162
definition compared with，与定义的比较，**3.2.1**:67，8.2.1:395–397
exception declaration in catch clause，catch 子句中的异常声明，11.3:455–458
explicit instantiation declaration，显式实例声明，
for class template，类模板的，16.8.3:699–700
for function template，函数模板的，10.5.3:423–424
explicit specialization declaration，显式特化声明，
for class template，类模板的，16.9:700–703
for function template，函数模板的，10.6:424–428
in for loop init-statement，for 循环 init-statement 中的，5.5:177–178
friend declaration，友元声明，
in class，类中的，13.1.4:507–508，15.2:614–616
in class template，类模板中的，**16.4**:682–687
function declaration，函数声明，7.2:280–281
located in header file，头文件中的，8.2.3:333–335
interfile declaration matching，不同文件之间声明的匹配，8.2.2:331
locality of，局部性，5.2:161
object declaration，对象声明，8.2.1:330
located in header file，头文件中的，8.2.3:333–335
in statement condition，在语句条件中，5.3:163，8.1.1:328
using declaration，using 声明，*参见* using declaration，using 声明
decrement (--) operator，递减操作符，

built-in，内置，4.5:126–127
overloaded operator，重载操作符，15.7:623–626
default keyword，default 关键字，
switch statement use，switch 用于语句，5.4:170–171，173–174
default argument，缺省实参，7.3.5:293–295
viable function and，可行函数，9.4.4:403–404
virtual function and，虚拟函数和，**17.5.4**:760–762
default constructor，缺省构造函数， *参见* constructor，构造函数
#define directive，#define 指示符， 1.3:10
definition，定义，1.2:4
另参见 declaration，声明
class definition，类定义，**13.1**:503–509
class declaration vs. class definition，类声明与类定义，13.1.5:508–509
declaration compared with，与声明的比较，3.2.1:67，8.2.1:331–332
function definition，函数定义，7.1:277
local scope and，局部域和，8.1.1:327
and header file，和头文件，8.2.3:333–335
namespace definition，名字空间定义，**8.5**:349–361
object definition，对象定义，**3.2.3**:69–71，8.2.1:331
delete expression，delete 表达式，4.9:134–135
另参见 dynamic memory deallocation，动态内存释放
allocator class encapsulation，(footnote)，分配器类的封装，（页下注），6.4:218
for array，数组的，8.4.3:346
of class，类的，14.4.1:584，15.8.1:630–631
for single object，单一对象的，8.4.1:340–341
of class type，类类型的，14.3:578，15.8:626–629
const，8.4.4:347
deletion generic algorithm，删除泛型算法，12.5.3:496
另参见 generic algorithm，泛型算法
deque container type，deque容器类型，
另参见 vector；generic algorithm，泛型算法；container type，容器类型
<deque> header file，<deque>头文件，6.4:217
as specialized vector，用作特化vector，6:209，6.2:213
efficient first element insertion and deletion，首元素的有效插入和删除，6:209，6.2:213–214
push_front()，6.4:218
underlying stack implementation，底层 stack 实现，6.16:270
dereference (*) operator，解引用操作符，2.2:22
另参见 address-of (&) operator，取地址操作符；pointer，指针
defining pointer with，用于定义指针，3.3:72
defining function pointer with，用于定义函数指针，7.9.1:316
use in expression，用于表达式中，3.3:73–75，4.1:117
accessing array element，访问数组元素，3.9.2:97
not required for function invocation，函数调用不必要，7.9.3:317
derivation，派生，
参见 derived class，派生类
derived class，派生类，
另参见 base class，基类；class，类；class member，类成员；inheritance，继承
assignment，memberwise，赋值，按成员，17.6:774–776
constructor，构造函数，17.4.2:745–746
virtual function calls in，其中的虚拟函数，17.5.8:770–772
definition，定义，
in multiple inheritance，多继承中的，18.2:794
in single inheritance，单继承中的，17.2.2:732–734
in virtual inheritance，虚拟继承中的，18.5.1:815–816
destructor，析构函数，17.4.5:749–751
virtual，虚拟的，17.5.5:763–764
virtual function calls in，其中的虚拟函数调用，17.5.8:770–772
initialization，初始化，17.4:743–751
memberwise，按成员，17.4:743–751
in multiple inheritance，多继承中的，18.2:794

in virtual inheritance，单继承中的，18.5.2:816–820
virtual function，虚拟函数，17.5:752–772
destructor，析构函数，14.3:576–581
for array element，数组元素的，14.4:581–582
dynamic memory deallocation，动态内存释放，14.4.1:584–585
for base class，基类的，
in single inheritance，单继承中的，17.4.5:749–751
in multiple inheritance，多继承中的，18.2:795
in virtual inheritance，虚拟继承中的，18.5.3:819–820
exception handling stack unwinding and，异常处理栈展开与，19.2.5:851–852
explicit invocation of，显式调用，**14.3.1**:579
inline destructor and potential code bloat，inline析构函数和潜在的代码膨胀，14.3.2:579–580
virtual，虚拟的，
destructor，析构函数，17.5.5:763–764
function call in，其中的函数调用，17.5.8:770–772
directive，指示符，
参见 linkage directive，链接指示符；preprocessor directive，预处理器指示符；using directive，using 指示符
divides function object，divides 函数对象，12.3.2:484
division (/) operator，除法操作符，2.1:18，4.2:118
complex number，复数支持，4.6:129
compound assignment (/=) operator，复合赋值操作符，4.4:126
do-while statement，do-while 语句，5.7:182
for and while statements compared with，与for和while语句的比较，5.5:176
dot (.) operator，句点操作符，13.3.2:513–514
另参见 class member，类成员
double data type，double 数据类型，
另参见 floating point type，浮点类型
double data type，double数据类型，2.1:18，**3.1:61**
long double data type，long double数据类型，3.1:61
dynamic_cast() operator，dynamic_cast() 操作符，19.1.1:836–840
dynamic memory allocation，动态内存分配，2.2:26–29
另参见 new expression，new 表达式
array，数组，4.9:134，**8.4.3**:345–346
of class，类的，14.4.1:583–584，15.8.1:629–631
auto_ptr management of，auto_ptr 管理，8.4.2:341–344
dangling pointer，空悬指针，8.4.1:340
growth requirement for container type，容器类型增长的必要条件，6.3:214
memory exhaustion，bad_alloc exception，存储区耗尽，bad_alloc 异常，8.4.1:340
object，对象，8.4:338–344
const object，const 对象，8.4.4:346–347
of class type，类类型的，15.8:626–629
placed in existing memory，在已有内存中定位，8.4.5:347–348，15.8.1:629–631
static and dynamic，differences between，静态和动态之间的区别，2.2:22
dynamic memory deallocation，动态内存释放，2.2:22–23
另参见 delete expression，delete 表达式
array，数组，4.9:134，**8.4.3**:345–346
of class，类的，14.4.1:584–585，15.8.2:631–632
auto_ptr management of，auto_ptr 管理，8.4.2:341–344
common programming error，常见编程错误，8.4.1:341
omitting array brackets in delete expression，delete表达式中遗漏了数组中括号，8.4.3:346
dangling pointer，空悬指针，8.4.1:340
memory leak，内存泄漏，8.4.1:341
object，对象，8.4.1:339–341
const object，const 对象，8.4.4:347
of class type，类类型的，15.8.1:630–631
placement delete，定位delete操作符，15.8.2:632

E

E suffix，E后缀，3.1:62
另参见 floating point type，浮点类型
edit-compile-debug cycle，编辑-编译-调试循环，1.2:7
efficiency，效率，
参见 performance，性能
ellipses (...)，省略号，7.3.6:295
参见...
else statement，else语句，5.3:164
另参见 if statement，if语句
dangling else problem，空悬else问题，5.3:165
encapsulation，封装，
参见 access，访问；class member，类成员；information hiding，信息隐藏
end() function，end()函数，6.5:221
另参见 iterator，迭代器
#endif directive，#endif 指示符， 1.3:10
另参见 preprocessor directive，预处理器指示符
endl iostream manipulator，endl iostream 操纵符， 1.5:15
另参见 iostream
(table)，（表20–1），20.9:916
enum keyword，enum 关键字，3.8:92
enumeration type，枚举类型，3.8:91–93
enumerator，枚举成员，3.8:91
function overload resolution，函数重载解析，
exact match with，精确匹配，9.3.1:382
promotions with，提升，9.3.2:386
promotions to arithmetic type，3.8:93
equal() generic algorithm，equal() 泛型算法，A:929
equal_range() generic algorithm，equal_range() 泛型算法，A:930
multiset and multimap use，用于multiset和multimap，6.15:267
equal_to function object，equal_to 函数对象，12.3.3:484
equality (==)，等于，
另参见 operator，操作符
built-in operator，内置操作符，4.3:120–122
overloaded operator，重载操作符，15-15.1.1:605–611
requirement for container element type，容器元素类型必须支持，6.4:220
error，错误，
另参见 exception handling，异常处理
abort() function，abort()函数，5.11:190
terminate() function default behavior，terminate() 函数的缺省行为，11.3.2:459
array，数组，
range error potential，可能的范围错误，2.1:19，3.4.1:77，3.9:95
omitting trailing null for character string，忘记字符串的结尾空字符，8.4.3:346
assert() macro use，assert()宏的使用，5.11: 190
assignment operator confusion with equality operator，等于操作符和赋值操作符的混淆，3.5:83
bitwise operator problems，按位操作符问题，4.11:137
dangling else problem，空悬else问题，5.3:165
dangling pointer，空悬指针，
to automatic object，自动对象的，8.3.1:336
to dynamically allocated memory，指向动态分配的内存的，8.4.1:340
dynamic memory allocation error，动态内存分配错误，8.4.1:341
omitting array brackets in delete expression，delete表达式中遗漏了数组中括号，8.4.3:346
global object pitfalls，全局对象中易犯的错误，7.4.1:300
infinite function recursion，无限函数递归，7.5:301
infinite loop，无限循环，3.4.1:77
lazy error detection，迟缓型错误检测，17.4.4:749
link phase，链接阶段，
missing function template definition，函数模板定义遗漏，10.8:433
multi-file declaration matching，多个声明的匹配，8.2.2:331–332
multiple definition，多个定义，8.2.3:333，8.5.5:358
standard error，represented by cerr，标准错误，用 cerr 表示，20:868
uninitialized object，未初始化对象，3.2.3:69，8.3.1:335，8.3.3:337–338

using declaration scope conflict，using 声明域冲突，9.1.4:375
using directive pitfalls，using 指示符易犯的错误，8.6.3:365

escape sequence，转义序列，
notation for，表示法，3.1:62

exact match，精确匹配，9.3.1:382–386
另参见 conversion，转换；function overload resolution，函数重载解析

example，范例，
Account class，Account类，13.5:526–530，13.6.3:538，14.2–14.7:567–599
Array class template，Array类模板，2.5:40–46，16.13:709–715
Array_RC_S derivation，Array_RC_S的派生，18.6.3:832–833
Array_RC derivation，Array_RC的派生，2.5:40–46，**18.6.1:825:827**
Array_Sort derivation，Array_Sort的派生，18.6.2:827–832
command line options handling，命令行选项处理，7.8:306–314
generic algorithm use，泛型算法的使用，12.2:471–480
IntArray class，IntArray类，2.3:23–32
IntArrayRC derivation，IntArrayRC的派生，2.4:32–40
IntSortedArray derivation，IntSortedArray的派生，2.4:32–40
iStack class，iStack类，4.15:154–158
changed to a Stack template，改变为Stack模板，6.18:272–274
support for dynamic memory allocation，对动态内存分配的支持，6.18:272–274
support for exception handling，异常处理的支持，11:449–467
linked list class，链表类，5.11:186–203
changed to a list template，改变为list模板，5.11.1:203–206
nested class implementation，嵌套类的实现，13.10:551–557
Queue class template，Queue类模板，16.1–16.12:664–709
Screen class，Screen类，13:503–525
sort() function，sort()函数，7.9:315–322
changed to a template function，改为模板函数，10.11:446–448
String class，String类，3.15:106–115
text query system，文本查询系统，
(Chapter 12)，**12:**468–500
(Chapter 17)，**17:**719–789
(Chapter 6)，**6:**209–274
ZooAnimal class hierarchy，ZooAnimal类层次，18.2–18.5:794–822

exception declaration，异常声明，11.3:455
另参见 exception handling，异常处理

exception handling，异常处理，
bad_alloc exception for memory exhaustion，存储区耗尽引起的 bad_alloc 异常，8.4.1:340
catch clause，catch 子句，11.3:455–462
catch–all handler，catch–all 处理代码，11.3.4:461–462
exception declaration in，其中的异常声明，11.3:455
virtual function and，虚拟函数与，19.2.4:849–850
with exception as class hiararchy，类层次形式的异常，19.2.3:847–849
design issue，设计问题，11.5:466–467
exception object，异常对象，11.3.1:456–459
exception specification，异常规范，11.4:463–466
empty as guarantee of no exception，空的异常规范保证不抛出异常，11.4:464
pointer to function and，函数指针与，11.4.1:465
with exception as class hierarchy，类层次形式的异常，19.2.6:852–854
exceptions as class hierarchy，类层次形式的异常，19.2.1:845–846
in C++ Standard Library，C++标准库中的，19.2.8:855–858
function try block，函数 try 块，11.2:454，19.2.7:854–855
handler，处理代码，*参见* catch clause，catch 子句
resource release with catch-all，用catch-all 释放资源，11.3.4:461
stack unwinding，栈展开，11.3.2:459
with destructor call，调用析构函数，19.2.5:851–852

terminate() function，terminate()函数，11.3.2:459
throw expression，throw 表达式，11.1:449–452
handling when not in a try block，不在try块中的处理，11.3.2:459
rethrow，重新抛出，11.3.3:459-461
with exception as class hiararchy，类层次形式的异常，19.2.2:846–847
try block，try 块，11.2:452–455
unexpected() function，unexpected()函数，11.4:464
exception specification，异常规范，11.4: 463–466
另参见 exception handling，异常处理
explicit instantiation declaration，显式实例声明，
class template，类模板，16.8.3:699
function template，函数模板，10.5.3:423–424
explicit keyword，explicit 关键字，2.3:28，15.9.2:641
explicit specialization，显式特化，
class template，类模板，16.9:700–703
function template，函数模板，10.6:424–428
overload resolution and，重载解析与，10.8:432–434
explicit type conversion，显式类型转换，4.2:119，**4.14:**146，**4.14.3:**149–153
另参见 cast，强制转换
export keyword，export 关键字
class template definition use，用于类模板定义，16.8.2:697
function template definition use，用于函数模板定义，10.5.2:422
member function of class template use，用于类模板的成员函数，16.8.2:698
expression，表达式，
另参见 delete expression，delete 表达式；new expression，new 表达式；throw expression，throw 表达式
(chapter)，**4:**116–158
compound expression，复合表达式，4.1:117
name resolution in，其中的名字解析，8.1:326
subexpression evaluation order，子表达式的计算顺序，4.1:117
extent，范围，
另参见 lifetime，生命期
automatic，自动，8.3.1:335
static，静态，8.3.3:337
extern，
constant，常量，8.2.3:334
function template，函数模板，10.1:410
function pointer，函数指针，7.9.6:322
as linkage directive，用作链接指示符，7.7:304
namespace member，名字空间成员，8.5.5:358
object，对象，8.2.1:331
located in header file，头文件中的，8.2.3:333
extern "C"，7.7:304–306
overloaded function and，重载函数与，9.1.5:376–377
pointer to function，函数指针，7.9.6:322–323
type-safe linkage not applicable to，类型安全链接不适用，9.1.7: 378

F
F suffix，F后缀，3.1:62
另参见 floating point type，浮点类型
false keyword，false 关键字，3.7:90
另参见 bool type，bool 类型
__FILE__，1.3:12
file，文件
另参见 header file，头文件；iostream
current file，当前文件，1.3:12
declaring entities local to，unnamed namespace use，声明局部于文件的实体，用于未命名的名字空间，8.5.6:359
file I/O，文件 I/O，*参见* iostream
header，头，*参见* header file，头文件
multiple，多个文件
declaration matching in，在多个文件中声明，8.2.2:331
explicit template specializations in，在多个文件中显式模板特化，10.6:427
namespace definition spanning，名字空间定义可以跨越，8.5.1:352
template point of instantiation in，在多个文件中的模板实例化点，10.9:441
fill() generic algorithm，fill() 泛型算法，A:932
fill_n() generic algorithm，fill_n() 泛型算法，A:933

find()，
generic algorithm，泛型算法，A:934
iterator requirement，迭代器的要求，12.1:468–471，12.5:494
map operation，map操作，6.12.2:251
multiset and multimap operation，multiset和multimap操作，6.15:267
set operation，set操作，6.13.3:257
string operation，字符串操作，6.8:231
find_end() generic algorithm，find_end() 泛型算法，A:936
find_first_not_of() string operation，find_first_not_of() 字符串操作，6.8:236
find_first_of()，
generic algorithm，泛型算法，A:937
string operation，字符串操作，6.8:231，6.9:237
find_if() generic algorithm，find_if() 泛型算法，A:935
find_last_not_of() string operation，find_last_not_of() 字符串操作，6.8:236
find_last_of() string operation，find_last_of()字符串操作，6.8:236
float data type，float 数据类型， 2.1:18，3.1:61
另参见 floating point type，浮点类型
floating point type，浮点类型，
arithmetic issue，算术问题，4.2:119
data type that represent，表示浮点数的数据类型，3.1:61
double data type，double 数据类型，2.1:18，3.1:61
float data type，float 数据类型，2.1:18，3.1:61
literal constant notation，文字常量记号，3.1:61
E suffix，exponent notation，E后缀，幂记号，3.1:62
F suffix，single precision notation，F后缀，单精度记号，3.1:62
L suffix，extended precision notation，L后缀，扩展精度记号，3.1:62
long double data type，long double 数据类型，3.1:61
standard type conversion，标准类型转换，4.14.2:148
during function overload resolution，在函数重载解析时，9.3.2:388
flow of control，控制流
参见 control flow，控制流
for_each() generic algorithm，for_each() 泛型算法，A:939
vector container type and，vector 容器类型与，12.2:476–477
for statement，for 语句， 5.5:176–179
另参见 control flow，控制流；loop，循环
format state，格式状态，20.9:911–917
另参见 iostream
formfeed (\f) escape sequence，进纸转义序列，3.1:62
ForwardIterator，12.4.6:493
free store，空闲存储区，4.9:134，**8.4:**338
allocating on，空闲存储区上的分配，*参见* dynamic memory allocation，动态内存分配；new expression，new 表达式
exhaustion，bad_alloc exception，空闲存储区的耗尽，bad_alloc 异常，8.4.1:340
freeing，空闲存储区的释放，*参见* dynamic memory deallocation，动态内存释放；delete expression，delete 表达式
friend，友元，13.1.4:507–508，**15.2:**614–616
另参见 access，访问；class，类；class template，类模板
overloaded operator as，声明为友元的重载操作符，15.2:614–615
of class template，类模板的友元，16.4:682–687
front() function，front()函数，6.17:271
另参见 queue and priority_queue container type，queue 和 priority_queue 容器类型
front_inserter() function adaptor，front_inserter()函数适配器，12.4.2:489
fstream，
class，file I/O with，类，文件I/O，20:869
header file，头文件，1.5.1:16–17，20:869
function，函数，7.1:277–279

另参见 operator，操作符
activation record，活动记录，7.3:282
automatic object allocation in，其中的自动对象分配，8.3.1:335
benefit of，函数的优势，303
block，块，7.1:277
body，体，7.1:277
enclosed within a try block，try块中的，11.2:454
call，调用，7.1:278
drawback of，函数调用的缺点，7.6:303
exception handling compared with，与异常处理的比较，11.3.2:459
candidate function，候选函数，*参见* function overload resolution，函数重载解析
(chapter)，7:277–324
conversion function，转换函数，**15.9.1**:637–640
constructor as，作为转换函数的构造函数，15.9.2:640–642
declaration，声明，7.2:280–281
as namespace member，声明为名字空间成员，8.5:349–351
as part of function template，声明为函数模板的一部分，10.1:407
definition compared with，与定义的比较，8.2.1:330
definition，定义，7.1:277
as part of function template，定义为函数模板的一部分，10.1:407
declaration compared with，与声明的比较，8.2.1:330
function call (()) operator，函数调用操作符，7.1:278
overloaded for class type，为类类型重载的，15.5:619–620
function name，函数名
evaluate as pointer to its type，解释成该类型的指针，7.9.2:317
overloading，重载，9.1.1–9.1.5:369–377
function-to-pointer conversion，函数到指针转换，9.3.1:384–385
functional header file，functional 头文件，*参见* function object，函数对象
function try block，函数 try 块，11.2:454
global object and，全局变量和，8.2:330–335
inline，内联，*参见* inline function，内联函数
interface，接口
exception specification in，接口中的异常规范，11.4:463–466
function declaration as，声明为接口的函数，7.1:279
function prototype as，用作接口的函数原型，7.1:279
invocation，调用，7.1:278–279
local scope and，局部域和，8.1.1:327
local storage area，局部存储区，7.3:282
member function，成员函数，*参见* member function，成员函数
non-native，linkage directives for，7.7:304–305
object，对象，*参见* function object，函数对象
overloaded function declaration，重载函数声明，9.1:369
另参见 function overload resolution，函数重载解析
how to overload function，如何重载函数，9.1.2:370–371
scope and，域和，9.1.4:373–376
when not to overload function，何时不重载函数，9.1.3:372–373
why overload function，为什么要重载函数，9.1.1:369–370
parameter list，参数表，**7.2.2**:280–281，7.3:282–295
参见 function parameter，函数参数
pointer，指针，*参见* function pointer，函数指针
prototype，原型，**7.2**:279–281，7.3–7.4:282–300
recursive，**7.5**:301–302
return type，返回类型，**7.2.1**:279–280
array type prohibited，禁用数组类型，7.2.1:280
constructor not permitted to have，构造函数不允许有返回类型，14.2:567
function pointer as，函数指针，7.9.5:319–322
function type prohibited，禁用函数类型，7.2.1:280

overloaded function, insufficient to disambiguate, 重载函数，不足以消除二义性，9.1.2:371
pair type, pair类型，5.3:166
reference, 引用，7.4:299
return value, 返回值，7.4:297–301
class object, 类对象，7.4:299–301
global object compared with, 与全局对象的比较，7.4.1:300–301
local object, problem for reference return type, 局部对象，引用返回值的问题，7.4:299
reference parameter use as additional return value, 引用参数用作额外的返回值，5.3:, 7.3.1:284
strategy for multiple, 多个返回值的策略，7.4:300
signature, 符号特征，7.2.2:281
template, 模板，*参见* function template，函数模板
type, 类型，
conversion to function pointer, 转换为函数指针，7.9.2:317
prohibited from being function return type, 禁止成为函数返回类型，7.2.1:280
viable, 可行的，*参见* function overload resolution，函数重载解析
virtual, 虚拟的，*参见* virtual function，虚拟函数

function adaptor，函数适配器，12.3.5:486
另参见 function object，函数对象
binder，绑定器，12.3.5:486
bind1st，12.3.5:486
bind2nd，12.3.5:486
negator，取反器，12.3.5:486

function object，函数对象，12.2:474–475, **12.3:481–487**
另参见 function adaptor，函数适配器
advantage over function pointer, 相对于函数指针的优势，12.3:481–482
arithmetic function object, 算术函数对象，
divides<Type>, 12.3.2:484
minus<Type>, 12.3.2: 484
modulus<Type>, 12.3.2: 484
multiplies<Type>, 12.3.2: 484
negate<Type>, 12.3.2: 484
plus<Type>, 12.3.2: 484
definition, 定义，**12.3.6:** 486
function adaptor for, 函数适配器，12.3.5: 486
<functional> header file, <functional>头文件，12.3.1: 482
logical function object, 逻辑函数对象，
logical_and<Type>, 12.3.4: 485
logical_not<Type>, 12.3.4: 485
logical_or<Type>, 12.3.4: 485
motivation for, 动机，12.3: 481
relational function object, 关系函数对象，
equal_to<Type>, 12.3.3: 484
greater<Type>, 12.3.3: 485
greater_equal<Type>, 12. 3.3: 485
less<Type>, 12.3.3: 485
less_equal<Type>, 12.3.3: 485
not_equal_to<Type>, 12.3.3: 484
use in generic algorithm, 用于泛型算法，12.1:468–469, 12.3:481–482

function overload resolution，函数重载解析，9.2:379–381
best viable function, 最佳可行函数，9.2:380, **9.4.3:**399–403
for call with argument of class type, 针对类类型实参的调用，15.10.4:648–651
inheritance and, 继承和，19.3.3:864–866
candidate function, 候选函数，9.2:380, **9.4.1:**394–397
for call in class scope, 类域中调用的，15.10.3:647–648
for call to member function, 成员函数调用的，15.11.2:653
for call with arguments of class type, 类类型实参调用的，15.10.2:645–647
for operator function, 操作符函数的，15.12.1:657–660
inheritance and, 继承和，19.3.1:859–862
(chapter), **9:**443–487
conversion on argument, 实参转换，*参见* conversion，转换
detailed description of process, 解析过程的详细说明，**9.4:**393–404

explicit cast as guidance for，显式强制转换作为指导，9.3.1:386
member function and，成员函数和，15.11:652–656
ranking，分级，
standard conversion sequence，标准转换序列，9.4.3:399–403
user-defined conversion sequence，用户定义转换序列，19.3.2:862–863
template with，模板的，
with template instantiation，模板实例化，10.8:430–436
with template explicit specialization，模板显式特化，10.8:432–433
viable function，可行函数，9.4.2:397–399
default argument and，缺省实参和，9.4.4:403–404
for call to member function，成员函数调用的，15.11.3:654–656
for operator function，操作符函数的，15.12.2:660–661
inheritance and，继承和，19.3.2:862–863
function parameter，函数参数，7.1:277，**7.2.2**:280–281
abstract container type as，抽象容器类型用作，7.3.4:291–292
array as，数组用作，**7.3.3**:289–291
default argument for，缺省实参，**7.3.5**:293–295
ellipses use for，省略号用作，**7.3.6**:295
exception declaration compared with，与异常声明的比较，11.3.2: 459
function pointer as，函数指针用作，7.9.5:319–322
global object vs.，全局对象与，7.4.1:300–301
overloaded function differentiated by，重载函数的区别，9.1.2:370–372
pointer as，指针用作，7.3:283
array parameter relationship to，与数组参数的关系，7.3.3:289–291
reference parameter relationship to，与引用参数的关系，7.3.2:286–289
reference as，引用用作，3.6:89，**7.3.1**:284–286
multiple return value use，用于多个返回值，5.3:166，7.3.1:284
passing array as，传递数组，7.3.3:290
performance advantage，性能优势，7.3.1:285
pointer parameter relationship to，指针参数的关系，7.3.2: 286– 289
reference to constant，常量的引用，7.3.1:285
type checking，类型检者，**7.2.3**:281–282
function pointer，函数指针，7.9:315–324
array of，数组，**7.9.4**:318–319
assignment，赋值，7.9.2:317
data pointer vs. (footnote)，数据指针和（页下注），3.3: 72
disadvantage vs. inlining，与内联相比的劣势，12.2:474，12.3:481
exception specification and，异常规范和，11.4.1:465–466
to extern "C" function，extern "C" 函数的，7.9.6: 322–323
function object benefit compared with，与函数对象的比较优势，12.2:474，12.3:482
initialization of，初始化，7.9.2:317
invocation，**调用**，7.9.3:317–318
to non-native function，非本机函数的，7.9.6: 322–323
to overloaded function，重载函数的，9.1.6:377–378
parameter，参数，7.9.5:319–322
return type，返回类型，7.9.5: 319–322
type of，类型，**7.9.1**:316–317
function template，函数模板，
(chapter)，**10**:489–545
compilation model，编译模式，10.5:420–424
inclusion，包含，10.5.1:421
separation，分离，10.5.2:421–423
definition，定义，10.1:405–411
explicit，显式的，
instantiation declaration，实例化声明，10.5.3: 523–524
specialization，特化，10.6:424–428
template argument，模板实参，10.4:417–420
instantiation，实例化，10.2:411–414

address of，取地址，10.2:413
name resolution in definition，定义中的名字解析，10.9:437–442
namespace and，名字空间和，10.10:442–445
overload resolution，重载解析，
with instantiation，实例化，10.8:430–436
with explicit specialization，显式特化，10.8:432
overloaded declaration，重载声明，10.7:428–430
template parameter，模板参数，10.1:406–411
limitation of generic algorithm，泛型算法的局限，12.3:481
non-type parameter，非类型参数，10.1:407
passing function object to，传递函数对象给，12.3.1:483
type parameter，类型参数，10.1: 407
point of instantiation，实例化点，10.9:440
template argument deduction，模板实参推演，10.3:414–417
return type and，返回类型和，10.4:418
functional header file，functional 头文件， 12.3.1:482

G

gcount() function，gcount()函数， 20.3:887
另参见 iostream
generic algorithm，泛型算法，
另参见 iterator，迭代器；function object，函数对象；container，容器
Appendix，alphabetical reference ，附录，字母顺序参考，**A:**919–983
（chapter），**12:**468
accumulate()，A:920
adjacent_difference()，A:921
adjacent_find()，A:922
<algorithm> header file，<algorithm>头文件，2.8:57，12.2:472
binary_search()，A:923
category and description，分类和说明，12.5：494–495
container and generic algorithm，容器与泛型算法，6.6.3:227–228
contrast with list member function，**list成员函数的比较，**12.6:497
copy version of algorithm，算法的拷贝版本，12.5:495
copy()，6.13.1:306，12.2:472，12.4.3:490，12.4.6:493，A:924
copy_backwards()，A:925
count()，A:926
count_if() ， 12.2:475 ， 12.3.5:486 ， 12.3.6:486，A:927
deletion generic algorithm，删除泛型算法，12.5.3:496
element range notation，元素范围的表示，12.5: 494
equal()，A:929
equal_range()，A:930
fill()，A:932
fill_n()，A:933
find()，12.1:468–471，12.5:494，A:934
find_end()，A:936
find_first_of()，A:937
find_if()，A:935
for_each()，12.2:476，A:939
function object as argument to，函数对象作为实参，12.3:482
generate()，A:939
generate_n()，A:940
heap generic algorithm，堆泛型算法，A:981–983
make_heap()，A:981
pop_heap()，A: 981
push_heap()，A: 982
sort_heap()，A: 982
includes()，A: 941
inner_product()，A:942
inplace_merge()，A:943
iterator as range-marker，iterator用作范围标志，12.1:469
iterator as parameter，iterator参数，12.5:494
iter_swap()，A:944
left-inclusion interval([])，左闭合区间，12.5:494
lexicographical_compare()，A:945
lower_bound()，A:947
max()，A: 948
max_element()，A: 948
min()，A:948
min_element()，A:949

merge()，A:950
mismatch()，A:951
mutation generic algorithm，异变泛型算法，12.5.6:496
next_permutation()，A:952
numeric generic algorithm，算术泛型算法，12.5.5:496
<numeric> header file，<numeric>头文件，12.5:495
nth_element()，A:953
ordering generic algorithm，整序泛型算法，12.5.2:495–496
overview，概述，**12.1:468**
partial_sort()，A:954
partial_sort_copy()，A: 955
partial_sum()，A: 956
partition()，A: 957
permutation generic algorithm，排列组合泛型算法，12.5.4: 496
prev_permutation()，A:958
program example，**程序实例，12.2**:471–480
random_shuffle()，A:959
relational generic algorithm，关系泛型算法，12.5.7:496
remove()，12.12:476，A:960
remove_copy()，A: 960
remove_copy_if()，A: 961
remove_if()，A: 961
replace()，A: 962
replace_copy()，A: 963
replace_copy_if()，A: 964
replace_if()，A: 964
reverse()，A: 965
reverse_copy()，A: 965
rotate()，A: 966
rotate_copy()，A: 966
search generic algorithm，查找泛型算法，12.5.1:495
search()，A:967
search_n()，A: 968
set_difference()，A: 969
set_symmetric_difference()，A: 970
set_union()，A:971
sorting generic algorithm，排序泛型算法，12.5.2:495–496
sort()，2.8:54，12.2:472，12.4.2:490，12.4.3:491，A:972
stable_partition()，A:973
stable_sort()，12.2:473，A:974
substitution generic algorithm，替换泛型算法，12.5.3:496
swap()，A:975
swap_range()，A: 975
transform()，A: 977
unique()，12.2:472，12.5:495，A:978
unique_copy()，12.4.1:489，12.4.3:490，A:978
upper_bound()，A:980
get() function，get()函数，20.3:886–890
另参见 iostream
getline() function，getline()函数，6.7:228，20.3:886–890
另参见 iostream
global，全局，
另参见 name，名字；namespace，名字空间；scope，域；visibility，可见性
function，函数，8.2:330–335
object，对象，8.2:330–335
parameter and return value vs.，参数和返回值与，7.4.1:300–301
name space pollution problem，名字空间污染问题，2.7:50，8.5:420
namespace scope，名字空间域，8.1:325，8.5:349
access hidden member with scope operator，用域操作符访问隐藏成员，8.5.2:353
goto statement，goto 语句，5.10:185
greater function object，greater 函数对象，12.3.3:485
greater_equal function object，greater_equal 函数对象，12.3.3:485
greater than (>) operator，大于操作符，
arithmetic data type support of，算术数据类型支持，2.1:18，4.3:120

H

header file，头文件，
constant definition in，其中的常量定义，8.2.3:334
declaration in，其中的声明，3.2.1:67，8.2.3:333–335

function declaration in，其中的函数声明，
 7.1:278–279
 with default argument，用缺省参数，
 7.3.5:294
 with exception specification，用异常规范，
 11.4:463
 with linkage directive，用链接指示符，
 7.7:304–305
function template，函数模板，
 definition in，其中的定义，10.5.1:421
 explicit specialization declaration in，其中的显式特化声明，10.6:426
 explicit instantiation declaration in，其中的显式实例化声明，10.5.3:423
inline function definition，内联函数定义，
 7.6:303，8.2.3:334
named，命名的，
 algorithm，2.8:56，12.5:495
 bitset，4.12:139
 complex，3.11:103
 ctype，6.10:239
 deque，6.4:217
 fstream，20:869
 functional，12.3.1:482
 iomanip，3.15:112
 iostream，20:868
 iterator，12.4.3:490
 limits，4.2:119
 list，6.4:217
 locale，6.10:240
 map，6.12:247
 map， multimap use， map， 用 于 multimap，6.15:267
 memory，8.4.2:341
 numeric，12.5:495
 queue，6.17:271
 set，6.13.1:256
 set，multiset use，set，用于 multiset，
 6.15:267
 sstream，20:871
 stack，6.16:269
 string，3.4.2:79
 typeinfo，19.1.3:842
 utility，3.14:105
 vector，2.8:54，3.10:99，6.4:217

 object declaration in，其中的对象声明，
 8.2.3:334
 pre-compiled，预编译的，8.2.3:335
heap，堆，4.9:134，12.5.9:497
 allocating on， 在 堆 中 分 配， *参 见* dynamic memory allocation，动态内存分配；new expression，new 表达式
 exhaustion，bad_alloc exception，耗尽，bad_alloc 异常，8.4.1:340
 freeing，释放，*参见* dynamic memory deallocation，动态内存释放；delete expression，delete 表达式
 generic algorithm，泛型算法，12.5.9:497，
 A:981
 另参见 generic algorithm，泛型算法
hexadecimal notation，十六进制表示，3.1:62
 另参见 integer type，整数类型
hierarchy，层次，
 另参见 derived class，派生类；inheritance，继承
 class mechanism support of，类机制的支持，
 3.15:106
 exception as class hierarchy，类层次形式的异常，19.2.1:845–846
 in C++ standard library，C++标准库中的，
 19.2.8:855–858
 in multiple and virtual inheritance (chapter)，多继承和虚拟继承中的， **18:**790–834
 multiple inheritance，defining a hierarchy，多继承，定义一个层次，18.2:794–798
 virtual inheritance，defining a hierarchy，虚拟继承，定义一个层次，18.5:813–814
 in single inheritance (chapter)，单继承中的，
 17:719–789
 defining a hierarchy，定义一个层次，
 17.1:721–728
 identifying member of a hierarchy，确定层次的成员，**17.2:**728
horizontal tab (\t) escape sequence，水平制表转义序列，3.1:62

I

I/O，输入/输出，
 参见 iostream
identifier，标识符，3.2.2:68

另参见 name，名字
if statement，if 语句， **5.3**:163–170
conditional operator as alternative to，可作替换的条件操作符，4.7:131
dangling else problem，空悬else问题，5.3:165
#ifdef directive，#ifdef 指示符， 1.3:10
另参见 preprocessor directive，预处理指示符
#ifndef directive，#ifndef 指示符， 1.3:10
另参见 preprocessor directive，预处理指示符
ifstream class，ifstream 类， 20:869
另参见 iostream
ignore() function，函数， 20.3:887–888
另参见 iostream
implicit type conversion，隐式类型转换，4.14.1:147
另参见 conversion，转换
#include directive，#include 指示符，1.3:10
另参见 preprocessor directive，预处理指示符
linkage directive use with，用于链接指示符，7.7:304
using directive use with，用于 using 指示符，2.7:52，8.6.3:363–364
and namespace std，和名字空间 std，8.6.4:366
includes() generic algorithm，includes() 泛型算法， A:941
inclusion compilation model，包含编译模式，
另参见 compilation model，编译模式
for class template，类模板的，16.8.1:696
for function template，函数模板的，10.5.1: 421
increment (++) operator，递增操作符，
built-in，内置，4.5:126–127
overloaded operator，重载操作符，15.7:623–626
postfix form，后置形式，4.5:127，623
prefix form，前置形式，4.5: 127，624
inequality (!=) operator，不等于操作符，
另参见 operator，操作符
built-in operator，内置操作符，4.3:120–122
infinite，无限的，
另参见 control flow，控制流
loop，循环，3.4.1:77，6.8:232
recursion，递归，**7.5**:301
information hiding，信息隐藏，2.3:26，**13.1.3**:507
另参见 access，访问；base class，基类；class member，类成员
inheritance，继承
另参见 base class，基类；derived class，派生类；hierarchy，层次
class scope under，类域，18.4:806–811
composition vs.，组合与，18.3.1:802–803
exception handling and，异常处理与，19.2:845–859
function overload resolution and，函数重载解析与，19.3:864–866
multiple，多，18.1-18.2:790–798
class scope under，类域，18.4.1:809–811
(example)，（例子），**18.6**:823–834
motivation for，动机，18.1:790–793
polymorphism and，多态与，17.1.1:724–726，17.5:752–753
protected inheritance，protected继承，**18.3.3**:804
public，private，and protected，public、private和protected继承，**18.3**:800–806
RTTI use of，RTTI的使用，19.1:835
dynamic_cast，19.1.1:836–840
typeid，19.1.2:840–842
single (chapter)，单继承，**17**:719–789
use of (chapter)，继承的使用，**19**:835–859
virtual，虚拟继承，**18.5**:813–821
(example)，（例子），18.6:823–834
motivation for，动机，18.1:790–793
initialization，初始化，3.2.3:70
另参见 assignment，赋值；constructor，构造函数
array，数组，2.1:19，3.9:94–95
dynamically allocated，动态分配的，8.4.3:345
dynamically allocated，of class，动态分配的，类的，14.4.1:583–585
multidimensional，多维的，3.9.1:96–97
of function pointer，函数指针的，**7.9.4**:318
with another array prohibited，禁止用另数组，3.9:95
assignment compared with，与赋值的比较，4.4:123
auto_ptr object，auto_ptr 对象，8.4.2:841–344
class，类，*参见* constructor，构造函数

class member，类成员，*参见* constructor，构造函数
complex number，复数，4.6:128–129
function pointer，函数指针，7.9.2:317
exception specification impact on，异常规范的影响，11.4.1:465
to overloaded function，重载函数的，9.1.6:377
memberwise，按成员，17.6:772–774
另参见 constructor，构造函数
object，对象，
automatic，自动，8.3.1:335
automatic，compared with static local object，自动，与静态局部对象的比较，8.3.3:337
constant，常量，3.5:84
dynamically allocated，动态分配的，8.4.1:339
global default initialization，全局缺省初始化，8.2.1:331
static local，静态局部，8.3.3:337–338
reference，引用，3.6:86
string，字符串，3.4.2:79–81
contrasted with C-style string，与C风格字符串的比较，3.4.2:78
vector，向量，3.10:100
compared with built-in array，与内置数组的比较，3.10:100
inline function，内联函数，7.1:278
advantage of，优势，7.6:303
declaration，声明，7.6:303
of function template as，函数模板的，10.1:410
definition in header file，头文件中的定义，8.2.3:333
function object and inline operator()，函数对象和内联 operator()，12.2:474，12.3:481
member function，non-inline vs.，成员函数，非内联与，13.3.1:512–513
performance problem with，性能问题，8.2.3:334
inner_product() generic algorithm，inner_product() 泛型算法，A:942
inplace_merge() generic algorithm，inplace_merge() 泛型算法，A: 943
input，输入，
参见 iostream
InputIterator，12.4.6: 493
另参见 container，容器；iterator，迭代器
insert() operation，insert()操作
of map container type，map容器类型的，6.12.1:249–250
of multiset and multimap container type，multiset 和 multimap容器类型的，6.15:268
of set container type，set容器类型的，6.13.1:256
of sequence container type，序列容器类型的，6.6:224–226
of string type，字符串类型的，6.11:289
inserter() function adaptor，inserter()函数适配器，6.13.1:256，12.4.1:489
另参见 function object，函数对象
instantiation，实例化，
另参见 class template，类模板；function template，函数模板
class template，类模板，**16.2**:671–679
explicit instantiation declaration，显式实例声明，
class template，类模板，**16.8.3**:699–700
function template，函数模板，10.5.3:423–424
function template，函数模板，10.2:411–414
explicit template argument，显式模板实参，10.4:417–420
overload resolution with，用于重载解析，10.8:430–436
template argument deduction，模板实参推演，10.3:414–417
point of，点，
class template，类模板，16.11:706–707
class template member function，类模板成员函数，16.11:706–707
function template，函数模板，10.9:440
integer type，整值类型，3.1:61
另参见 arithmetic，算术；type，类型
char type，char 类型，2.1:18，3.61
character constant，字符常量，3.1:62
data type，数据类型，2.1:18，3.1:61

enumerator as grouping of integral constant，用作整值常量分组的枚举成员，3.8:91
int type，int 类型，3.1: 61
literal constant notation，文字常量表示，
decimal notation，十进制表示，3.1:62
hexadecimal notation，十六进制表示，3.1:62
L suffix notation，L后缀表示，3.1:62
octal notation，八进制表示，3.1:62
U suffix notation，U后缀表示，3.1: 62
long type，long类型，3.1: 61
promoting bool constant to，将bool常量提升为，4.3:120
integral promotion，整值提升，4.14.2:148
during function overload resolution，函数重载解析中的，9.3.2:386–388
short type，short类型，3.1:61
standard conversion，标准转换，4.14:146
during function overload resolution，函数重载解析中的，9.3.3:388–389
wide-character constant，宽字符常量，3.1:63

iomanip header file，iomanip 头文件， 3.15:112，20.2.1:882，20.9:913，20.9:917

iostream，1.5:15，**20:**868
>>，input operator，输入操作符，1.5:15–16，20:868–869，**20.2:**876
>>，overloading，input operator，重载，输入操作符，**20.5:**895
<<，output operator，输出操作符，1.5:15–16，20:868–869，**20.1:**872
<<，overloading，output operator，重载，输出操作符，**20.4:**891
buffer，缓冲区，20.9:915
tie()，20.9:915
tying ostream to istream，20.9:915
unitbuf，20.9:915
condition state，条件状态，**20.7:**906
<fstream> header file，<fstream>头文件，1.5.1:16，20:869
bad()，20.7:906
clear()，20.6:905，20.7:907
eof()，20.7:906
fail()，20.7:906
good()，20.7:906
ios_base::badbit，20.7:907
ios_base::eofbit，20.7:907
ios_base::failbit，20.7:907
ios_base::goodbit，20.7:907
rdstate()，20.7:907
setstate()，20.5:896，207:907
cerr，standard error，标准错误，1.5:15，20:868
cin，standard input，标准输入，1.5:15，20: 868
cout，standard output，标准输出，1.5:15，20: 868
file input/output，文件输入和输出，**1.5.1:**16–17，**20.6:**897–906
<fstream> header file，<fstream>头文件，1.5.1:17，20:869
close()，20.6:901
fstream class，file I/O，fstream 类，文件 I/O，1.5.1:17，20:869，20.6:901–905
ifstream class，file input，ifstream 类，文件输入，1.5.1:17，20:867，20.6:899–901
ios_base::app，20.6: 898
ios_base::beg，20.6:902
ios_base::cur，20.6:902
ios_base::end，20.6:902
ios_base::in，20.6:901
ios_base::out，20.6:897
ofstream class，file output，ofstream 类，文件输出，1.5.1:17，20:869，20.6:897–898
open()，20.6:900
seekg()，20.6:902，20.6:902，20.6:904
seekp()，20.6:902
tellg()，20.6:902，20.6:904
tellp()，20.6:902
verify file is open，判断文件是否打开，20.6:898
format state，格式状态，**20.9:**911
boolalpha，20.1:874，20.9:911，20.9:916
dec，20.9:912，20.9:917
endl，1.5:20，20.1:872，20.9:915
ends，20.9:915，20.9:917

fixed，20.9:914，20.9: 917
flush，20.9:915，20.9: 917
hex，20.9:912，20.9: 917
left，20.9:916，20.9: 917
noboolalpha，20.9:912，20.9:916
noshowbase，20.9:913，20.9: 916
noshowpoint，20.9:914，20.9: 916
noshowpos，20.9:916
noskipws，20.2:879，20.9:914，20.9:917
nouppercase，20.9:912，20.9:914，20.9: 917
oct，20.9:912，20.9:917
precision()，20.9:913
right，20.9:916，20.9:917
scientific，20.9:914，20.9:917
setf()，20.9:911
setfill，20.9:916，20.9:917
setprecision()，20.9:913，20.9:917
setw()，20.2.1:882，20.9:914，20.9:916，20.9:917
showbase，20.9:912，20.9:916
showpoint，20.9:914，20.9:917
showpos，20.9:916
skipws，20.9:915，20.9:917
unsetf()，20.9:911
uppercase，20.9:912，20.9:914，20.9:917
ws，20.9:917
input，输入，
>>，1.5:15–16，20:868–869，**20.2:**876
>>，overloading，重载，**20.5:**895
C-style character string，C风格字符串，20.2.1:880–881
end-of-file (EOF)，文件结束，1.5:16，20.2:877，20.3:886
gcount()，a count of the characters read，gcount()，读入字符计数，20.3:887
get()，20.2:879，20.3:886–888，20.6:904
get vs. getline()，get 和 getline()，20.3:888
getline()，6.7:228，20.3:886，20.3:888–890
ifstream class，reading from a file，ifstream 类，读文件，20:869
ignore()，20.3:887–888
input error，输入错误，20.2:877，20.2:878，20.5:896，20.7:906–907
istream class，istream 类，20:868
istream as false，结果为 false，20.2:877，20.2:878
istream_iterator，6.5:223，6.13.1:306，**12.4.4:**491–492，20.2:879
istringstream class，reading from a string，istringstream 类，读字符串，20:871
peek()，20.3:890
putback()，20.3:890
read()，reading byte，read()，读字节，20.3:889
standard input (cin)，标准输入，1.5:15，20:868
string，字符串，**20.2.1**:880
unget()，push back one character，unget()，退回一个字符，20.3:890
white space，空格，20.2:879
<iostream> header file，<iostream>头文件，1.5:15，20:868
iostream iterator，iostream 迭代器，**12.4.3:**490
manipulator，操纵符，1.5:15，20.9:911–917
另参见 iostream/format state，iostream/格式状态
<iomanip> header file，<iomanip>头文件，3.15:112，20.2.1:882，20.9:913，20.9:917
predefined (table)，预定义（表20.1），20.9:916–917
output，输出，
另参见iostream/format state，iostream/格式状态
<<，1.5:15–16，20:868–869，**20.1:**872
<<，overloading，重载，**20.4:**891
bool，20.1:874
C-style character string，C风格字符串，20.1:873
ofstream class，ofstream 类，20:869
ostream class，ostream 类，20:868
ostream_iterator，**12.4.5:**492，20.1:875
put()，20.2:879，20.3:886
standard error (cerr)，标准错误（cerr），

1.5:15，20:868
standard output (cout)，标准输出(cout)，1.5:15，20:868
write(), writing byte，write()，写字节，20.3:889
string stream，字符串流，**20.8**:908
<sstream> header file，<sstream>头文件，20:871，20.8:908
collect nonfatal diagnostic errors，非致命的诊断错误，20.8:909
conversion，string to numeric，转换，字符串到数值，20.8:909
data formatting，数据格式化，20.8:909
istringstream class，istringstream 类，20:871，20.8:909
ostringstream class，ostringstream 类，20:871，20.8:908–909
str()，20.8:908
stringstream class，stringstream 类，20:871
wcerr，20:871
wcin，20: 871
wcout，20: 871
iostream iterator，iostream 迭代器，12.4.3:490
另参见 iostream；iterator，迭代器
isalpha() function，isalpha()函数，5.4:174，6.10:239
isdigit() function，isdigit()函数，6.10:239
ispunct() function，ispunct()函数，6.10:239
isspace() function，isspace()函数，6.10:239
istream_iterator，6.5:223，6.13.1:257，12.4.4:491，20.2:879
另参见 iostream；iterator，迭代器
special end-of-stream object，专门的流结束对象，12.4.4:491
istringstream class，istringstream 类，20:871
另参见 iostream
<iterator> header file，<iterator>头文件，12.4.3:490
iterator，迭代器，6.5:221–224，**12.4**:488
另参见 pointer，指针；container type，容器类型；generic algorithm，泛型算法
accessing container element with，用于访问容器元素，6.5–6.6.1:221–226
map element，map元素，6.12:297–304
multiset and multimap element，multiset和multimap元素，6.15:267–269
set element，set元素，6.13:256–257
string，3.4.2:81
vector element，vector元素，2.8:55–57，3.10:102
advance one element，指向下一个元素，2.8:70，3.10:102
back_inserter，12.4.1:489
begin()，accessing container element using，begin()，用于访问容器元素，2.8:69，6.5:221
categories of iterator，迭代器的分类，**12.4.6**:493
BidirectionalIterator，12.4.6:493
ForwardIterator，12.1:469，12.4.6:493
InputIterator，12.4.6:493
OutputIterator，12.4.6:493
RandomAccessIterator，12.4.6:493
const_iterator，6.5:222，12.4:488
container use，用于容器，2.8:56，6.5:221
definition，定义，2.8:56，6.5:221
dereference，解引用，2.8:56，3.10:102
difference_type，6.13.1:257
end()，accessing container elements using，end()，用于访问容器元素，2.8:55，6.5:221
front_inserter，12.4.1:489
generic algorithm use，用于泛型算法，12.1:469，12.4:488–494
iterator category requirement，迭代器分类的要求，12.5:494
insert iterator，插入迭代器，**12.4.1**:488
inserter，12.4.1:489
iostream iterator，iostream 迭代器，**12.4.3**:490
istream_iterator，**12.4.4**:491
ostream_iterator，**12.4.5**:492
iterator adaptor，迭代器适配器，12.2:472，12.4.1:489
iterator arithmetic，迭代器算术，6.5: 222
left-inclusive interval notation([ ])，左闭合区间表示，12.5: 494
reverse iterator，反向迭代器，**12.4.2**:489
sentinel，哨兵，12.1:469

J

Japanese，日语，
wide-character literal support，宽字符文字支持，3.1:63
wide string literal support，宽字符串文字支持，3.1:63

K

keyword，关键字，（表3.1），3.2.2:68

L

L prefix，L前缀，
wide-character literal notation，宽字符文字表示，3.1:63
wide string literal notation，宽字符串文字表示，3.1:63
L suffix，L后缀，
floating point constant notation，浮点常量表示，3.1:62
long integer literal constant notation，long 整数文字常量表示，3.1:62
lazy error detection，迟缓错误检测，17.4.4:749
left-inclusive interval ([，])，左闭合区间，12.5:494
另参见 iterator，迭代器
less_equal function object，less_equal 函数对象，12.3.3:458
less function object，less 函数对象，12.3.3:458
less than (<) operator，小于操作符，
arithmetic data type support of，算术数据类型支持，2.1:18，4.3:120
requirement for container element type，容器元素类型必须支持，6.4:220
lexigraphical ordering，字典顺序排序，
comparing string，比较字符串，6.11:244
in permutation generic algorithm，排列组合泛型算法中的，12.5.4:496
in relational generic algorithm，关系泛型算法中的，12.5.7:496
lexicographical_compare() generic algorithm，lexicographical_compare() 泛型算法，A:945
lifetime，生命期，8.2:330
另参见 object，对象；scope，域
dynamically allocated object，动态分配的对象，8.4:338
auto_ptr，impact on，auto_ptr，影响，8.4.2:341–344
vs. pointer to，与指针，8.4.1:340
local object，局部对象，
automatic object，自动对象，8.3.1:335
automatic vs. static，自动与静态，8.3:335，8.3.3:337–338
problem as return value for reference return type，引用返回类型作返回值的问题，7.4:299
stack unwinding impact，11.3.2:459
scope and (chapter)，**8**:325–368
limits header file，limits 头文件，4.2:119
__LINE__，1.3:12
linkage directive，链接指示符，7.7:304–306
function pointer use，用于函数指针，7.9.6:322–323
overloading consideration，重载的考虑，9.1.5:376–377
<list> header file，<list>头文件，6.4:217
list container type，list容器类型，6.4:217–220
另参见 container type，容器类型
assignment，赋值，6.6.2:227
constraints on type support，类型支持上的限制，6.4:220
criteria for choosing，选择准则，**6.2**:254
definition，定义 ，**6.4**:217
deletion，删除操作，6.2:213，**6.6.1**:226
element，small vs. large，元素，小和大，6.3:215–216
insertion，插入，6.2:213，6.3:216，6.4:218，6.6:225
insertion and access requirement，插入和访问的必要条件，6:209，6.2:213–214
generic algorithm and，泛型算法和，6.6.3:227
random access generic algorithm not applicable，随机访问泛型算法不适用，12.6:497
iterator and，迭代器和，6.5:222
random access iterator not possible with，随机访问迭代器不可用，12.4.6:494
member function vs. generic algorithm，成员函数与泛型算法，**12.6**:497–500
member function，成员函数，

begin()，6.5:221
empty()，6.4:218
end()，6.5:221
erase()，6.6.1:226
insert()，6.6:225
merge()，12.6.1:498
pop_back()，6.6.1:226
push_back()，6.4:218
push_front()，6.4:218
remove()，12.6.2:498
remove_if()，12.6.3:498
resize()，6.4:219
reverse()，12.6.4:499
sort()，12.6.5:499
splice()，12.6.6:499
swap()，6.6.2:227
unique()，12.6.7:500
object size performance impact，对象大小对性能的影响，6.3:214
random access，随机访问，6.2:213，12.6:497
relational operator，关系操作符，6.4:219
storage，doubly-linked，内存区域，双向链接的，6.2:213
traversal，遍历，6.2:213 *另参见* iterator，迭代器
vector compared with，与vector的比较，6.2:213
literal constant，文字常量，3.1:63
C-style character string，C风格的字符串，3.1:63
as array of const characters，用作 const 字符数组，3.1:62
character literals compared with，与字符文字的比较，3.1:63，3.9:115
character，字符，3.1:62
wide-character，宽字符，3.1:62
E suffix，E后缀，3.1: 62
F suffix，F后缀，3.1: 62
floating point，浮点，3.1: 62
integer，整数，3.1: 62
L suffix，L后缀，3.1: 62
numeric，数值，3.1: 62
U suffix，U后缀，3.1: 62
variables compared with，与变量的比较，3.2.1: 66
local class，局部类，13.12:562–564
另参见 class，类
local object，局部对象，3.2.3:69，**8.3**:335
automatic，自动，8.3.1:335
register，寄存器，8.3.2:336
problem as return value for reference return type，引用返回类型作返回值的问题，7.4:299
static，静态，8.3:335，**8.3.3**:337
local scope，局部域，8.1:325，**8.1.1**:327
accessing global scope member hidden in，隐藏其中的访问全局域成员，8.5.2:353
name resolution in，其中的名字解析，8.1.1:327
namespace names hidden by local object，局部对象隐藏的名字空间名字，8.5.3:355
try block as a，用作局部域的try块，11.2:454
locale header file，locale 头文件，6.10:240
localization，局部化，
constant object for，常量对象，3.5:83
unnamed namespace use，用于未命名名字空间，8.5.6:359
global object impact on，全局对象影响，7.4.1:300
header file and，头文件和，8.2.3:333
locality of declaration，声明的局部性，5.2:161
logical built-in operator，逻辑内置操作符，4.3:120
AND (&&) operator，与操作符，4.1:117，4.3:120
NOT (!) operator，非操作符，4.3:120
OR (| |) operator，或操作符，4.3:120
logical function object，逻辑函数对象，12.3.4:485
另参见 function object，函数对象
long，3.1:61
另参见 integer type，整数类型
long double，3.1:61
另参见 floating point type，浮点类型
loop，循环，1.2. 1:9，2.1:19，5.1: 159–160
另参见 control flow，控制流；recursion，递归
statement，语句，
for，**5.5**:176–180
do-while，**5.7**:182–183
while，**5.6**:180–181
termination，终止，

break statement use，用于 break 语句，**5.8:**183–184
continue statement use，用于 continue 语句，**5.9:**184–185
termination error，终止错误，
stopping condition error，停止条件错误，6.8:232
infinite loop，无限循环，3.4.1:77，6.8:232
lower_bound() generic algorithm，lower_bound() 泛型算法，A:947
lvalue，左值，3.2.1:66
另参见 conversion，转换；function overload resolution，函数重载解析
assignment operation requirement，赋值操作要求，4.4:123
lvalue transformation，左值转换，9.3.1:386
function template argument deduction，函数模板实参推演，10.3:414
ranking in function overload resolution，函数重载解析中的分级，9.4.3:394–400
lvalue-to-rvalue conversion，左值—右值转换，9.3.1:382–384
as function return value，用作函数返回值，7.4:299–300

M
macro，宏，
参见 preprocessor macro，预处理器宏
main()，1.2:4
command line option handling，命令行选项处理，7.8:306–315
make_heap() generic algorithm，make_heap() 泛型算法，A:981
manipulator，操纵符，
参见 iostream
<map> header file，<map> 头文件，6.12:247，6.15:267
map container type，map容器类型，6.12:247
另参见 container type，容器类型；multimap container type，multimap容器类型
definition，定义，6.12.1:247
deleting an element，删除一个元素，**6.12.5:**255–256
generic algorithm，constraint using，泛型算法，使用限制，12.4.6:493，12.6:497
inserting element，插入元素，6.12.1:248
insertion using subscript operator，用下标操作符插入，6.12.1:248
member function，成员函数，
count()，6.12.2:251
erase()，6.12.5:255
find()，6.12.2:251
insert()，6.12.1:249
size()，6.12.3:252
program example，**程序范例，6.12.4:**253–255
random access iterator not possible with，随机访问迭代器不可用，12.4.6:494
reordering not possible，重新排序不可能，12.6:497
retrieving an element，获取元素，6.12.2:251
retrieval with subscript operator，用下标操作符获取，6.12.2:251
retrieval with count() or find()，用count()或find()获取，6.12.2:251
set compared with，与set的比较，6.12:247
traversal，遍历，6.12.3::252
map::value_type，6.12.1:249
max() generic algorithm，max() 泛型算法，A:948
max_element() generic algorithm，max_element() 泛型算法，A:948
member，成员，
参见 class member，类成员；namespace，名字空间
memory，内存，
allocation，分配，*参见* dynamic memory allocation，动态内存分配
deallocation，释放，*参见* dynamic memory deallocation，动态内存释放
memory header file，memory 头文件，8.4.2:342
merge()
generic algorithm，泛型算法，A:950
list container type member function，list容器类型成员函数，12.6:497
method，方法，
参见 member function，成员函数
min() generic algorithm，min() 泛型算法，A:948
min_element() generic algorithm，min_element() 泛型算法，A:949
minus (-) operator，减法操作符，2.1:18，

4.2:118
另参见 arithmetic，算术；operator，操作符
compound assignment (-=) operator，复合赋值操作符，4.4:126
minus function object，minus 函数对象，12.3.2:484
mismatch() generic algorithm，mismatch() 泛型算法，A:951
models，模式，
参见 compilation model，编译模式
modulus function object，modulus 函数对象，12.3.2:484
modulus (%) operator，取模操作符，4.2:118
另参见 arithmetic，算术；operator，操作符
compound assignment (%=) operator，复合赋值操作符，4.4:126
multimap container type，multimap容器类型，6.15:267
另参见 container type，容器类型； map container type，map容器类型，
definition，定义，6.15:267
insertion of element，元素的插入，6.15:268
map comparison，与map的比较，6.12.5:255
<map> header file，<map>头文件，6.15:267
member function，成员函数，
count()，6.15:267
equal_range()，6.15:267
erase()，6.15:268
find()，6.15:267
insert()，6.15:269
removal of elements，元素的删除，6.15:268
retrieval of elements，元素的获取，6.15:267
subscript operator not supported，下标操作符不支持，6.15:269
traversal，遍历，6.15:267–268
multiple inheritance，多继承，
参见 inheritance，继承
multiplication (*) operator，乘法操作符，2.1:18，4.2:118
另参见 arithmetic，算术；operator，操作符
compound assignment (*=) operator，复合赋值操作符，4.4:126
multiplies function object，multiplies 函数对象，12.3.2:484
multiset container type，multiset容器类型，6.15:267
另参见 container type，容器类型； set container type，set容器类型
definition，定义，6.15:267
insertion of elements，元素的插入，6.15:268
set comparison，与set的比较，6.13:256
<set> header file，<set>头文件，6.15:267
member function，成员函数，
count()，6.15:267
equal_range()，6.15:267
erase()，6.15:268
find()，6.15:267
insert()，6.15:269
removal of elements，元素的删除，6.15:268
retrieval of elements，元素的获取，6.15:267
subscript operator not supported，下标操作符不支持，6.15:269
traversal，遍历，6.15:267–268
mutable data member，易变数据成员，13.3.6:520–521
mutation generic algorithm，异变泛型算法，12.5.6:496
另参见 generic algorithm，泛型算法

N

name，名字，3.2.2:68
另参见 namespace，名字空间；scope，域
name resolution，名字解析，8.1:325
in class scope，类域中的，**13.9.1:**548–550，13.11:560–562
in class template definition，类模板定义中的，16.11:705
in function template definition，函数模板定义中的，10.9:437
in local scope，局部域中的，8.1.1:327
in nested class scope，嵌套类域中的，**13.10.1:**557–559
namespace alias，as synonym for namespace name，名字空间别名，用作名字空间名字的同义词，**8.6.1:**361
naming class member，命名类成员，13.1:503–506
base class member，基类成员，17.3:736–743
overloaded operator name，重载操作符的名字，**15.1.2:**611–612

qualified name，限定修饰名字，8.5.2:353
- for class static member，类静态成员的，13.5:526–529
- for class template as namespace member，类模板的，用作名字空间成员，16.12:707–709
- for class as namespace member，类的，用作名字空间成员，13.11:559–562
- for function template as namespace member，函数模板的，用作名字空间成员，10.10:445
- for nested namespace member，嵌套名字空间成员，8.5.3:355

scope of a declaration，声明的域，8.1:326

template parameter name，模板参数的名字，
- for class template，类模板的，16.1:667–668
- for function template，函数模板的，10.1:406–410

typedef，as synonym，typedef，用作同义词，3.12:104

variable name，变量名，3.2.2:68

namespace，名字空间，8.5:350
- *另参见* name，名字；scope，域；using declaration，using 声明；using directive，using 指示符
- alias，别名，**8.6.1:**361
- definition，定义，**8.5.1:**351
- global namespace，全局名字空间，8.1:325
  - accessing hidden member with scope operator，用域操作符访问隐藏成员，8.5.2:353
  - name space pollution problem，名字空间污染问题，8.5:350
- member，成员，
  - class template，类模板，16.12:707–709
  - class，类，13.11:559–562
  - definition，定义，8.5.4:357
  - function template，函数模板，10.10:442-445
  - ODR requirement，ODR要求，8.5.5:358
  - using namespace member，用名字空间成员，**8.6:**361
- nested namespace，嵌套名字空间，**8.5.3:**355
  - overloaded function declaration within，其中的重载函数声明，9.1.4:373–376
- namespace scope，名字空间域，8.1:326
- namespace std，名字空间 std，8.6.4:366
- unnamed namespace，未命名名字空间，**8.5.6:**359
- user-defined namespace，用户定义名字空间，8.5:350

naming conventions，命名习惯，3.2.2:68

negate function object，negate 函数对象，12.3.2:484

Negator function adaptor，取反器函数适配器，12.3.5:486

nested，嵌套的，
- class，类，*参见* class，类
- comment pair，注释对，*参见* comment，注释
- if-else statement，if-else语句，*参见* if statement，if 语句
- namespace，名字空间，*参见* namespace，名字空间

new expression，new 表达式，4.9:134
- *另参见* dynamic memory allocation，动态内存分配:218
- allocator class use，(footnote)，用于分配器类，（页下注），6.4:218
- array，数组，8.4.3:345
  - of class，类的，14.4.1:584，15.8.1:629–631
- object，对象，8.4.1:339
  - class object，类类型，15.8:626–633
  - const object，const对象，8.4.4:347
- placement new expression，定位new表达式，8.4.5:347
  - class object，类类型，15.8.2:631–633
- simulating virtual new operator，模拟虚拟new操作符，17.5.7:768–770

newline (\n) escape sequence，换行转义序列，3.1:62

next_permutation() generic algorithm，next_permutation() 泛型算法，A:952

not_equal_to function object，not_equal_to 函数对象，12.3.3:484
- *另参见* function object，函数对象

not1() function adaptor，not1()函数适配器， 12.3.5:486

not2() function adaptor，not2()函数适配器， 12.3.5:486

nth_element() generic algorithm，nth_element() 泛型算法，A:953

null character，空字符，
string literal termination by，字符串文字结束，3.1:63
null pointer value，空指针值，9.3.3:389–390
as operand to delete expression，用作 delete 表达式的操作数，8.4.1:339
null statement，空语句，5.1:159
numeric data type，数值数据类型，3.1:61
另参见 type，类型
numeric generic algorithm，算术泛型算法，12.5.5:496
另参见 generic algorithm，泛型算法
numeric header file，numeric 头文件，12.5:495，12.5.5:496
numeric literal constant，数值文字常量，3.1:61
另参见 literal constant，文字常量

O

object，对象，
另参见 dynamic memory allocation，动态内存分配；dynamic memory deallocation，动态内存释放；lifetime，生命期；variable，变量
automatic object，自动对象，**8.3.1**:335
declaration with register，用register声明，8.3.3:337
problem as return value for reference return type，引用返回类型返回值的问题，7.4:299
stack unwinding and，栈展开和，11.3.2:459
vs. static object，与静态对象，8.3:335，8.3.3:337
const object，const对象，3.5:85
definition，定义，3.2.3:69
use of memory，内存的使用，3.2.1:66–67
literal constant compared with，与文字常量的比较，3.2.1:66
dynamically allocated object，动态分配的对象，8.4.1:339
auto_ptr，impact on，对auto_ptr影响，8.4.2:341
vs. pointer to，与指针，8.4.1:339
exception object，异常对象，11.3.1:456–458
另参见 exception handling，异常处理
function object，函数对象，**12.3**:481
参见 function object，函数对象
global object，全局对象，
function and，函数和，8.2:331
parameter and return value vs.，参数和返回值与，**7.4.1**:300–301
local object，局部对象，**8.3**:325
declaration with static，用static声明，8.3.3:337
namespace member，名字空间成员，8.5:349
variable and，变量和，3.2.1:62，3.2.2:68
object-based programming，基于对象的程序设计，
另参见 class，类
design，设计，2.3:23–32
object-oriented design difference，与面向对象设计的区别，2.4:35
（Part4），（第四篇），503–715
object-oriented programming，面向对象程序设计，
另参见 base class，基类；derived class，派生类；inheritance，继承；polymorphism，多态；virtual function，虚拟函数
design，设计，2.4:32–39，17.1.1:724–728
(Part 5)，（第五篇），717–917
octal literal constant，八进制文字常量，3.1:62
另参见 literal constant，文字常量
ODR (One Definition Rule)，一次定义法则，8.2:330，8.5.5:358
另参见 namespace，名字空间
ofstream class，ofstream类，1.5.1:17，20:869，20.6:897–906
另参见 iostream
operator，操作符
built-in，内置的，
arithmetic，算术的，4.2:118
assignment (=)，赋值，**4.4**:123
binary，二元，4.1:117
bitwise，按位，**4.11**:136
(chapter)，**4**:116–158
class member access (. and ->)，类成员访问（. 和->），2.3:25，13.2:509

comma，逗号，4.10:135
compound assignment，复合赋值，4.4:126
conditional，条件，4.7:131
conditional，if-else shorthand use，条件，if-else的简便写法，5.3:168–169
decrement (--)，递减，4.5:126
dynamic_cast()，**19.1.1**:835–840
equality，等于，4.3:120
function call (())，函数调用，7.1:278
increment (++)，递增，4.5:126
inequality，不等于，4.3:120
logical，逻辑，4.3:120
precedence，优先级，**4.13**:142
relational，关系，4.3:120
scope (::)，域，**8.5.2**:353
sizeof，**4.8**:132
typeid()，**19.1.2**:840–842

function overload resolution and，函数重载解析和，**15.12**:656–657
另参见 function overload resolution，函数重载解析
ambiguity issue，二义性问题，15.12.3:661
candidate function，候选函数，15.12.1:657–660
viable function，可行函数，**15.12.2**:660

overloaded，重载
assignment (=)，赋值，14.7:597–599，15.3:616–618
(chapter)，**15**:605
declared as friend，声明为友元，15.2:614–616
decrement (--)，递减，15.7:622–625
delete，15.8:626
delete[]，15.8.1:629–631
delete placement，delete定位，15.8.2:631–632
design issue，设计问题，15.1.3:612
equality (==)，等于，15.1:606
function call (())，函数调用，15.5:620
function call (()) for function object，函数对象的函数调用，12.3:481，12.3.6:486
increment (++)，递增，15.7:622
input (>>)，输入，**20.5**:895
input (>>)，iostream library support，输入，iostream库支持，20.2:878
member access (->)，成员访问，15.6:621
member vs. nonmember，成员与非成员，15.1.1:608
name of，名字，15.1.2:611
new，15.8:626
new[]，15.8.1:629–630
new placement，new定位，15.8.2:631–632
output (<<)，输出，**20.4**:891
output (<<)，iostream library support，输出，iostream库支持，20.1:872–875
reference parameter，advantage of，引用参数，优势，7.3.2:287–288
subscript ([])，下标，15.4:618

option，选项，
command line，handling，命令行，处理，7.8:306–314

ostream_iterator，12.4.5:492，20.1:875
另参见 iostream；iterator，迭代器

ostringstream class，ostringstream 类，20:871
另参见 iostream

output，输出，
参见 iostream

OutputIterator，12.4.6: 493
另参见 container，容器；iterator，迭代器

overloading，重载
参见 class member function，类成员；function overload resolution，函数重载解析；function，函数；operator，操作符

P

pair class，pair 类，3.14:105
multiple return values use，用于多个返回值，5.3:166

parameter，参数，
for class template，类模板的，*参见* class template，类模板

for function template，函数模板的，*参见* function template，函数模板
for function，函数的，*参见* function parameter，函数参数
partial_sort() generic algorithm，partial_sort() 泛型算法，A:954
partial_sort_copy() generic algorithm，partial_sort_copy() 泛型算法，A:955
partial_sum() generic algorithm，partial_sum() 泛型算法，A:956
partition() generic algorithm，partition() 泛型算法，A:957
peek() function，peek()函数，20.3:890
另参见 iostream
performance，性能，
auto_ptr use，使用auto_ptr ，8.4.2:342
class initialization vs. assignment，类的初始化与赋值，14.5:589–590，14.6.1:596，14.8:602–603
compile-time，编译时，
function template instantiation，函数模板实例化，10.5.3:423
header file size，头文件的大小，8.2.3.333
container，容器，
capacity，容量，6.3:214–217
list vs. vector，列表与向量，6.2:213–214
tradeoff in container selection，容器选择上的权衡，6.2:213–214
exception handling vs function call，异常处理和函数调用，11.5:466–467
function pointer，函数指针，
disadvantage vs. inlining，与内联比较的劣势，12.2:474
function object vs.，函数对象和，12.3:481
function template definition in header file，头文件中的函数模板定义，10.5.2:421
function，函数，
drawback，缺点，7.6:303
inline advantage，内联的优点，2.3:26，7.6:303
pass-by-value argument，按值传递实参，7.3:282–284
recursive function cost，递归函数的开销，7.5:202
return value issue，返回值的问题，7.2.1:280
locality of declaration for class object，类对象声明的局部性，5.2:161–162
memory allocation，内存分配，2.2:21–22
name return value optimization，名字返回值优化，14.8:600–602
reference，引用，
as exception declaration in catch clause，用作catch子句中的异常声明，11.3.1:458
parameter，参数，7.3.1:285
parameter and return type，参数和返回值，7.4:299
register automatic object，寄存器自动对象，8.3.2:336
permutation generic algorithm，排列组合泛型算法，12.5.4:496
另参见 generic algorithm，泛型算法
placement delete，定位delete，
参见 new expression，new表达式
placement new，定位new，
参见 new expression，new表达式
plus function object，plus函数对象，12.3.1:482，12.3.2:484
plus (+) operator，加法操作符，
参见 addition (+) operator，加法操作符
point of instantiation，实例化点
另参见 class template，类模板；function template，函数模板
class template，类模板，16.11:706–707
for their member function，成员函数的，16.11:706–707
function template，函数模板，10.9:440
pointer，指针，3.3:77–74
另参见 dynamic memory allocation，动态内存分配；dynamic memory deallocation，动态内存释放；iterator，迭代器；pointer to member，成员指针
array compared with，与数组的比较，3.9.2:97–99
auto-ptr，*参见* auto-ptr
to class member，类成员的，*参见* pointer to member，成员指针

to const object，const对象的，3.5:84–85
const pointer，const指针，3.5:84–85
dangling pointer，空悬指针，
to automatic object，自动对象的，8.3.2:336
to dynamically deallocated memory，指向动态释放内存的，8.4.1:340
to function，函数的，**7.9**:315–324
另参见 function pointer，函数指针
as iterator，用作迭代器，
generic algorithm use，用于泛型算法，3.10:101–102
to built-in array，内置指针的，6.5:223–224
null pointer value，空指针值，9.3.3:389
as operand to delete expression，用作delete表达式的操作数，8.4.1:340
parameter，参数，7.3:283–284
array parameter relationship to，与数组参数的关系，7.3.3:289–290
reference parameter relationship to，与引用参数的关系，7.3.2:286–289
pointer conversion，指针转换，*参见* conversion，转换
reference compared with，与引用的比较，3.6:86–89
referring to，指向，
array element，数组元素，3.9.2:97
C-style string，C风格字符串，3.4.1:76
class object，use of operation ->，类对象，->操作的使用，13.2:511
dynamically allocated memory，动态分配内存，参见 dynamic memory allocation，动态内存分配
object，对象，3.3:89–92
sizeof() use with，用于sizeof() ，4.8:132–133
this pointer，this指针，**13.4**:521–525
另参见 class member，类成员
vector of pointer，advantage，指针向量，优势，6.3:216
void*，3.3:72
conversion to and from，转换，4.14.3:150
pointer to member，指向成员的指针，
13.6:532–538
pointer to data member，数据成员的指针，13.6.1:534–535，13.6.2:536
pointer to member function，成员函数的指针，13.6.1:535，13.6.2:536–537
pointer to static member，静态成员的指针，13.6.3:538–539
polymorphism，多态， 17.1.1:724–726，17.5:752
另参见 inheritance，继承；virtual function，虚拟函数
pop_back() function，pop_back()函数，
for sequence container，序列容器的，6.6.1:226
pop_heap() generic algorithm，pop_heap() 泛型算法， A:981
preprocessor，预处理器，
comment，注释，
pair (/*，*/)，对，1.4:13–14
single line (//)，单行，1.4:14
constant，常量，
defining on command line，命令行中定义，1.3:11
__cplusplus，1.3:12
__DATE__，1.3:12
__FILE__，1.3:12
__LINE__，1.3:12
__STDC__，1.3:12
__TIME__，1.3:12
directive，指示符，**1.3**:10–13
#define，1.3:10
#endif，1.3:10
#ifdef，1.3:10
#ifndef，1.3:10
#include，1.3:10
macro，宏，1.3:12
assert()，2.4:37
function tempate as safer alternative to，函数模板用作更安全的替代，10.1:405–406
pre-compiled header file，预编译头文件，8.2.3:333
prev_permutation() generic algorithm，
prev_permutation() 泛型算法， A:958
primitive type，基本类型， 1.2:7
另参见 type，类型
(chapter)，**3**:75–139

priority_queue container type，priority_queue容器类型，6.17:323
另参见 container type，容器类型； queue container type，队列容器类型
<queue> header file，<queue>头文件，6.17:271
table of operation，操作表，6.17:272
private，私有，
base class，基类，参见 base class，基类
class member，类成员，参见 class member，类成员
procedural-based programming，基于过程的程序设计，275-276
另参见 exception handling，异常处理；function，函数；function template，函数模板
(Part 3)，（第三篇），275-501
program，程序，1.2:4–8
promotion，提升，4.14:146
另参见 conversion，转换；function overload resolution，函数重载解析
of enumeration type to arithmetic type，枚举类型到算术类型，3.8:92
on argument，实参的，9.3.2:386–388
ranking in function overload resolution，函数重载解析中的分级，9.4.3:399–400
protected，
base class，基类，参见 base class，基类
class member，类成员，*参见* class member，类成员
prototype，原型，7.2:279–281
另参见 function，函数
pure virtual function，纯虚函数，17.5.2:758–759
另参见 abstract base class，抽象基类；virtual function，虚拟函数
push_back() function，push_back()函数，
sequence container，序列容器， **6.4:**217
vector，inserting element into，向量，插入元素，3.10:101–102
push_front() function，push_front()函数，
list container type member function，list容器类型成员函数，6.4:218
push_heap() generic algorithm，push_heap() 泛型算法，A:982
put() function，put()函数，20.3:886
另参见 iostream
putback() function，putback()函数，20.3:890
另参见 iostream

Q
qualification conversion，限定修饰转换，9.3.1:385
另参见 conversion，转换；function overhead resolution，函数重载解析
in function template argument deduction，在函数模板实参推演中的，10.3:415
ranking in function overload resolution，函数重载解析中的分级，9.4.3:401–403
qualifier，限定修饰符，
const，*参见* const
volatile，*参见* volatile
queue container type，队列容器类型，6.17:271–272
另参见 container type，容器类型；priority_queue container type，priority_queue容器类型
<queue> header file，<queue>头文件，6.17:271
table of operation，操作表，（表6.6），6.17:271

R
random_shuffle() generic algorithm，random_shuffle() 泛型算法，A:959
randomAccessIterator，12.4.6:493
ranking，分级，
另参见 function overload resolution，函数重载解析
function template definition，函数模板定义，10.7:430
standard conversion sequence，标准转换序列，9.4.3:399–403
user-defined conversion sequence，用户定义转换序列，19.3.2:862–863
read() function，read()函数，20.3:889
另参见 iostream
readability，可读性，
const qualifier to declare constant，用const修饰符声明常量，3.5:83
function parameter name，函数参数名，7.2.2:281
overloaded function name，重载函数名，9.1.3:372–373

recursive function，递归函数，7.5:302
reference parameter，引用参数，7.3.2:288
separation of exception handler，异常处理代码的分离，11.2:453–455
typedef，3.13:104
in function pointer declaration，函数指针声明中的，7.9.4:318
to container type，容器类型的，6.12.1:249
recursion，递归，7.5:301–302
reference，引用，3.6:86–89
另参见 parameter，参数
array of reference prohibited in，禁止使用引用数组，3.9:95
as exception declaration in catch clause，用作catch子句中的异常声明，11.3.3:460–461
as function return type，用作函数返回值，7.4:297–299
initialization，初始化，3.6:86–89
as exact match conversion，用作精确匹配转换，9.3.4:391–393
ranking during function overload resolution，函数重载解析中的分级，9.4.3:402–403
reference to const，const的引用，3.7:90–91
parameter，参数，3.7:89，7.3:284–289
necessity for operator overloading，操作符重载的必要性，7.3.2:288
passing array as，传递数组作为参数，7.3.3:290
performance advantage，性能上的优势，7.3.1:285
pointer parameter relationship to，与指针参数的关系，7.3.2:286–288
reference to constant，常量的引用，7.3.1:285
pointer compared with，与指针的比较，3.6:86
sizeof() use with，用于sizeof()，4.8:132
register automatic object，register自动对象，8.3.2:336–337
reinterpret_cast operator，reinterpret_cast操作符，4.14.3:152
danger of，危险，4.14.3:152
relational function object，关系函数对象，12.3.3:484
另参见 function object，函数对象
relational generic algorithm，关系泛型算法，12.5.7:496
另参见 generic algorithm，泛型算法
relational operator，关系操作符，2.1:18，**4.3:**120–122
requirement for container element type，容器元素类型必须支持，6.4:219
release() function，release()函数，
auto_ptr object management with，用于auto_ptr对象管理，8.4.2:344
remainder (%) operator，求余操作符，4.2:118
另参见 arithmetic，算术；operator，操作符
compound assignment (%=) operator，复合赋值操作符，4.4:126
remove() generic algorithm，remove() 泛型算法，A:960
remove_copy() generic algorithm，remove_copy() 泛型算法，A:960
remove_if() generic algorithm，remove_if() 泛型算法，A:961
remove _copy_if () generic algorithm，remove _copy_if () 泛型算法，A:961
replace_copy() generic algorithm，replace_copy() 泛型算法，A:963
replace_copy_if() generic algorithm，replace_copy_if() 泛型算法，A:964
replace() generic algorithm，replace() 泛型算法，A:962
replace_if() generic algorithm，replace_if() 泛型算法，A:964
reserve() function，reserve()函数，
setting container capacity with，用于设置容器的容量，6.3:216
reset() function，reset()函数，
bitset class，bitset类，4.12:139
setting an auto_ptr pointer，设置一个auto_ptr指针，8.4.2:343
resize() function，resize()函数，
container resizing with，用于调整容器的大小，6.4:219
resolution，解析，

function overload resolution，函数重载解析，参见 function overload resolution，函数重载解析
name resolution，名字解析，*参见* name，名字

rethrow，重新抛出，11.3.3:459–461
另参见 exception handling，异常处理

return statement，return语句，
compared with throw expression，与throw表达式的比较，11.1:450
function termination with，用于函数终止，7.4:297
implicit type conversion in，其中的隐式类型转换，4.14.1:147

reverse() generic algorithm，reverse() 泛型算法，A:965

reverse_copy() generic algorithm，reverse_copy() 泛型算法，A:965

reverse iterator，反转迭代器，12.4.2:489

rfind() string operation，rfind()字符串操作，6.9:235

rotate() generic algorithm，rotate() **泛型算法**，A:966

rotate_copy() generic algorithm，rotate_copy() 泛型算法，A:966

RTTI (Runtime Type Identification) facility，运行时刻类型识别设施，19.1:835–845
adding to standard RTTI support，添加标准RTTI支持，19.1.3:843
dynamic_cast() operator，dynamic_cast()操作符，**19.1.1:**836–840
vs virtual function call，与虚拟函数调用，19.1.1:836–837
type_info class，type_info类，**19.1.3:**842–843
typeid operator，typeid操作符，**19.1.2:**840–842
typeinfo header file，typeinfo头文件，19.1.2:840，840，19.1.3:842

rvalue，右值，3.2.1:66
另参见 conversion，转换；function overload resolution，函数重载解析
expression evaluation as，表达式的结果为，4.1:116
lvalue-to-rvalue conversion，左值—右值转换，9.3.1:382–384

S

scope，域，8.1:325–330
另参见 lifetime，生命期；name，名字；namespace，名字空间；visibility，可见性
class scope，类域，**13.9:**545–550
class definition and，类定义和，13.1:503
name resolution in，其中的名字解析，**13.9.1:**549–550
nested class scope，name resolution in，嵌套类域，其中的名字解析，**13.10.1:**557–559
under multiple inheritance，多继承中的，**18.4.1:**809–812
under single inheritance，单继承中的，18.4:806–809
under virtual inheritance，虚拟继承中的，18.5.4:820–821
of control variable in condition，条件中控制变量的，5.5:178，8.1.1:327–330
of exception declaration in catch clause，catch子句中的异常声明的，11.3.1:458
global scope，全局域，8.1:323–327
lifetime and (chapter)，生命期和，**8:**325–368
local scope，局部域，**8.1.1:**326–330
name resolution in，其中的名字解析，8.1.1:327
referring to global scope member hidden in，指向局部域中隐藏的全局域成员，8.5.2:354
namespace scope，名字空间域，8.1:325，8.5:349–351
overloading and，重载和，9.1.4:373–376
of template parameter，模板参数的
class template，类模板，16.1:667–668
function template，函数模板，10.1:408–409

scope (::) operator，域操作符，8.5.2:353–354
class static member accessed with，用于类的静态成员的访问，13.5：520–531
class template as namespace member accessed with，用于按名字空间成员访问类模板，16.12:707–708
class as namespace member accessed with，用于按名字空间成员访问类，13.11:559–562

function template as namespace member, accessed with，用于按名字空间成员访问函数模板，10.10:445
global scope member accessed with，用于访问全局域成员，8.5.2:354
nested namespace member accessed with，用于访问嵌套名字空间成员，8.5.3:354–356
search() generic algorithm，search() 泛型算法，A:967
search_n() generic algorithm，search_n() 泛型算法，A:968
separation compilation model，分离编译模式，
另参见 compilation model，编译模式
for class template，类模板的，16.8.2:697–699
for function template，函数模板的，10.5.2:421–423
sequence container，序列容器，
参见 container type，容器类型
set container type，set容器类型，6.13:256
另参见 container type，容器类型，multiset container type，multiset容器类型
cannot preassign a size，不能重新赋值大小，6.13.1:256
definition，定义，6.13.1:256
generic algorithm，constraint using，泛型算法，使用限制，6.13.1:257，12.4.6:494，12.6:497–498
inserting element，插入元素，6.13.1:256
map compared with，与map的比较，6.12:247
member function，成员函数，
count()，6.13.2:257
empty()，6.13.3:258
find()，6.13.2:257
insert()，6.13.1:256
size()，6.13.3:258
random access iterator not possible with，随机访问迭代器不可用，12.4.6:494
reordering not possible，重新排序不允许，12. 6:497
searching for an element，查找元素，6.13.2:257
traversal，遍历，**6.13.3:**257–258
set generic algorithm，set泛型算法，12.5.8:497
另参见 generic algorithm，泛型算法
set header file，set头文件，6.13.1:256
multiset use，用于multiset ，6.15:267
set_difference() generic algorithm，
set_difference() 泛型算法，A:969
set_intersection() generic algorithm，
set_intersection() 泛型算法，A:970
set_symmetric_difference() generic algorithm，set_symmetric_difference() 泛型算法，A:970
set_union() generic algorithm，set_union() 泛型算法，A:971
short type，short 类型，3.5:83
另参见 integer type，整数类型
signature，符号特征，7.2.3:281
另参见 function，函数
sizeof operator，sizeof 操作符， 4.8:132–134
as constant expression，用作常量表达式，4.8:134
pointer type use with，用于指针类型，4.8:134
reference type use with，用于引用类型，4.8:134
sort() generic algorithm，sort() 泛型算法，3.10:99，A:972，A:982
function object use as argument to，函数对象用作实参，12.3.1:483
sorting generic algorithm，排序泛型算法，12.5.2:495
另参见 generic algorithm，泛型算法
specialization，特化，
参见 class template，类模板；function template，函数模板
sstream header file，sstream头文件，20:871
stable_partition() generic algorithm，
stable_partition() 泛型算法，A:973
stable_sort() generic algorithm，stable_sort() 泛型算法，A:974
stack container type，栈容器类型，**6.16:**269
<stack> header file，<stack>头文件，6.16:269
definition，定义，6.16:310
program example，程序范例，17.7:776–784
relational operator，关系操作符，6.16:271
table of operation，操作表，（表6.5），6.16:219
stack unwinding，栈展开，11.3.2:459
另参见 exception handling，异常处理
with destructor call，与析构函数调用，19.2.5:851–852
standard conversion，标准转换，4.14.1:147

另参见 conversion，转换；function overload resolution，函数重载解析
on function argument，函数实参的，9.3.3:388–391
standard conversion sequence，标准转换序列，9.4.3:399–400
ranking in function overload resolution，函数重载解析中的分级，9.4.3:399–403
standard error (cerr)，标准错误，20:868
参见 iostream
standard input (cin)，标准输入，20:868
参见 iostream
standard output，(cout)，标准输出，20:868
参见 iostream
statement，语句，
block，块，5.1:160
break，**5.8:**183–184
switch statement termination use，用于switch语句终止，5.4:173–175
(chapter)，**5:**159–208
compound，复合，**5.1:**160
continue，**5.9:**184–185
declaration，声明，**5.2:**160–163
do-while，**5.7:**182–183
for and while statements compared with，与for和while语句的比较，5.5:176
for，**5.5:**176–180
goto，**5.9:**185–186
if，1.2.1:12，**5.3:**163–170
if-else，conditional operator as alternative to，条件操作符可替代，4.7:131
null statement，空语句，5.1:159
simple，简单，5.1:159–160
switch，**5.4:**170–176
as if-else chain alternative，作为if-else链的替换，5.3:169
default keyword use，使用default关键字，5.4:170–171，173–174
while，1.2.1:9，**5.6:**180–181
for and do-while statements compared with，与for和do-while语句的比较，5.5:176
static_cast operator，static_cast 操作符，
4.14.3:151
danger of，危险，4.14.3:151
implicit conversion compared with，与隐式转换的比较，4.14.3:151
static class member，静态类成员，13.5:525–531
data member，数据成员，13.5:526–530
of class template，类模板的，16.5:687–689
member function，成员函数，**13.5.1:**529–531
pointer to，指针，13.6.3:538–539
static memory allocation，静态内存分配，2.2:21
另参见 dynamic memory allocation，动态内存分配
dynamic memory allocation difference，与动态内存分配的区别，2.2:22
static object，静态对象，
local static object，局部静态对象，8.3.3:337–339
unnamed namespace member compared with global static object，未命名名字空间成员与全局静态对象的区别，8.5.6:360
std namespace，std 名字空间，8.6.4:366–367
__STDC__，1.3:16
STL (Standard Template Library)，标准模板库，
参见 container type，容器类型； generic algorithm，泛型算法；iterator，迭代器
STL-idiom use，STL习惯用法，3.10:101
storage，存储区，
参见 dynamic memory allocation，动态内存分配；dynamic memory deallocation，动态内存释放；object，对象
stream，流，
参见 iostream；string type，string类型
<string> header file，<string>头文件，3.4.2:79
string type，string类型，3.4.2:78
另参见 C-style character string，C风格字符串；istringstream；ostringstream；stringstream
assignment，赋值，3.4.2:80
concatenation，连接，3.4.2:80
conversion to C-style string，转换为C风格字符串，3.4.2:80
definition，定义，3.4.2:97–80
getline()，6.7:228–229
initializing with C-style character string，用C风格字符串初始化，3.4.2:80
input/output，输入/输出，20.2.1:880

member function，成员函数，
append()，6.11:242–243
assign()，6.11:242–243
at()，6.11:244
c_str()，3.4.2:80
compare()，6.10:240，6.11:244–245
empty()，3.4.2:79
erase()，6.9:237–238，6.11:241–242
find()，6.8:231
find_first_not_of()，6.8:236
find_first_of()，6.8:231–236，6.9:237
find_last_not_of()，6.8:236
find_last_of()，6.8:236
insert()，6.11:242
replace()，3.4.2:81，6.10:240，6.11:245–246
rfind()，6.8:235–236
size()，3.4.2:79，6.11:244
swap()，6.11:243
substr()，6.8:233
mix with C-style string，与C风格字符串混合，3.4.2:80
range exception，范围异常，6.11:244
string::npos，6.8:231
string::size_type，6.8:231
string stream，字符串流，**20.8:908**
subscript access，下标访问，3.4.2:81
substring，locating，子字符串，查找，**6.8:231**
stringstream class，stringstream类，20:871
另参见 iostream
subscript ([]) operator，下标操作符，2.1:19，3.9:93–95
另参见 array，数组；container type，容器类型
bitset use，用于bitset，4.12:140
map use，用于map，6.12.1:248
not supported for multiset and multimap，不支持multiset和multimap，6.15:269
overloaded operator，重载操作符，15.5:619
vector use，用于vector，3.10:100–101
substitution generic algorithm，替换泛型算法，12.5.3:496
另参见 generic algorithm，泛型算法
subtraction (-) operator，减法操作符，2.1:18，**4.2:118**
另参见 arithmetic，算术
complex number support，复数支持，4.6:128
compound assignment (-=)，复合赋值，4.4:126
subtype，子类型，
参见 derived class，派生类
suffix for literal constant，文字常量的后缀，
E suffix，floating point exponent literal constant notation，E后缀，浮点幂文字常量记号，3.1:62
F suffix，floating point single precision literal constant notation，F后缀，浮点单精度文字常量记号，3.1:62
L suffix，L后缀
floating point extended precision literal constant notation，浮点扩展文字常量精度记号，3.1:62
long integer constant notation，long整数常量记号，3.1:62
U suffix，integer unsigned literal constant notation，U后缀，整数无符号文字常量记号，3.1:62
swap() generic algorithm，swap() 泛型算法，A:975
swap_range() generic algorithm，
swap_range() 泛型算法，A:975
switch statement，switch语句，5.4:170–175
另参见 control flow，控制流
case keyword use，用于case关键字，5.4:170
default keyword use，用于default关键字，5.4:170–171，173–174
as if-else chain alternative，作为if-else链的替换，5.3:169

T

template keyword，template 关键字，2.5:41
template，模板，
class，类，参见 class template，类模板
function，函数，*参见* function template，函数模板
terminate() function，terminate() 函数，11.3.2:459
另参见 exception handling，异常处理

this pointer，this指针，13.4:521–526
  另参见 class member，类成员
throw expression，throw表达式，11.1:449–452
  另参见 exception handling，异常处理
  handling when not in a try block，不在try块中的处理，11.3.2:459
  rethrow，重新抛出，11.3.3:459–461
  with exception as class hiararchy，类层次形式的异常，19.2.2:846–847
__TIME__，1.3:12
tolower() function，tolower() 函数，6.10:239
toupper() function，toupper() 函数，6.10:239
transform() generic algorithm，transform() 泛型算法，A:977
true keyword，true关键字，3.7:90
try block，try块，11.2:452–455
  另参见 exception handling，异常处理
  function try block，函数try块，11.2:455，19.2.7:854–855
type，类型，
  abstract container type，容器类型，参见 container type，容器类型
  arithmetic type，算术类型，2.1:18–21
    另参见 floating point type，浮点类型；integer type，整数类型
  array type，数组类型，*参见* array，数组
  basic type，基本类型，1.2:8
    (chapter)，**3:**75–139
    bool type，bool类型，2.1:18，3.7:90–91
    operator that evaluate to，结果为bool类型的操作符，4.3:120
  conversion to，during function overload resolution，转换为，函数重载解析中的，9.3.3:388
  built-in type，内置类型，1.2:7
  C-style character string，C风格的字符串，**3.4.1:**76–78
    dynamic array allocation for，动态数组分配，8.4.3:345
  character type，字符类型，*参见* character type，字符类型
  class type，类类型，*参见* class
  complex type，复数类型，*参见* complex number，复数
  const qualifier，const限定修饰符，*参见* const
  container type，容器类型，*参见* container type，容器类型
  enumeration type，枚举类型，*参见* enumeration type，枚举类型
  floating point type，浮点类型，*参见* floating point type，浮点类型
  function type，函数类型，*参见* function，函数
  function pointer type，函数指针类型，*参见* function pointer，函数指针
  integer type，整数类型，
    参见 integer type，整数类型
  modifier，修饰符，
    参见 const
    参见 volatile
  numeric，数值，3.1:61
  parameter type checking，参数类型检查，**7.2.3:**281–282
    ellipse and absence of，省略号和类型检查的挂起，**7.3.6:**295
    multi-file declaration and，多文件声明和，8.2.2:331–332
  pointer type，指针类型，参见 pointer，指针
  primitive，基本，1.2:7
  reference type，引用类型，*参见* reference，引用
  return type，返回类型，*参见* function，函数
  sequence container，序列容器，*参见* container type，容器类型
  string，*参见* string type，string类型
  type checking，类型检查，
    casting danger and motivation，强制转换的危险和动机，4.14.3:151–152
    declaration and type checking，声明和类型检查，3.2.1:67
  type conversion，类型转换，*参见* conversion，转换
  typedef alias for，类型的typedef别名，3.12:103–104
  volatile qualifier，volatile限定修饰符，*参见* volatile
type-safe linkage，类型安全链接，8.2.2:332
  to overloaded function，支持重载函数，9.1.7:378

typedef，3.12:103–104
improving readability for，改善可读性，
nested type within container type，容器类型中的嵌套类型，6.12.1:243
array of function pointers，函数指针数组，7.9.4:318
function pointer return type，函数指针返回类型，7.9.5:322
overloaded function and parameter type，重载函数和参数类型，9.1.2:370–371
typeid() operator，typeid()操作符，19.1.2:840–842
另参见 RTTI
typeinfo header file，typeinfo头文件，19.1.2:840，841，19.1.3:842，843
type_info class，type_info类，19.1.3: 842–843
typename，5.11.1:204
class template parameter use，用于类模板参数，16.1:666–667
function template parameter use，用于函数模板参数，10.1:407

U
unary operator，一元操作符，4.1:117
unexpected() function，unexpected() 函数，11.4:463
另参见 exception handling，异常处理
unget() function，unget() 函数，20.3:890
另参见 iostream
uninitialized，未初始化的，3.2.3:69
automatic object，自动对象，8.3.1:335
global object，全局对象，8.2.1:331
local static object，局部静态对象，8.3.3:336
union，联合，**13.7:**539–543
unique() generic algorithm，unique() 泛**型算法，**A:978
use with vector container type，用于vector容器类型，12.2:472–473
unique_copy() generic algorithm，unique_copy() 泛型算法，A:978
use with vector container type，用于vector容器类型，12.4.3:490
unnamed namespaces，未命名名字空间，8.5.6:359–360
另参见 namespace，名字空间
unwinding stack，展开的栈，11.3.2:459
另参见 exception handling，异常处理
with destructor call，用于析构函数，19.2.5:851–852
upper_bound() generic algorithm，upper_bound() 泛型算法，A:980
user-defined conversion sequence，用户定义转换序列，**15.10:**642–644
另参见 conversion，转换；function overload resolution，函数重载解析
ranking in function overload resolution，函数重载解析中的分级，15.10.4:648–651，19.3.2:862–863
with inheritance，在继承机制下，19.3.2:862–863
user-defined type，用户定义类型，
参见 class，类
using declaration，using声明，8.6.2:362–363
另参见 namespace，名字空间
declaring overloaded function with，用于声明重载函数，9.1.4:373–375
impact on function overload resolution，对函数重载解析的影响，9.4.1:395–397
using directive compared with，与using指示符的比较，8.6.3:364
using directive，using指示符，8.6.3:363–366
另参见 namespace，名字空间
declaring overloaded function with，用于声明重载函数，9.1.4:375–376
impact on function overload resolution，对函数重载解析的影响，9.4.1:396–397
include preprocessor directive use with，用于include预处理器指示符，2.7:52
using declaration compared with，与using声明的比较，8.6.3:364
utility header file，utility头文件，3.14:105

V
variable，变量，3.2:64–71
另参见 object，对象
const variable，const变量，3.5:83–85
declaration as namespace member，声明为名字空间成员，8.5.1:351
global，vs. parameter and return value，全局变量与参数和返回值，**7.4.1:**300–301

literal constant compared with，与文字常量的比较，3.2:66
variable name，变量名，3.2:69
<vector> header file，<vector>头文件，2.8:55，3.10:100，6.4:217
vector container type，vector容器类型，2.8:54–56，3.10:99
另参见 array，数组；container type，容器类型；iterator，迭代器
array idiom，数组习惯，3.10:99–100
array compared with，与数组的比较，2.8:54–58
assignment comparison，赋值比较，3.10:101
assignment，赋值，3.10:101，6.6.2:227
assignment vs. insertion，赋值与插入，2.8:54
capacity，relationship to size，容量，与长度的关系，6.3:214–217
constraint on type support，类型支持上的限制，6.4:220
contiguous memory area，连续内存区域，6.2:213
class object，类对象，14.4.2:585–586
criteria for choosing，选择准则，6.2:213
definition，定义，2.8:55，3.10:100–101，6.4:217
deletion，删除，2.8:56，6.2:213–214，**6.6.1**:226
dynamic growth，动态增长，2.8:54，6.3:214
element characteristics，元素的特性，
large class object，大的类对象，6.3:216
object vs. pointer，对象与指针，6.3:216
small vs. large type，小类型与大类型，6.3:216
generic algorithm use，用于泛型算法，
copy()，12.2:472
find()，12.1:469
for_each()，12.2:476–477
unique()，12.2:472–473
unique_copy()，12.4.3:490–491
increasing size of vector，向量增长长度，6.3:214–217
initialization，初始化，参见definition，定义
insertion，插入，6.2:213，6.3:216，6.4:218，6.6:224
list compared with，与list的比较，6.2:213–214
member function，成员函数，
begin()，2.8:55，3.10:101，6.5:221
empty()，3.10:100，6.4:218
end()，2.8:55，3.10:101，6.5:221
erase()，6.6.1:226
insert()，6.6:224–226
push_back()，3.10:101–102，6.4:218，6.6:224
pop_back()，6.6.1:226
reserve()，6.3:216
resize()，6.4:219
size()，3.10:100
swap()，6.6.3:243
parameter as，用作参数，7.3.4:291–292
relational operator，关系操作符，6.4:219
STL idiom，STL习惯用法，3.10:100–101
subscript operator，下标操作符，2.8:55，3.10:100
traversal，遍历，2.8:55–56，3.10:101–102
viable function，可行函数，9.4.2:397–399
另参见 function overload resolution，函数重载解析
best viable function，最佳可行函数，9.2:380，**9.4.3:399–403**
for call with argument of class type，针对类类型实参的调用，15.10.4:648–651
inheritance and，继承和，19.3.3:864–866
default argument and，缺省实参和，9.4.4:403
for calls to member function，用于成员函数的调用，15.11.3:654–656
for operator function，用于操作符函数，15.12.2:660–661
inheritance and，继承和，19.3.2:864–866
virtual base class，虚拟基类，18.5:813–821
另参见 base class，基类；virtual inheritance，虚拟继承
virtual function，虚拟函数，
called from constructor and destructor，从构造函数和析构函数调用，**17.5.8:**770–771
default argument and，缺省实参和，**17.5.4:**760–762
destructor，析构函数，17.5.5:763–764
exception object and，异常对象和，**19.2.4:**849–851
I/O，**17.5.1:**753–757

in base and derived class，基类和派生类中的，**17.5**:752–772
pure，纯，**17.5.2**:758–759
simulating virtual new operator，模拟虚拟new操作符，17.5.7:768–770
static invocation of，静态调用，17.5.3:759–760

virtual inheritance，虚拟继承，
参见 inheritance，继承
(chapter)，**18**:790–834
class scope under，类域，18.5.4:820–821
defining a hierarchy with，用于定义层次，18.5:813–814
defining base class in，定义基类，18.5.1:815–816
destructor in，其中的析构函数，18.5.3:819–820
initialization in，其中的初始化，18.5.2:816–820
member visibility in，其中的成员可见性，18.5.4:820–821

visibility，可见性，
另参见 name，名字；scope，域
of base class member，基类成员的，
in multiple inheritance，多继承中的，18.4.1:809–811
in single inheritance，单继承中的，18.4:806–809
in virtual inheritance，虚拟继承中的，18.5.3:820–821
of class member，类成员的，13.3.2:513–514，13.9:545–550
inline function requirement，内联函数的必要条件，7.6:303，8.2.3:400–401
of local class member，局部类成员的，13.12:562–564
of nested class member，嵌套类成员的，13.10:551–559
of variable defined in condition，条件中定义的变量，5.5:178，8.1.1:393–394
role in candidate function selection during function overload resolution，函数重载解析中候选函数的角色，9.4.1:394
symbolic constant requirement，符号常量的必要条件，8.2.3:333–334

void type，void类型，
in function parameter list，函数参数表中的，7.2.2:280
pointer to，void类型的指针，4.14.3:149
conversion to，as standard conversion，转换，标准转换，9.3.3: 390–391

volatile，3.13:127
另参见 const
function overload resolution issue，函数重载解析问题，
qualification conversion，限定修饰转换，9.3.1:385–386
ranking of reference initialization，引用初始化的分级，9.4.3:402
member function，成员函数，13.3.5:517–520
overloaded function declaration and volatile parameter type，重载函数声明和volatile参数类型，9.1.2:371–372
use to avoid optimization，用于避免优化，3.13:105

W

wchar_t，
as wide-character literal type，用作宽字符文字类型，3.1:63
wide string literal as arrays of const wchar_t，用作const wchar_t数组的宽字符文字，3.1:63

while statement，while语句， 5.6:180–181
for and do-while statements compared with，与for和do-while语句的比较，5.5:176

write() function，write() 函数，20.3:889
另参见 iostream