9

STL 算法

STL Algorithms

本章描述 C++ 标准程序库提供的所有算法。首先对所有算法及其主要特征进行概述，然后展示每一种算法的确切形式，并运用一个或多个例子说明其用法。

9.1 算法头文件（header files）

要运用C++ 标准程序库的算法，首先必须含入头文件 <algorithm>[1]：

```
#include <algorithm>
```

此文件也包含了一些辅助函数，其中 min(),max(),swap()在 4.4.1 节(p66)和 4.4.2 节(p67)已有详述，迭代器相关函数 iter_swap()在 7.3.3 节(p263)已有详述。

某些 STL 算法用于数值处理，因此被定义于头文件 <numeric>[1]：

```
#include <numeric>
```

C++ 标准程序库的数值相关组件在本书第 12 章讨论。但我决定在这里先讨论数值算法。因为就我的观点而言，它们被视为 STL 算法，远比被用来处理数值更重要。

使用 STL 算法时，你经常需要用到仿函数（functor，或称为 function objects）及函数配接器（function adapters）。它们都被定义于 <functional>[2]之中，本书第 8 章对仿函数和函数配接器有详细的讲解。

```
#include <functional>
```

---

[1] 早期 STL 中，所有算法都定义于 <algo.h>。

[2] 早期 STL 中，仿函数和函数配接器定义于 <function.h>。

9.2 算法概览

本节概述 C++ 标准程序库提供的所有算法。你可以从中体会它们的能力，进而在遇到某些问题时选择最适用的算法。

9.2.1 简介

在第 5 章中，借着整体讲解 STL 的大背景，我曾经介绍过算法。特别是 5.4 节（p94）和 5.6 节（p111）讨论了算法的角色和使用上的一些重要限制。所有 STL 算法都被设计用来处理一个或多个迭代器区间。第一个区间通常以起点和终点表示，至于其它区间，多数情况下你只需提供起点便足矣，其终点可自动以第一区间的元素数量推导出来。调用者必须确保这些区间的有效性，也就是说起点和终点必须指向同一个容器，而且起点位置不得在终点之后，第二（及其它）区间必须有足够的空间。

STL 算法采用覆盖（overwrite）模式而非安插（insert）模式。所以调用者必须保证目标区间拥有足够的元素空间。当然，你也可以运用特殊的安插型迭代器（参见 7.4.2 节，p271）将覆盖模式改变为安插模式。

为了提高灵活性和功效，某些 STL 算法允许使用者传递自定的操作，以便由 STL 算法调用之。这些操作既可以是一般函数，也可以是仿函数；如果其返回值是 `bool`，便称为条件判断式（predicates）。你可以运用条件判断式完成以下工作：

- 对于搜寻算法，你可以传递一个函数或仿函数，指定一个一元判断式作为搜寻准则。该一元判断式用来判断某元素是否符合条件。例如你可以搜寻第一个“小于 50”的元素。
- 对于排序算法，你可以传递一个函数或仿函数，指定一个二元判断式作为排序准则。该二元判断式用来比较两元素。例如你可以传递一个准则，让 `Person` 对象按姓氏排序（参见 p294 实例）。
- 你可以传递一个一元判断式作为准则，判断是否应该对某元素施以某项运算。例如你可以令奇数值元素被移除。
- 你可以为某个数值算法指定一个数值运算。例如你可以让通常用来求“总和(sum) 的 `accumulate()`算法改为求取乘积（product）。

注意，判断式不应该在函数调用过程中改变其自身状态（参见 8.1.4, p302）。

关于算法所用的函数和仿函数，请参考 5.8 节（p119），5.9 节（p124），以及第 8 章。

9.2.2 算法分门别类

不同的算法满足不同的需求。所以，我们可以根据它们的主要目的加以分类。例如某些算法的操作是只读性的，某些算法会改动元素本身，某些则改动元素顺序。本小节简单介绍每种算法的功能，并指出相似算法之间的区别。

为了让人顾名思义，STL 设计者为算法命名时，引入两个特别的尾词：

1. 尾词 `_if`

   如果算法有两种形式，参数个数都相同，但第一形式的参数要求传递一个值，第二形式的参数要求传递一个函数或仿函数，那么尾词 `_if` 就派上了用场。无尾词的那个要求传递数值，有尾词的那个要求传递函数或仿函数。例如，`find()` 用来搜寻具有某值的元素，而 `find_if()` 接受一个被当做搜寻准则的函数或仿函数，并搜寻第一个满足该准则的元素。

   不过并非所有“要求传递仿函数”的算法都有尾词 `_if`。如果算法以额外参数来接受这样的函数或仿函数，那么不同版本的算法就可以采用相同命名（译注：重载，overloaded）。例如当你以两个参数调用 `min_element()`，该算法以 `operator<` 为比较准则，返回区间中的最小元素，但如果你传递第三参数，这个参数会被当做比较准则。

2. 尾词 `_copy`

   这个尾词用来表示在此算法中，元素不光被操作，还会被复制到目标区间。例如 `reverse()` 将区间中的元素颠倒次序，而 `reverse_copy()` 则是逆序将元素复制到另一个区间。

接下来数小节按以下分类方式描述各个 STL 算法：（译注：请注意，不同的书籍对于以下的 modifying 和 mutating 两字意义和用法可能不尽相同）

- 非变动性算法（nonmodifying algorithms）
- 变动性算法（modifying algorithms）
- 移除性算法（removing algorithms）
- 变序性算法（mutating algorithms）
- 排序算法（sorting algorithms）
- 已序区间算法（sorted range algorithms）
- 数值算法（numeric algorithms）

如果某个算法同时隶属多个分类，我会把它放在我认为最贴切的分类中讲述。

非变动性算法（nonmodifying algorithms）

非变动性算法既不改动元素次序，也不改动元素值。它们透过 input 迭代器和 forward 迭代器完成工作，因此可作用于所有标准容器身上。表 9.1 展示 C++ 标准程序库涵盖的所有非变动性算法。还有一些非变动性算法专门用来操作已序（*sorted*）输入区间，参见 p330。

表 9.1 非变动性算法

名称	作用	页次
for_each()	对每个元素执行某操作	334
count()	返回元素个数	338
count_if()	返回满足某一准则（条件）的元素个数	338
min_element()	返回最小值元素（译注：以一个迭代器表示）	340
max_element()	返回最大值元素（译注：以一个迭代器表示）	340
find()	搜寻等于某值的第一个元素	341
find_if()	搜寻满足某个准则的第一个元素	341
search_n()	搜寻具有某特性的第一段“n 个连续元素”	344
search()	搜寻某个子区间第一次出现位置	347
find_end()	搜寻某个子区间最后一次出现位置	347
find_first_of()	搜寻等于“某数个值之一”的第一个元素	352
adjacent_find()	搜寻连续两个相等（或说符合特定准则）的元素	354
equal()	判断两区间是否相等	356
mismatch()	返回两个序列的各组对应元素中，第一对不相等元素	358
lexicographical_compare()	判断某一序列在“字典顺序”（lexicographically）下是否小于另一序列	360

最重要的算法之一便是 for_each()，它将调用者提供的操作施加于每一个元素身上。例如你可以用 for_each() 来打印区间内的每个元素。for_each() 也可以用来变动元素（如果你传给它的操作行为会变动元素值的话）。所以它既可说是非变动性算法，也可以说是变动性算法。不过在此情况下，如果有其它算法可以满足你的需求，你最好避免使用 for_each()，毕竟其它那些算法是为特别任务而量身定做的。

有好几个非变动性算法具有搜寻能力。不幸的是，搜寻算法的命名方式却是一团混乱。此外，搜寻算法的命名方式又和 string 搜寻函数的命名方式大相径庭（表 9.2）。为什么会这样？还不是老掉牙的“历史因素”所致。首先，STL classes 和 string classes 乃是各自独立发展设计的。其次 find_end()，find_first_of()，search_n() 算法并不涵盖于早期的 STL。所以最后选择的名称是 find_end() 而不是 search_end()，这完全不是刻意的（说实在话，一旦你钻入细节，像“一致性”这种大局观问题就很容易被忽略）。同样是偶发事件，search_n() 的某一形式竟然违背了原始 STL 的一般性概念。这一问题的描述请见 p346。

表 9.2 String 搜寻函数和 STL 搜寻算法的比较

搜寻	String 函数	STL 算法
某元素第一次出现位置	find()	find()
某元素最后一次出现位置	rfind()	find(),采用逆向迭代器
某子区间第一次出现位置	find()	serach()
某子区间最后一次出现位置	rfind()	find_end()
某数个元素第一次出现位置	find_first_of()	find_first_of()
某数个元素最后一次出现位置	find_last_of()	find_first_of(),采用逆向迭代器
n 个连续元素第一次出现位置		search_n()

变动性算法（modifying algorithms）

变动性算法，要不直接改变元素值，要不就是在复制到另一区间的过程中改变元素值。如果是第二种情况，原区间不会发生变化。表 9.3 列出了 C++ 标准程序库涵括的变动性算法。

最基本的变动性算法是 for_each()（唔，又是它！）和 transform()。两者都可以变动序列中的所有元素值。它们的行为有以下不同点：

- **for_each()**接受一项操作，该操作可变动其参数。因此该参数必须以 by reference 方式传递。例如：

```
void square (int& elem)  // call-by-reference
{
   elem = elem * elem;   // assign processed value directly
}
...
for_each(coll.begin(),coll.end(),  // range
         square);                  // operation
```

- **transform()**运用某项操作，该操作返回被改动后的参数。此间奥妙在于它可以被用来将结果赋值给原元素。例如：

```
int square (int elem)                    // call-by-value
{
   return elem * elem;                   // return processed value
}
...
transform (coll.begin(), coll.end(),     // source range
           coll.begin(),                 // destination range
           square);                      // operation
```

表 9.3 变动性算法（modifying algorithms）

名称	效果	页次
for_each()	针对每个元素执行某项操作	334
copy()	从第一个元素开始，复制某段区间	363
copy_backward()	从最后一个元素开始，复制某段区间	363
transform()	变动（并复制）元素，将两个区间的元素合并	367
merge()	合并两个区间	416
swap_ranges()	交换两区间内的元素	370
fill()	以给定值替换每一个元素	372
fill_n()	以给定值替换 n 个元素	372
generate()	以某项操作的结果替换每一个元素	373
generate_n()	以某项操作的结果替换 n 个元素	373
replace()	将具有某特定值的元素替换为另一个值	375
replace_if()	将符合某准则的元素替换为另一个值	375
replace_copy()	复制整个区间，同时并将具有某特定值的元素替换为另一个值	376
replace_copy_if()	复制整个区间，同时并将符合某准则的元素替换为另一个值	376

transform()的速度稍许慢些，因为它是将操作返回值赋值给元素，而不是直接变动元素。不过其灵活性比较高，因为它可以把某个序列复制到目标序列中，同时变动元素内容。transform()的第二形式可以将两个源区间的元素的组合结果放到目标区间。

严格地说，merge()不算是变动性算法的当然一员。因为它要求输入区间必须已序（*sorted*），所以应该归为“作用于已序区间之算法”（详见 p330）。然而实用上 merge()也可用来合并无序区间——当然其结果也是无序的。不过基于安全考量，你最好只对已序区间调用 merge()。

注意，关联式容器的元素被视为常数，惟其如此，你才不会在变动元素的时候有任何可能违反整个容器的排序准则。因此，你不可以将关联式容器当做变动性算法的目标区间。

除了这些变动性算法，C++ 标准程序库还提供了不少专门用来处理已序（*sorted*）区间的算法。细节详见 p330。

移除性算法（Removing Algorithms）

移除性算法是一种特殊的变动性算法。它们可以移除某区间内的元素，也可以在复制过程中执行移除动作。和变动性算法类似，移除性算法的目标区间也不能是个关联式容器。表 9.4 列出 C++ 标准程序库涵括的所有移除性算法：

表 9.4 移除性算法（Removing Algorithms）

名称	效果	页次
remove()	将等于某特定值的元素全部移除	378
remove_if()	将满足某准则的元素全部移除	378
remove_copy()	将不等于某特定值的元素全部复制到它处	380
remove_copy_if()	将不满足某准则的元素全部复制到它处	380
unique()	移除毗邻的重复元素	381
unique_copy()	移除毗邻的重复元素，并复制到它处	384

注意，移除算法只是在逻辑上移除元素，手段是：将不需被移除的元素往前覆盖（overwrite）应被移除的元素。因此它并不改变操作区间内的元素个数，而是返回逻辑上的新终点位置。至于是否使用这个位置进行诸如“实际移除元素”之类的操作，那是调用者的事情。这个问题的详细讨论请见 5.6.1 节，p111。

变序性算法（Mutating Algorithms）

所谓变序性算法是，透过元素值的赋值和交换，改变元素顺序（但不改变元素值）。表 9.5 列出 C++ 标准程序库涵盖的所有变序性算法。和变动性算法（modifying algorithms）一样，变序性算法也不能以关联式容器作为目标，因为关联式容器的元素都被视为常数，不可更改。

表 9.5 变动性算法（Mutating Algorithms）

名称	效果	页次
reverse()	将元素的次序逆转	386
reverse_copy()	复制的同时，逆转元素顺序	386
rotate()	旋转元素次序	388
rotate_copy()	复制的同时，旋转元素次序	389
next_permutation()	得到元素的下一个排列次序	391
prev_permutation()	得到元素的上一个排列次序	391
random_shuffle()	将元素的次序随机打乱	393
partition()	改变元素次序，使“符合某准则”者移到前面	395
stable_partition()	与 partition() 相似，但保持符合准则与不符合准则之各个元素之间的相对位置	395

排序算法（Sorting Algorithms）

排序算法是一种特殊的变序性算法，但比一般的变序性算法复杂，花费更多时间。事实上排序算法的复杂度通常低于线性算法的[3]，而且需要动用随机存取迭代器。表 9.6 列出所有的排序算法。

表 9.6 排序算法（Sorting Algorithms）

名称	效果	页次
sort()	对所有元素排序	397
stable_sort()	对所有元素排序，并保持相等元素间的相对次序	397
partial_sort()	排序，直到前 n 个元素就位	400
partial_sort_copy()	排序，直到前 n 个元素就位，结果复制于它处	402
nth_element()	根据第 n 个位置进行排序	404
partition()	改变元素次序，使符合某准则的元素放在前面	395
stable_partition()	与 partition()相同，但保持符合准则和不符合准则的各个元素之间的相对位置	395
make_heap()	将一个区间转换成一个 heap	406
push_heap()	将元素加入一个 heap	406
pop_heap()	从 heap 移除一个元素	407
sort_heap()	对 heap 进行排序（执行后就不再是个 heap 了）	407

对排序算法而言，时间往往很重要。所以 C++ 标准程序库提供了多个排序算法。这些算法使用不同的排序方式，有些算法并不对所有元素进行排序。例如 nth_element()在第 n 个元素就位之后即停止排序，对其它元素而言，它只保证凡是小于“已就位之第 n 个元素”的所有元素，都排在前面，大的元素则排在 n 位置之后。如果要对所有元素进行排序，可以考虑以下算法：

- **sort()** 内部采用 quicksort 算法。因此保证了很好的平均性能，复杂度为 n*log(n)，但最差情况下也可能具有非常差的性能（二次复杂度）。

```
/* sort all elements
* - best n*log(n) complexity on average
* - n*n complexity in worst case
*/
sort (coll.begin(), coll.end());
```

[3] 关于复杂度的讨论，请见 2.3 节，p21。

所以如果“避免最差情况”对你是一件很重要的事，你应该采用其它算法，例如接下来要讨论的 partial_sort()或 stable_sort()。

- **partial_sort()** 内部采用 heapsort 算法。因此，它在任何情况下保证 n*log(n) 复杂度。大多数情况下 heapsort 比 quicksort 慢 2~5 倍，所以大多数时候虽然 partial_sort()具有较佳复杂度，但 sort()具有较好的执行效率。partial_sort()的优点是它在任何时候都保证 n*log(n) 复杂度，绝不会变成二次复杂度。

  partial_sort()还有一种特殊能力：如果你只需要前 n 个元素排序，它可以在完成任务后立刻停止。所以如果想对所有元素进行排序，你可以将序列的终点作为第二参数和最后一个参数传进去：

  ```
  /* sort all elements
  * - always n*log(n) complexity
  * - but usually twice as long as sort()
  */
  partial_sort (coll.begin(), coll.end(), coll.end());
  ```

- **stable_sort()** 内部采用 mergesort。它对所有元素进行排序：

  ```
  /* sort all elements
  * - n*log(n) or n*log(n)*log(n) complexity
  */
  stable_sort (coll.begin(), coll.end());
  ```

  然而只有在具备足够内存时，它才具有 n*log(n) 复杂度。否则其复杂度为 n * log(n) * log(n)。stable_sort()的优点是会保持相等元素之间的相对次序。

现在你对于哪种算法最合乎你的需求，应该有了一个大致印象。但是事情还没完。标准规格书中虽然规范了复杂度，却没有规范实作法。这是有好处的，任何实作版本可以在不违反标准的情况下采用新发明的、更好的排序法。例如 SGI 实作版本中的 sort()内部就是采用 introsort，这是一种新式算法，缺省状态下类似 quicksort，一旦情况转坏即将走入二次复杂度时，就转而采用 heapsort。标准规格中未能明定复杂度，也有一个缺点，那就是某些实作版本可能会采用合乎标准但表现很糟的算法。例如以 heapsort 来实现 sort()就符合标准。当然你也可以测试看看哪个算法最合乎需求，但这种量测可能不具移植性。

还有其它排序算法。例如 heap 算法中有个函数，可直接实作出一个 heap（这是个二叉树，应用于 heapsort 内部）。heap 算法是 priority queues（参见 10.3 节，p453）的实作基础。你可以像下面这样将群集内的所有元素排序：

```
/* sort all elements
* - n+n*log(n) complexity
*/
```

```
make_heap (coll.begin(), coll.end());
sort_heap (coll.begin(), coll.end());
```

关于 heap 和 heap 算法，细节详见 p406。

如果你只需要排序后的第 n 个元素，或只需要令最先或最后的 n 个元素（无序）就位，可以使用 nth_element()。可以利用 nth_element()将元素依照某排序准则分割成两个子集。也可利用 partition()或 stable_partition()达到相同效果。三者区别如下：

- 对于 **nth_element()**，传入第一子集的元素个数（当然也就确定了第二子集的元素个数）。例如：

```
// move the four lowest elements to the front
nth_element (coll.begin(),    // beginning of range
             coll.begin()+3,  // position between first and second
             coll.end());     // end of range
```

然而调用之后，你并不精确知道第一子集和第二子集之间有什么不同。事实上，两部分都可能包含和第 n 个元素相等的元素。

- 对于 **partition()**，你必须传入“将第一子集和第二子集区别开来”的精确排序准则：

```
// move all elements less than seven to the front
vector<int>::iterator pos;
pos = partition (coll1.begin(), coll1.end(),     // range
                 bind2nd(less<int>(),7));        // criterion
```

调用之后，你并不知道第一和第二子集内各有多少元素。返回的 pos 指出第二子集起点。第二子集的所有元素都不满足上述（被传入的）准则。

- **stable_partition()** 的行为类似 partition()，不过更具额外能力，保证两个子集内的元素的相对次序保持不变。

你永远可以把排序准则当做可有可无的参数，传给所有排序算法。缺省的排序准则是仿函数 less<>，所以元素按升序排列。

和变动性算法的情况一样，关联式容器不可作为排序算法的目标，因为关联式容器的元素被视为常数，不可改动。

Lists 没有提供随机存取迭代器，所以不可以对它使用排序算法。然而 lists 本身提供了一个成员函数 sort()，可用来对其元素排序。参见 p245。

已序区间算法（Sorted Range Algorithms）

所谓已序区间算法，意指其所作用的区间在某种排序准则下已序。表 9.7 列出 C++ 标准程序库中涵括的所有已序区间算法。和关联式容器一样，这些算法的优势

就是具有较佳复杂度。

表 9.7 已序区间算法（Sorted Range Algorithms）

名字	效果	页次
binary_search()	判断某区间内是否包含某个元素	410
includes()	判断某区间内的每一个元素是否都涵盖于另一区间中	411
lower_bound()	搜寻第一个“大于等于给定值”的元素	413
upper_bound()	搜寻第一个“大于给定值”的元素	413
equal_range()	返回“等于给定值”的所有元素构成的区间	415
merge()	将两个区间的元素合并	416
set_union()	求两个区间的并集	418
set_intersection()	求两个区间的交集	419
set_difference()	求位于第一区间但不位于第二区间的所有元素，形成一个已序（*sorted*）区间。	420
set_symmetric_difference()	找出只出现于两区间之一的所有元素，形成一个已序（*sorted*）区间。	421
inplace_merge()	将两个连续的已序（*sorted*）区间合并	423

表 9.7 中前 5 个算法属于非变动性算法，它们只是依命令搜寻元素。其它算法用来将两个已序（*sorted*）区间组合，然后把结果写到目标区间。一般而言，这些算法的结果仍然是已序的。

数值算法（Numeric Algorithms）

数值算法以不同方式组合数值元素。表 9.8 列出 C++ 标准程序库涵括的所有数值算法。很容易顾名思义。然而它们实际上比你所得的第一印象更加灵活强劲。例如，缺省状态下，accumulate()求取所有元素的总和。如果你把它作用在 strings 身上，就可以产生字符串连接功效。当你不采用 operator+ 而改用 operator*，得到的会是所有元素的乘积。另一个例子是，你应该了解，adjacent_difference() 和 partial_sum()可以将某区间在相对值和绝对值之间互相转换。

accumulate()和 inner_product()返回一个值，而且不变动区间。其它算法会把结果写到目标区间内，该区间与源区间的元素个数相同。

表 9.8 数值算法（Numeric Algorithms）

名字	功能	页次
accumulate()	组合所有元素（求总和、求乘积…）	425
inner_product()	组合两区间内的所有元素	427
adjacent_difference()	将每个元素和其前一元素组合	431
partial_sum()	将每个元素和其先前的所有元素组合	429

9.3 辅助函数

本章剩余部分将对所有 STL 算法逐一详细讨论。每个算法至少配备一个应用实例。为了简化这些例子，使你集中精力于真正重要的问题上，我用了一些辅助函数：

```
// algo/algostuff.hpp

#ifndef ALGOSTUFF_HPP
#define ALGOSTUFF_HPP
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <functional>
#include <numeric>

/* PRINT_ELEMENTS()
* - prints optional C-string optcstr followed by
* - all elements of the collection coll
* - separated by spaces
*/
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;
```

```
    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
    std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

/* INSERT_ELEMENTS (collection, first, last)
* - fill values from first to last into the collection
* - NOTE: NO half-open range
*/
template <class T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(),i);
    }
}

#endif /*ALGOSTUFF_HPP*/
```

algostuff.hpp 首先含入本章程序中可能用到的所有头文件，这样程序本身就不必多劳了。此外它定义了两个辅助函数：

1. PRINT_ELEMENTS() 此函数将第一参数所带入的“容器内的所有元素”打印出来，间以空格。第二参数可有可无，会成为前导词，打印于元素值之前（参见 p118）。
2. INSERT_ELEMENTS() 将元素安插于第一参数所带入的容器内。元素值来自第二参数和第三参数；两参数之间的所有元素值都将供安插之用。请注意这里并不是一个半开区间（half-open range）。

9.4 for_each() 算法

for_each() 算法非常灵活，它可以不同的方式存取、处理、修改每一个元素。

```
UnaryProc
for_each (InputIterator beg, InputIterator end, UnaryProc op)
```

- 对区间 [*beg*, *end*) 中的每一个元素调用：
  op(*elem*)
- 返回 *op*（已在算法内部被变动过）的一个副本。
- *op* 可以变动元素。不过请参考 p325 对 for_each() 和 transform() 之间的比较，后者具有相同能力，但行事方式有细微不同。
- *op* 的任何返回值都会被忽略。
- for_each() 算法的实作，参见 p126。
- 复杂度：线性。调用 op() 共 *numberOfElements* 次。

下面的例子将 print() 传给 for_each()，使得 for_each() 对每一个元素调用 print()，从而打印所有元素：

```
// algo/foreach1.cpp

#include "algostuff.hpp"
using namespace std;

// function called for each element
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // call print() for each element
    for_each (coll.begin(), coll.end(),   // range
              print);                     // operation
    cout << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6 7 8 9
```

如果想要调用元素的成员函数，可以使用 mem_fun 配接器。细节和范例参见 8.2.2 节, p307。

下面这个例子展示如何利用仿函数来改变每一个元素内容：

```
// algo/foreach2.cpp

#include "algostuff.hpp"
using namespace std;

// function object that adds the value with which it is initialized
template <class T>
class AddValue {
  private:
    T theValue; // value to add

  public:
    // constructor initializes the value to add
    AddValue (const T& v) : theValue(v) {
    }

    // the function call for the element adds the value
    void operator() (T& elem) const {
       elem += theValue;
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // add ten to each element
    for_each (coll.begin(), coll.end(),   // range
              AddValue<int>(10));         // operation
    PRINT_ELEMENTS(coll);

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),       // range
              AddValue<int>(*coll.begin()));  // operation
    PRINT_ELEMENTS(coll);
}
```

class AddValue<> 定义了一个仿函数，把构造函数所获得的值加到每一个元素身上。使用仿函数的好处就是，你可以在执行期间传入欲加的数值。程序输出如下：

```
11 12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29 30
```

关于这个例子的细节讨论，请参考 p128。同时请注意，你可以如此这般地运用 transform()算法完成同样任务（参见 p367）：

```
transform (coll.begin(), coll.end(),      // source range
           coll.begin(),                  // destination range
           bind2nd(plus<int>(),10));      // operation
...
transform (coll.begin(), coll.end(),      // source range
           coll.begin(),                  // destination range
           bind2nd(plus<int>(),*coll.begin())); // operation
```

关于 for_each()和 transform()的一般性比较，请见 p325。

第三个例子展示如何利用 for_each()的返回值。由于 for_each()能够返回一项操作，所以我们可以利用这一特性，在该项操作中处理返回结果：

```
// algo/foreach3.cpp

#include "algostuff.hpp"
using namespace std;

// function object to process the mean value
class MeanValue {
  private:
    long num; // number of elements
    long sum; // sum of all element values

  public:
    // constructor
    MeanValue () : num(0), sum(0) {
    }

    // function call
    // - process one more element of the sequence
    void operator() (int elem) {
        num++; // increment count
        sum += elem; // add value
    }
```

```
    // return mean value (implicit type conversion)
    operator double() {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,8);

    // process and print mean value
    double mv = for_each (coll.begin(), coll.end(), // range
                          MeanValue());             // operation
    cout << "mean value: " << mv << endl;
}
```

程序输出如下：

```
mean value: 4.5
```

p300 有一个例子，和此例非常相似，但有些微不同，请参考。

9.5 非变动性算法（Nonmodifying Algorithms）

本节讲述的算法不会变动元素值，也不会改变元素次序。

9.5.1 元素计数

```
difference_type
count (InputIterator beg, InputIterator end, const T& value)

difference_type
count_if (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- 第一形式会计算区间[*beg*, *end*)中元素值等于 *value* 的元素个数。
- 第二形式会计算区间[*beg*, *end*)中令以下一元判断式结果为 true 的元素个数：
  op(*elem*)
- 返回值型别 difference_type，是表现迭代器间距的型别：
  typename iterator_traits<InputIterator>::difference_type
  7.5 节, p283 曾经介绍过 iterator traits（迭代器特性）[4]
- 注意 *op* 在函数调用过程中不应改变自身状态。细节见 8.1.4 节, p302。
- *op* 不应该改动传进来的参数。
- 关联式容器（sets，multisets，maps 和 multimaps）提供了一个等效的成员函数 count()，用来计算等于某个 *value* 或某个 *key* 的元素个数（见 p234）。
- 复杂度：线性。执行比较动作（或调用 op()）共 *numberOfElements* 次。

以下范例根据不同的准则对元素进行计数：

```
// algo/count1.cpp

#include "algostuff.hpp"
using namespace std;

bool isEven (int elem)
{
    return elem % 2 == 0;
}
```

[4] 早期的 STL 中，count()和 count_if()有第四参数，可作为输入，亦能输出，当做计数器使用，返回值型别是 void。

```
int main()
{
    vector<int> coll;
    int num;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // count and print elements with value 4
    num = count (coll.begin(), coll.end(), // range
                 4); // value
    cout << "number of elements equal to 4: " << num << endl;

    // count elements with even value
    num = count_if (coll.begin(), coll.end(),      // range
                    isEven);                       // criterion
    cout << "number of elements with even value: " << num << endl;

    // count elements that are greater than value 4
    num = count_if (coll.begin(), coll.end(),      // range
                    bind2nd(greater<int>(),4));    // criterion
    cout << "number of elements greater than 4: " << num << endl;
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
number of elements equal to 4: 1
number of elements with even value: 4
number of elements greater than 4: 5
```

你也可以不必使用上述的 `isEven()`函数，改用这个表达式：

```
not1(bind2nd(modulus<int>(),2))
```

关于这个表达式的细节，请见 p306。

9.5.2 最小值和最大值

```
InputIterator
min_element (InputIterator beg, InputIterator end)

InputIterator
min_element (InputIterator beg, InputIterator end, CompFunc op)
```

```
InputIterator
max_element (InputIterator beg, InputIterator end)

InputIterator
max_element (InputIterator beg, InputIterator end, CompFunc op)
```

- 所有这些算法都返回区间 [*beg*, *end*) 中最小或最大元素的位置。
- 上述无 *op* 参数的版本（第一版本），以 operator< 进行元素的比较。
- *op* 用来比较两个元素：
  op(elem1,elem2)
  如果第一个元素小于第二个元素，应当返回 true。
- 如果存在多个最小值或最大值，上述算法返回找到的第一个最小或最大值。
- *op* 不应该改动传入的参数。
- 复杂度：线性。执行比较操作（或调用 *op*()）共 *numberOfElements–1* 次。

以下程序打印 coll 之中的最小元素和最大元素，并通过 absLess()打印最小元素和最大元素的绝对值：

```
// algo/minmax1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

bool absLess (int elem1, int elem2)
{
    return abs(elem1) < abs(elem2);
}

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,2,8);
    INSERT_ELEMENTS(coll,-3,5);

    PRINT_ELEMENTS(coll);

    // process and print minimum and maximum
    cout << "minimum: "
         << *min_element(coll.begin(),coll.end())
         << endl;

    cout << "maximum: "
         << *max_element(coll.begin(),coll.end())
         << endl;
```

```
    // process and print minimum and maximum of absolute values
    cout << "minimum of absolute values: "
         << *min_element(coll.begin(),coll.end(),
                         absLess)
         << endl;

    cout << "maximum of absolute values: "
         << *max_element(coll.begin(),coll.end(),
                         absLess)
         << endl;
}
```

程序输出如下：

```
2 3 4 5 6 7 8 -3 -2 -1 0 1 2 3 4 5
minimum: -3
maximum: 8
minimum of absolute values: 0
maximum of absolute values: 8
```

注意，算法分别返回最大和最小元素的位置，所以你必须使用一元运算 operator* 来打印其值。

9.5.3 搜寻元素

搜寻第一个匹配元素

```
InputIterator
find (InputIterator beg, InputIterator end, const T& value)

InputIterator
find_if (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- 第一形式返回区间 [*beg*, *end*) 中第一个“元素值等于 *value*”的元素位置。
- 第二形式返回区间 [*beg*, *end*) 中令以下一元判断式结果为 true 的第一个元素：
  op(*elem*)
- 如果没有找到匹配元素，两种形式都返回 *end*。

- 注意 *op* 在函数调用过程中不应改变自身状态。细节见 8.1.4 节, p302。
- *op* 不应该改动传递来的参数。
- 如果是已序区间，应使用 `lower_bound()`, `upper_bound()`, `equal_range()` 或 `binary_search()`算法以获取更高性能（参见 9.10 节, p409）。
- 关联式容器（sets, multisets, maps, multimaps）提供了一个等效的成员函数 `find()`，拥有对数复杂度，而非线性复杂度。
- 复杂度：线性。最多比较（或调用 *op*()）共 *numberOfElements* 次。

下面这个例子展示如何运用 `find()`来搜寻一个子区间：以元素值为 4 的第一个元素开始，以元素值为 4 的第二个元素结束：

```
// algo/find1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"coll: ");

    // find first element with value 4
    list<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),     // range
                 4);                           // value

    /* find second element with value 4
    * - note: continue the search behind the first 4 (if any)
    */
    list<int>::iterator pos2;
    if (pos1 != coll.end()) {
       pos2 = find (++pos1, coll.end(),   // range
                    4);                   // value
    }
```

```
    /* print all elements from first to second 4 (both included)
    * - note: now we need the position of the first 4 again (if any)
    * - note: we have to pass the position behind the second 4
    * (if any)
    */
    if (pos1!=coll.end() && pos2!=coll.end()) {
       copy (--pos1, ++pos2,
             ostream_iterator<int>(cout," "));
       cout << endl;
    }
}
```

为了搜寻第二个 4，你必须从第一个 4 的位置上前进。然而如果在群集的 end() 位置上再前进，会导致未定义的行为。所以如果你没有十足把握，最好在前进之前先检查 find() 的返回值。程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3 4
```

你可以在同一个区间中以不同的值先后两次调用 find()。然而，当你使用搜寻结果作为子区间的起点和终点时，务必十分小心，否则该子区间可能形成一个无效区间。此问题的讨论和范例请见 p97。

下面这个程序展示 find_if() 的用法，以极为特殊的搜寻准则来搜寻某个元素：

```
// algo/find2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    vector<int>::iterator pos;

    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"coll: ");

    // find first element greater than 3
    pos = find_if (coll.begin(), coll.end(),        // range
                   bind2nd(greater<int>(),3));      // criterion
```

```
    // print its position
    cout << "the "
         << distance(coll.begin(),pos) + 1
         << ". element is the first greater than 3" << endl;

    // find first element divisible by 3
    pos = find_if (coll.begin(), coll.end(),
                   not1(bind2nd(modulus<int>(),3)));

    // print its position
    cout << "the "
         << distance(coll.begin(),pos) + 1
         << ". element is the first divisible by 3" << endl;
}
```

第一次调用 find()时，使用了一个以 bind2nd 配接器组合而成的简单仿函数，搜寻第一个大于 3 的元素。第二次调用使用一个比较复杂的组合，搜寻第一个可被 3 整除的元素。

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
the 4. element is the first greater than 3
the 3. element is the first divisible by 3
```

p121 有一个例子，以 find()搜寻第一个质数。

搜寻前 n 个连续匹配值

```
InputIterator
search_n (InputIterator beg, InputIterator end,
          Size count, const T& value)

InputIterator
search_n (InputIterator beg, InputIterator end,
          Size count, const T& value, BinaryPredicate op)
```

- 第一形式返回区间[*beg*, *end*)中第一组“连续 *count* 个元素值全等于 *value*”的元素位置。
- 第二形式返回区间[*beg*, *end*)中第一组“连续 *count* 个元素造成以下一元判断式结果为 true”的元素位置：
  op(*elem*, *value*)
- 如果没有找到匹配元素，两种形式都返回 *end*。

- 注意，*op* 在函数调用过程中不应改变自身状态。细节见 8.1.4 节，p302。
- *op* 不应该变动被传进去的参数。
- 这两个算法并不在早期的 STL 规范中，也没有获得谨慎的对待，因此，第二形式使用了一个二元判断式，而非一元判断式，这破坏了早期 STL 的一致性。请参见 p346。
- 复杂度：线性。最多比较（或调用 *op*()）共 *numberOfElements* * *count* 次。

下面这个例子搜寻连续 4 个“数值大于等于 3”的元素：

```
// algo/searchn1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // find four consecutive elements with value 3
    deque<int>::iterator pos;
    pos = search_n (coll.begin(), coll.end(), // range
                    4,                        // count
                    3);                       // value

    // print result
    if (pos != coll.end()) {
    cout << "four consecutive elements with value 3 "
         << "start with " << distance(coll.begin(),pos) +1
         << ". element" << endl;
    }
    else {
       cout << "no four consecutive elements with value 3 found"
            << endl;
    }
```

```
    // find four consecutive elements with value greater than 3
    pos = search_n (coll.begin(), coll.end(), // range
                    4,                        // count
                    3,                        // value
                    greater<int>());          // criterion

    // print result
    if (pos != coll.end()) {
        cout << "four consecutive elements with value > 3 "
             << "start with " << distance(coll.begin(),pos) +1
             << ". element" << endl;
    }
    else {
        cout << "no four consecutive elements with value > 3 found"
             << endl;
    }
}
```

程序输出如下：

```
1 2 3 4 5 6 7 8 9
no four consecutive elements with value 3 found
four consecutive elements with value > 3 start with 4. element
```

关于 search_n()的第二形式，有一个险恶的问题。请看以下调用操作：

```
pos = search_n (coll.begin(), coll.end(),     // range
                4,                            // count
                3,                            // value
                greater<int>());              // criterion
```

以这种方法来搜寻符合某特殊准则的元素，其方法和 STL 其它组件可谓大相径庭。按照 STL 的一般观念，应该这么做：

```
pos = search_n_if (coll.begin(), coll.end(),       // range
                   4,                              // count
                   bind2nd(greater<int>(),3));     // criterion
```

可惜，当这个新算法被引入 C++ 标准时（它不属于早期的 STL），没有人注意到其中的不一致。当然，也许你觉得 4 个参数的形式更加便捷。不过它实际上只需要一个一元判断式的时候，却要求获得一个二元判断式，这恐怕非你所愿。例如，为了运用自己写的一个一元判断式，你通常会这么做：

```
bool isPrime (int elem);
...
pos = search_n_if (coll.begin(), coll.end(),  // range
                   4,                         // count
                   isPrime);                  // criterion
```

然而，根据实际定义，你却必须写一个二元判断式。所以你要不改变函数标记式（signature），要不就写一个简单的包装：

```
bool binaryIsPrime (int elem1, int) {
    return isPrime(elem1);
}
...
pos = search_n (coll.begin(), coll.end(),     // range
                4,                            // count
                0,                            // required dummy valu
                binaryIsPrime);               // binary criterion
```

搜寻第一个子区间

```
ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
        BinaryPredicate op)
```

- 两种形式都返回区间 [*beg*, *end*) 内“和区间 [*searchBeg*, *searchEnd*) 完全吻合”的第一个子区间内的第一个元素位置。
- 第一形式中，子区间的元素必须完全等于 [*searchBeg*, *searchEnd*) 的元素。
- 第二形式中，子区间的元素和 [*searchBeg*, *searchEnd*) 的对应元素必须造成以下二元判断式的结果为 true：
  op(elem, searchElem)
- 如果没有找到符合条件的子区间，两种形式都返回 *end*。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应变动传入的参数。
- 如果你在“只知道第一个元素和最后一个元素”的情况下要搜寻一个子区间，请参见 p97。

- 复杂度：线性。最多比较（或调用 *op*()）共 *numberOfElements* * *numberOfSearchElements* 次。

下面这个例子展示如何在另一个序列中搜寻一个子序列。请将这个例子和 p351 的 find_end()运用实例作一番比较。

```
// algo/search1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);

    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(subcoll,"subcoll: ");

    // search first occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),        // range
                  subcoll.begin(), subcoll.end()); // subrange

    // loop while subcoll found as subrange of coll
    while (pos != coll.end()) {
        // print position of first element
        cout << "subcoll found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;
        // search next occurrence of subcoll
        ++pos;
        pos = search (pos, coll.end(),                    // range
                      subcoll.begin(), subcoll.end());    // subrange
    }
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 3
subcoll found starting with element 10
```

下面这个例子展示如何运用 search()算法的第二形式，以更复杂的准则来搜寻某个子序列。这里搜寻的是“偶数、奇数、偶数”排列而成的子序列：

```
// algo/search2.cpp

#include "algostuff.hpp"
using namespace std;

// checks whether an element is even or odd
bool checkEven (int elem, bool even)
{
    if (even) {
        return elem % 2 == 0;
    }
    else {
        return elem % 2 == 1;
    }
}

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    /* arguments for checkEven()
    * - check for: "even odd even"
    */
    bool checkEvenArgs[3] = { true, false, true };

    // search first subrange in coll
    vector<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),        // range
                  checkEvenArgs, checkEvenArgs+3,  // subrange values
                  checkEven);                      // subrange criterion
```

```
    // loop while subrange found
    while (pos != coll.end()) {
        // print position of first element
        cout << "subrange found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // search next subrange in coll
        pos = search (++pos, coll.end(),              // range
                      checkEvenArgs, checkEvenArgs+3, // subr. values
                      checkEven);                     // subr. criterion
    }
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
subrange found starting with element 2
subrange found starting with element 4
subrange found starting with element 6
```

（译注：以上表示共找出三个满足“偶、奇、偶”条件的子序列，分别始自元素 2,4,6）

搜寻最后一个子区间

```
ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
          ForwardIterator searchBeg, ForwardIterator searchEnd)

ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
          ForwardIterator searchBeg, ForwardIterator searchEnd,
          BinaryPredicate op)
```

- 两种形式都返回区间[*beg*, *end*)之中“和区间[*searchBeg*, *searchEnd*)完全吻合”的最后一个子区间内的第一个元素位置。
- 第一形式中，子区间的元素必须完全等于[*searchBeg*, *searchEnd*)的元素。
- 第二形式中，子区间的元素和[*searchBeg*, *searchEnd*)的对应元素必须造成以下二元判断式的结果为 true：

  op(elem, searchElem)

- 如果没有找到符合条件的子区间，两种形式都返回 *end*。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 如果你在“只知道第一个元素和最后一个元素”的情况下要搜寻一个子区间，请参考 p97。
- 这些算法并不是早期 STL 的一部分。很不幸它们被命名为 `find_end()`而不是 `search_end()`，如果是后者，比较具有一致性，因为用来搜寻第一个子区间的算法名为 `search()`。
- 复杂度：线性。最多比较（或调用 `op()`）共 *numberOfElements* * *numberOfSearchElements* 次。

下面这个例子展示如何在一个序列中搜寻“与某序列相等”的最后一个子序列（请和 p348 的 `search()`例子作比较）：

```
// algo/findend1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);

    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(subcoll,"subcoll: ");

    // search last occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = find_end (coll.begin(), coll.end(), // range
                    subcoll.begin(), subcoll.end()); // subrange

    // loop while subcoll found as subrange of coll
    deque<int>::iterator end(coll.end());
```

```
    while (pos != end) {
        // print position of first element
        cout << "subcoll found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // search next occurrence of subcoll
        end = pos;
        pos = find_end (coll.begin(), end,                    // range
                        subcoll.begin(), subcoll.end()); // subrange
    }
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 10
subcoll found starting with element 3
```

这个算法的第二形式，可参考 p349 search()的例子。你可以采用类似的手法来使用 find_end()。

搜寻某些元素的第一次出现地点

```
ForwardIterator
find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
               ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator
find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
               ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
               BinaryPredicate op)
```

- 第一形式返回第一个“既在区间 [*beg*, *end*) 中出现，也在区间 [*searchBeg*, *searchEnd*) 中出现”的元素的位置。
- 第二形式返回区间 [*beg*, *end*) 中第一个这样的元素：它和区间 [*searchBeg*, *searchEnd*) 内每一个元素进行以下动作的结果都是 true：
  op(elem, searchElem)
- 如果没有找到吻合元素，两种形式都返回 *end*。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节，p302。

- op 不应改动传入的参数。
- 你可以使用逆向迭代器来搜寻最后一个这样的元素。
- 这几个算法并不在早期的 STL 规范中。
- 复杂度：线性。最多比较（或调用 op()）共 *numberOfElements* * *numberOfSearchElements* 次。

下面这个例子展示 find_first_of()的用法：

```
// algo/findof1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    list<int> searchcoll;

    INSERT_ELEMENTS(coll,1,11);
    INSERT_ELEMENTS(searchcoll,3,5);

    PRINT_ELEMENTS(coll, "coll: ");
    PRINT_ELEMENTS(searchcoll,"searchcoll: ");

    // search first occurrence of an element of searchcoll in coll
    vector<int>::iterator pos;
    pos = find_first_of (coll.begin(), coll.end(), // range
                         searchcoll.begin(), // beginning of search set
                         searchcoll.end());  // end of search set

    cout << "first element of searchcoll in coll is element "
         << distance(coll.begin(),pos) + 1
         << endl;

    // search last occurrence of an element of searchcoll in coll
    vector<int>::reverse_iterator rpos;
    rpos = find_first_of (coll.rbegin(), coll.rend(), // range
                          searchcoll.begin(), // beginning of search set
                          searchcoll.end());  // end of search set
    cout << "last element of searchcoll in coll is element "
         << distance(coll.begin(),rpos.base())
         << endl;
}
```

第二次调用系采用逆向迭代器，搜寻最后一个“与 searchcoll 内某一元素相等”的元素。为了打印这个元素的位置，此处调用 base()将逆向迭代器转化成一般(正向)迭代器。这样你就可以得到从起点算起的距离。通常你应该将 distance()的结果加 1，因为第一个元素的距离是 0。然而因为 base()背地里改变了迭代器的所指数值位置，所以你得到相同的效果。关于 base()，请见 7.4.1 节, p269。

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9 10 11
searchcoll: 3 4 5
first element of searchcoll in coll is element 3
last element of searchcoll in coll is element 5
```

搜寻两个连续且相等的元素

```
InputIterator
adjacent_find (InputIterator beg, InputIterator end)

InputIterator
adjacent_find (InputIterator beg, InputIterator end,
               BinaryPredicate op)
```

- 第一形式返回区间 [*beg*, *end*) 中第一对“连续两个相等元素”之中的第一元素位置。
- 第二形式返回区间 [*beg*, *end*) 中第一对“连续两个元素均使以下二元判断式的结果为 true”的其中第一元素位置：
  op(elem,nextElem)
- 如果没有找到吻合元素，两者都返回 *end*。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 复杂度：线性。最多比较（或调用 *op*()）共 ***numberOfElements*** 次。

下面这个程序展示 adjacent_find()两种形式的用法：

```
// algo/adjfind1.cpp

#include "algostuff.hpp"
using namespace std;

// return whether the second object has double the value of the first
bool doubled (int elem1, int elem2)
{
```

```
    return elem1 * 2 == elem2;
}

int main()
{
    vector<int> coll;

    coll.push_back(1);
    coll.push_back(3);
    coll.push_back(2);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(5);
    coll.push_back(0);

    PRINT_ELEMENTS(coll,"coll: ");

    // search first two elements with equal value
    vector<int>::iterator pos;
    pos = adjacent_find (coll.begin(), coll.end());

    if (pos != coll.end()) {
        cout << "first two elements with equal value have position "
             << distance(coll.begin(),pos) + 1
             << endl;
    }

    // search first two elements for which the second has
    // double the value of the first
    pos = adjacent_find (coll.begin(), coll.end(), // range
                         doubled);                 // criterion

    if (pos != coll.end()) {
        cout << "first two elements with second value twice the "
             << "first have pos. "
             << distance(coll.begin(),pos) + 1
             << endl;
    }
}
```

第一次调用 adjacent_find()是为了搜寻相等值。第二次调用以 doubled()来搜寻“连续两元素，后一个元素是前一个元素的两倍”，找到后返回其中第一元素的位置。程序输出如下：

```
coll: 1 3 2 4 5 5 0
first two elements with equal value have position 5
first two elements with second value twice the first have pos. 3
```

9.5.4 区间的比较

检验相等性

```
bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg)

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg, BinaryPredicate op)
```

- 第一形式判断区间[*beg*, *end*)内的元素是否都和“以 *cmpBeg* 开头的区间”内的元素相等。
- 第二形式判断区间[*beg*, *end*)内的元素和“以 *cmpBeg* 开头的区间”内的对应元素”是否都能够使以下二元判断式得到 true：
  op(*elem*, *cmpElem*)
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 调用者必须确保“以 *cmpBeg* 开头的区间”内含足够元素。
- 当序列不相等时，如果想要了解其间的不同，应使用 mismatch()算法（参见 p358）。
- 复杂度：线性。最多比较（或调用 *op*()）共 *numberOfElements* 次。

下面这个例子展示 equal() 两种形式的用法。第一次调用用来检查元素是否相等。第二次调用使用一个辅助判断式，检查两个群集中的元素是否具备一一对应的奇偶关系：

```
// algo/equal1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
bool bothEvenOrOdd (int elem1, int elem2)
{
    return elem1 % 2 == elem2 % 2;
}

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,7);
    INSERT_ELEMENTS(coll2,3,9);

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // check whether both collections are equal
    if (equal (coll1.begin(), coll1.end(),     // first range
               coll2.begin())) {               // second range
        cout << "coll1 == coll2" << endl;
    }
    else {
        cout << "coll1 != coll2" << endl;
    }

    // check for corresponding even and odd elements
    if (equal (coll1.begin(), coll1.end(), // first range
               coll2.begin(),              // second range
               bothEvenOrOdd)) {           // comparison criterion
        cout << "even and odd elements correspond" << endl;
    }
    else {
       cout << "even and odd elements do not correspond" << endl;
    }
}
```

程序输出如下：

```
coll1: 1 2 3 4 5 6 7
coll2: 3 4 5 6 7 8 9
coll1 != coll2
even and odd elements correspond
```

搜寻第一处不同点

```
pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg)

pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg,
          BinaryPredicate op)
```

- 第一形式返回区间 [*beg*, *end*) 和“以 *cmpBeg* 开头的区间”之中第一组两两相异的对应元素。
- 第二形式返回区间 [*beg*, *end*) 和“以 *cmpBeg* 开头的区间”之中第一组“使以下二元判断式获得 false”的对应元素：
  op(*elem*, *cmpElem*)
- 如果没有找到相异点，就返回一个 pair，以 *end* 和第二序列的对应元素组成。这并不意味两个序列相等，因为第二序列有可能包含比较多的元素。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 调用者必须确保“以 *cmpBeg* 开头的区间”内含足够元素。
- 如果想知道两个序列是否相等，应当使用 equal() 算法（p356）。
- 复杂度：线性。最多比较（或调用 *op*()）共 *numberOfElements* 次。

下面这个例子展示 mismatch() 两种形式的用法：

```
// algo/misma1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,6);

    for (int i=1; i<=16; i*=2) {
       coll2.push_back(i);
    }
    coll2.push_back(3);
```

```
    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // find first mismatch
    pair<vector<int>::iterator,list<int>::iterator> values;
    values = mismatch (coll1.begin(), coll1.end(), // first range
                       coll2.begin()); // second range
    if (values.first == coll1.end()) {
        cout << "no mismatch" << endl;
    }
    else {
        cout << "first mismatch: "
             << *values.first << " and "
             << *values.second << endl;
    }

    /* find first position where the element of coll1 is not
    * less than the corresponding element of coll2
    */
    values = mismatch (coll1.begin(), coll1.end(), // first range
                       coll2.begin(), // second range
                       less_equal<int>()); // criterion
    if (values.first == coll1.end()) {
        cout << "always less-or-equal" << endl;
    }
    else {
        cout << "not less-or-equal: "
             << *values.first << " and "
             << *values.second << endl;
    }
}
```

第一次调用 mismatch()用以搜寻第一对互异的对应元素。如果找到了，就把它们的值写到标准输出装置。第二次调用用来搜寻符合以下条件的第一对元素："第一序列的元素比第二序列的对应元素大"，找到后返回该两个元素。程序输出如下：

```
coll1: 1 2 3 4 5 6
coll2: 1 2 4 8 16 3
first mismatch: 3 and 4
not less-or-equal: 6 and 3
```

检验“小于”

```
bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2)

bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2,
                         CompFunc op)
```

- 两种形式都用来判断区间[*beg1*, *end1*)的元素是否小于区间[*beg2*, *end2*)的元素。所谓“小于”是指本着“字典（lexicographical）次序”的意义。
- 第一形式以 operator< 来比较元素。
- 第二形式以二元判断式

  op(elem1,elem2)

  比较元素。如果 *elem1* 小于 *elem2*，则判断式应当返回 true。
- “字典次序”意味两个序列中的元素一一比较，直到以下情况发生：
  - 如果两元素不相等，则这两个元素的比较结果就是整个两序列的比较结果。
  - 如果两序列中的元素数量不同，则元素较少的那个序列小于另一序列。所以，如果第一序列的元素数量较少，比较结果是 true。
  - 如果两序列都没有更多的元素可作比较，则这两个序列相等，整个比较结果是 false。
- 注意，*op* 在函数调用过程中不应改变自身状态。详见 8.1.4 节, p302。
- *op* 不应改动传入的参数。
- 复杂度：线性。最多比较（或调用 *op*()）共

  *2 * min(numberOfElements1, numberOfElements2)* 次。

下面这个例子展示如何利用这个算法对群集完成“字典次序”的排序：

```
// algo/lexico1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
void printCollection (const list<int>& l)
{
    PRINT_ELEMENTS(l);
}

bool lessForCollection (const list<int>& l1, const list<int>& l2)
{
    return lexicographical_compare
              (l1.begin(), l1.end(), // first range
              l2.begin(), l2.end()); // second range
}

int main()
{
    list<int> c1, c2, c3, c4;

    // fill all collections with the same starting values
    INSERT_ELEMENTS(c1,1,5);
    c4 = c3 = c2 = c1;

    // and now some differences
    c1.push_back(7);
    c3.push_back(2);
    c3.push_back(0);
    c4.push_back(2);

    // create collection of collections
    vector<list<int> > cc;

    cc.push_back(c1);
    cc.push_back(c2);
    cc.push_back(c3);
    cc.push_back(c4);
    cc.push_back(c3);
    cc.push_back(c1);
    cc.push_back(c4);
    cc.push_back(c2);
```

```
    // print all collections
    for_each (cc.begin(), cc.end(),
              printCollection);
    cout << endl;

    // sort collection lexicographically
    sort (cc.begin(), cc.end(), // range
          lessForCollection); // sorting criterion

    // print all collections again
    for_each (cc.begin(), cc.end(),
              printCollection);
}
```

其中 vector cc 是由好几个群集（都是 lists）初始化而来。调用 sort()时使用了二元判断式 lessForCollection()来比较两群集（关于 sort()，详见 p397）。在 lessForCollection()中，算法 lexicographical_compare()用来对群集进行字典序比较。程序输出如下：

```
1 2 3 4 5 7
1 2 3 4 5
1 2 3 4 5 2 0
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 2
1 2 3 4 5

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 2
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 7
```

9.6 变动性算法（Modifying Algorithms）

本节描述的算法会变动区间内的元素内容。有两种方法可以变动元素内容：

1. 运用迭代器遍历序列的过程中，直接加以变动。
2. 将元素从源（source）区间复制到目标（destination）区间的过程中加以变动。

某些算法同时提供以上两种方法，那么使用第二法的版本会有尾词 _copy。

注意，目标区间不可以是关联式容器，因为关联式容器的元素被视为常数。如果没有这个限制，其自动排序特性将无法得到保证。

所有具有单一目标区间的算法，都返回区间内“最后一个被复制元素”的下一位置。

9.6.1 复制（Copying）元素

```
OutputIterator
copy (InputIterator sourceBeg,
      InputIterator sourceEnd,
      OutputIterator destBeg)

BidirectionalIterator1
copy_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- 这两个算法都将源区间 [sourceBeg, sourceEnd) 中的所有元素复制到以 destBeg 为起点或以 destEnd 为终点的目标区间去。
- 返回目标区间内最后一个被复制元素的下一位置，也就是第一个未被覆盖（overwritten）的元素的位置。
- destBeg 或 destEnd 不可处于 [sourceBeg, sourceEnd) 区间内。
- copy() 正向遍历序列，而 copy_backward() 逆向遍历序列。只有在源区间和目标区间存在重复区域时，这个不同点才会导致一些问题：

  ——如果要把一个区间复制到前端，应使用 copy()。所以 *destBeg* 的位置应该在 *sourceBeg* 之前。

  ——如果要把一个区间复制到后端，应使用 copy_backward()。所以 *destEnd* 的位置应该在 *sourceEnd* 之后。

  所以，只要第三参数位于由前两个参数所确定下来的源区间中，你就应该使用另一形式。注意，如果转而使用另一形式，就意味你原本应传递目标区间的起点，现在要转而传递终点了。关于两者的区别，请参考 p365 的例子。
- STL 并没有所谓 copy_if() 算法，所以如果要复制符合某特定准则的元素，请使用 remove_copy_if()，参见 p380。

- 如果希望在复制过程中逆转元素次序，应使用 reverse_copy()（参见 p386）。该算法比“copy()算法搭配逆向迭代器”略快。
- 调用者必须确保目标区间有足够空间，要不就得使用 Insert 迭代器。
- 关于 copy()算法的实作，请看 p271。
- 如果想把容器内的所有元素赋值（*assign*）给另一个容器，应当使用 assignment 运操作符（当两个容器的型别相同时才能这么做，参见 p236）或使用容器的 assign()成员函数（当两个容器的型别不同时就采用此法，参见 p237）。
- 如果希望在复制过程中删除某些元素，应使用算法 remove_copy()和 remove_copy_if()（参见 p380）。
- 如果希望在复制过程中改变元素，请使用算法 transform()（参见 p367）或 replace_copy（参见 p367）。
- 复杂度：线性，执行 *numberOfElements* 次赋值（*assign*）动作。

下面这个例子展示 copy()的一些简单用法：

```
// algo/copy1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);

    /* copy elements of coll1 into coll2
    * - use back inserter to insert instead of overwrite
    */
    copy (coll1.begin(), coll1.end(),     // source range
          back_inserter(coll2));          // destination range

    /* print elements of coll2
    * - copy elements to cout using an ostream iterator
    */
    copy (coll2.begin(), coll2.end(),          // source range
          ostream_iterator<int>(cout," "));    // destination range
    cout << endl;
```

```
    /* copy elements of coll1 into coll2 in reverse order
    * - now overwriting
    */
    copy (coll1.rbegin(), coll1.rend(),   // source range
          coll2.begin());                 // destination range

    // print elements of coll2 again
    copy (coll2.begin(), coll2.end(),           // source range
          ostream_iterator<int>(cout," "));     // destination range
    cout << endl;
}
```

在这个例子里，back inserters（参见 7.4.2 节，p272）用来在目标区间中安插元素。如果不使用这类安插型迭代器，copy()算法就会在空的 coll2 容器上实施覆盖（overwritten）操作，导致未定义的行为。同样道理，我们可使用 ostream iterator 把标准输出装置当成目标，参见 7.4.3 节，p278 实例。

程序输出如下：

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
```

下面这个例子展示 copy()和 copy_backward()之间的区别：

```
// algo/copy2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    /* initialize source collection with ".....abcdef....."
    */
    vector<char> source(10,'.');
    for (int c='a'; c<='f'; c++) {
         source.push_back(c);
    }
    source.insert(source.end(),10,'.');
    PRINT_ELEMENTS(source,"source: ");

    // copy all letters three elements in front of the 'a'
    vector<char> c1(source.begin(),source.end());
    copy (c1.begin()+10, c1.begin()+16,   // source range
          c1.begin()+7);                  // destination range
    PRINT_ELEMENTS(c1,"c1: ");
```

```
    // copy all letters three elements behind the 'f'
    vector<char> c2(source.begin(),source.end());
    copy_backward (c2.begin()+10, c2.begin()+16, // source range
                   c2.begin()+19); // destination range
    PRINT_ELEMENTS(c2,"c2: ");
}
```

注意，无论是调用copy()或是copy_backward()，第三参数都不处于源区间中。程序输出如下：

```
source: . . . . . a b c d e f . . . . . .
c1:     . . . . . a b c d e f d e f . . . . .
c2:     . . . . . a b c a b c d e f . . . . .
```

第三个例子示范如何使用copy()作为标准输入装置和标准输出装置之间的数据过滤器：程序读取strings，并以一行一个的方式打印它们：

```
// algo/copy3.cpp

#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main()
{
   copy (istream_iterator<string>(cin),  // beginning of source
         istream_iterator<string>(),     // end of source
         ostream_iterator<string>(cout,"\n")); // destination
}
```

9.6.2 转换（Transforming）和结合（Combining）元素

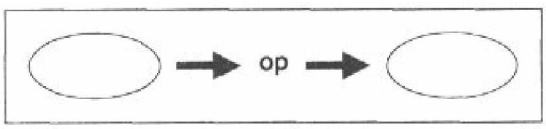
算法transform()提供以下两种能力：

1. 第一形式有4个参数。把源区间的元素转换到目标区间。也就是说，复制和修改元素一气呵成。
2. 第二形式有5个参数，将前两个源序列中的元素合并，并将结果写入目标区间。

转换元素

```
OutputIterator
transform (InputIterator sourceBeg, InputIterator sourceEnd,
           OutputIterator destBeg, UnaryFunc op)
```

- 针对源区间[*sourceBeg*, *sourceEnd*)中的每一个元素调用：
  op(*elem*)
  并将结果写到以 *destBeg* 起始的目标区间内。



- 返回目标区间内“最后一个被转换元素”的下一位置，也就是第一个未被覆盖（overwritten）的元素的位置。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- *sourceBeg* 与 *destBeg* 可以相同，所以，和 for_each()算法一样，你可以使用这个算法来变动某一序列内的元素。请看 p325 对两者的比较。
- 如果想以某值替换符合某一准则的元素，应使用 replace()算法（见 p375）。
- 复杂度：线性，对 *op*()执行 *numberOfElements* 次调用。

下面这个例子展示以上所说的 transform()用法：

```
// algo/transf1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");

    // negate all elements in coll1
    transform (coll1.begin(), coll1.end(), // source range
               coll1.begin(),              // destination range
               negate<int>());             // operation
    PRINT_ELEMENTS(coll1,"negated: ");
```

```
    // transform elements of coll1 into coll2 with
    // ten times their value
    transform (coll1.begin(), coll1.end(),     // source range
               back_inserter(coll2),           // destination range
               bind2nd(multiplies<int>(),10)); // operation
    PRINT_ELEMENTS(coll2,"coll2: ");

    // print coll2 negatively and in reverse order
    transform (coll2.rbegin(), coll2.rend(),   // source range
              ostream_iterator<int>(cout," "), // destination range
              negate<int>());                  // operation
    cout << endl;
}
```

程序输出如下：

```
coll1: 1 2 3 4 5 6 7 8 9
negated: -1 -2 -3 -4 -5 -6 -7 -8 -9
coll2: -10 -20 -30 -40 -50 -60 -70 -80 -90
90 80 70 60 50 40 30 20 10
```

关于如何“将两种不同操作组合起来，对元素进行处理”，请见 p315 实例。

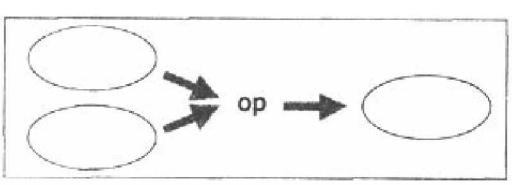
将两序列的元素加以结合（Combining）

```
OutputIterator
transform (InputIterator1 sourc1Beg, InputIterator1 sourc1End,
           InputIterator2 source2Beg,
           OutputIterator destBeg,
           BinaryFunc op)
```

- 针对第一个源区间（*sourc1Beg*, *sourc1End*）以及“从 *source2Beg* 开始的第二个源区间”的对应元素，调用：

  op(*source1Elem*, *source2Elem*)

  并将结果写入以 *destBeg* 起始的目标区间内。



- 返回目标区间内的“最后一个被转换元素”的下一位置，就是第一个未被覆盖（overwritten）的元素的位置。
- 调用者必须保证第二源区间有足够空间（至少拥有和第一源区间相同的空间大小）。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- *source1Beg*, *source2Beg*, *destBeg* 可以相同。所以，你可以让元素自己和自己结合，然后将结果覆盖（overwritten）至某个序列。
- 复杂度：线性，对 `op()` 执行 *numberOfElements* 次调用。

下面这个例子展示以上所说的 `transform()` 用法：

```
// algo/transf2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");

    // square each element
    transform (coll1.begin(), coll1.end(), // first source range
               coll1.begin(),              // second source range
               coll1.begin(),              // destination range
               multiplies<int>());         // operation
    PRINT_ELEMENTS(coll1,"squared: ");

    /* add each element traversed forward with each element traversed
    backward
    * and insert result into coll2
    */
    transform (coll1.begin(), coll1.end(), // first source range
               coll1.rbegin(),             // second source range
               back_inserter(coll2),       // destination range
               plus<int>());               // operation
    PRINT_ELEMENTS(coll2,"coll2: ");
```

```
    // print differences of two corresponding elements
    cout << "diff: ";
    transform (coll1.begin(), coll1.end(), // first source range
               coll2.begin(),              // second source range
               ostream_iterator<int>(cout, " "), // destination range
               minus<int>());              // operation
    cout << endl;
}
```

程序输出如下：

```
coll1: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
coll2: 82 68 58 52 50 52 58 68 82
diff: -81 -64 -49 -36 -25 -16 -9 -4 -1
```

9.6.3 互换（Swapping）元素内容

```
ForwardIterator2
swap_ranges (ForwardIterator1 beg1, ForwardIterator1 end1,
             ForwardIterator2 beg2)
```

- 将区间[*beg1*,*end1*)内的元素和“从 *beg2* 开始的区间”内的对应元素互换。
- 返回第二区间中“最后一个被交换元素”的下一位置。
- 调用者必须确保目标区间有足够空间。
- 两区间不得重迭。
- 如果要将相同型别的两个容器内的全部元素互换，应使用 swap()成员函数，因为该成员函数通常具有常数复杂度（参见 p237）。
- 复杂度：线性，执行 *numberOfElements* 次交换操作。

下面这个例子展示 swap_ranges()的用法：

```
// algo/swap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    deque<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);
```

```
    INSERT_ELEMENTS(coll2,11,23);

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // swap elements of coll1 with corresponding elements of coll2
    deque<int>::iterator pos;
    pos = swap_ranges (coll1.begin(), coll1.end(), // first range
                       coll2.begin());             // second range

    PRINT_ELEMENTS(coll1,"\ncoll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");
    if (pos != coll2.end()) {
        cout << "first element not modified: "
             << *pos << endl;
    }

    // mirror first three with last three elements in coll2
    swap_ranges (coll2.begin(), coll2.begin()+3,   // first range
                 coll2.rbegin());                  // second range

    PRINT_ELEMENTS(coll2,"\ncoll2: ");
}
```

第一次调用 swap_ranges()是为了将 coll1 的元素和 coll2 的对应元素交换。coll2之内的剩余元素不予变动。swap_ranges()算法返回第一个未被改动的元素。第二次调用 swap_ranges()是为了将 coll2 内的前 3 个元素和后 3 个元素交换。由于运用了逆向迭代器，所以元素以镜像方式交换（从外向内交换）。程序输出如下：

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 11 12 13 14 15 16 17 18 19 20 21 22 23

coll1: 11 12 13 14 15 16 17 18 19
coll2: 1 2 3 4 5 6 7 8 9 20 21 22 23
first element not modified: 20

coll2: 23 22 21 4 5 6 7 8 9 20 3 2 1
```

9.6.4 赋予（Assigning）新值

赋予完全相同的数值

```
Void
fill (ForwardIterator beg, ForwardIterator end,
      const T& newValue)

Void
fill_n (OutputIterator beg, Size num,
        const T& newValue)
```

- fill()将区间[beg, end)内的每一个元素都赋予新值newValue。
- fill_n()将“从beg开始的前num个元素”赋予新值newValue。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- 复杂度：线性，分别进行*numberOfElements*次或*num*次赋值（*assign*）操作。

以下程序展示fill()和fill_n()的用法：

```
// algo/filll.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    // print ten times 7.7
    fill_n(ostream_iterator<float>(cout, " "), // beginning of destination
           10,              // count
           7.7);            // new value
    cout << endl;

    list<string> coll;

    // insert "hello" nine times
    fill_n(back_inserter(coll), // beginning of destination
           9,                   // count
           "hello");            // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // overwrite all elements with "again"
    fill(coll.begin(), coll.end(),   // destination
         "again");                   // new value
    PRINT_ELEMENTS(coll,"coll: ");
```

```
    // replace all but two elements with "hi"
    fill_n(coll.begin(),          // beginning of destination
           coll.size()-2,         // count
           "hi");                 // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // replace the second and up to the last element but one with "hmmm"
    list<string>::iterator pos1, pos2;
    pos1 = coll.begin();
    pos2 = coll.end();
    fill (++pos1, --pos2,         // destination
          "hmmm");                // new value
    PRINT_ELEMENTS(coll,"coll: ");
}
```

第一次调用动作示范如何使用 fill_n()打印特定数量的值。其它针对 fill()和 fill_n()的调用动作则示范如何在一个 strings list 中安插和替换元素。程序输出如下：

```
7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7
coll: hello hello hello hello hello hello hello hello hello
coll: again again again again again again again again again
coll: hi hi hi hi hi hi hi again again
coll: hi hmmm hmmm hmmm hmmm hmmm hmmm hmmm again
```

赋予新产生的数值

```
void
generate (ForwardIterator beg, ForwardIterator end,
          Func op)
void
generate_n (OutputIterator beg, Size num,
            Func op)
```

- generate()会调用以下动作：

  op()

  产生新值，并赋值给区间[*beg*, *end*)内的每个元素。

- generate_n()会调用以下动作：
  op()
  产生新值，并赋值给“以 *beg* 起始的区间”内的前 *num* 个元素。
- 调用者必须确保目标区间有足够空间，要不就得使用插入型迭代器。
- 复杂度：线性，分别进行 ***numberOfElements*** 次或 *num* 次赋值（***assign***）操作。

以下程序展示如何利用 generate()和 generate_n()安插和赋值一些随机数：

```
// algo/generate.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert five random numbers
    generate_n (back_inserter(coll), // beginning of destination range
                5,                   // count
                rand); // new value generator
    PRINT_ELEMENTS(coll);

    // overwrite with five new random numbers
    generate (coll.begin(), coll.end(),   // destination range
              rand);                      // new value generator
    PRINT_ELEMENTS(coll);
}
```

其中所用的 rand()函数将于 12.3 节，p581 介绍。程序可能输出如下：

```
41 18467 6334 26500 19169
15724 11478 29358 26962 24464
```

实际输出结果视平台而定——因为 rand()产生的随机数序列并无一定标准。

8.1.2 节，p296 有一个例子展示如何将 generate()和一个仿函数（functor，另名 function object）搭配使用，产生一个数列。

9.6.5 替换（Replacing）元素

替换序列内的元素

```
void
replace (ForwardIterator beg, ForwardIterator end,
         const T& oldValue, const T& newValue)
void
replace_if (ForwardIterator beg, ForwardIterator end,
            UnaryPredicate op, const T& newValue)
```

- replace()将区间[*beg*,*end*)之内每一个“与 *oldValue* 相等”的元素替换为 *newValue*。
- replace_if()将区间[*beg*,*end*)之内每一个令以下一元判断式：
  op(*elem*)
  获得 true 的元素替换为 *newValue*。
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- 复杂度：线性，执行比较操作（或调用 *op*()）*numberOfElements* 次。

以下程序示范 replace()和 replace_if()的用法：

```
// algo/replace1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,2,7);
    INSERT_ELEMENTS(coll,4,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // replace all elements with value 6 with 42
    replace (coll.begin(), coll.end(),    // range
             6,                           // old value
             42);                         // new value
    PRINT_ELEMENTS(coll,"coll: ");
```

```
    // replace all elements with value less than 5 with 0
    replace_if (coll.begin(), coll.end(),      // range
                bind2nd(less<int>(),5),        // criterion for replacement
                0);                            // new value
    PRINT_ELEMENTS(coll,"coll: ");
}
```

程序输出如下：

```
coll: 2 3 4 5 6 7 4 5 6 7 8 9
coll: 2 3 4 5 42 7 4 5 42 7 8 9
coll: 0 0 0 5 42 7 0 5 42 7 8 9
```

复制并替换元素

```
OutputIterator
replace_copy (InputIterator sourceBeg, InputIterator sourceEnd,
              OutputIterator destBeg,
              const T& oldValue, const T& newValue)

OutputIterator
replace_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                 OutputIterator destBeg,
                 UnaryPredicate op, const T& newValue)
```

- replace_copy()是 copy()和 replace()的组合。它将源区间[*beg*,*end*)中的元素复制到“以 *destBeg* 为起点”的目标区间去，同时将其中“与 *oldValue* 相等”的所有元素替换为 *newValue*。
- replace_copy_if()是 copy()和 replace_if()的组合。[*beg*,*end*)中的元素被复制到“以 *destBeg* 为起点”的目标区间，同时将其中“令以下一元判断式：*op*(*elem*)
  结果为 true”的所有元素替换为 *newValue*。
- 两个算法都返回目标区间中“最后一个被复制元素”的下一位置，也就是第一个未被覆盖（overwritten）的元素位置。
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- 复杂度：线性，执行比较动作（或调用 *op*()）*numberOfElements* 次。

以下程序示范如何使用 replace_copy()和 replace_copy_if()：

```
// algo/replace2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    PRINT_ELEMENTS(coll);

    // print all elements with value 5 replaced with 55
    replace_copy(coll.begin(), coll.end(),          // source
                 ostream_iterator<int>(cout," "),  // destination
                 5,                                 // old value
                 55);                               // new value
    cout << endl;

    // print all elements with a value less than 5 replaced with 42
    replace_copy_if(coll.begin(), coll.end(),       // source
                    ostream_iterator<int>(cout," "), // destination
                    bind2nd(less<int>(),5), // replacement criterion
                    42);                            // new value
    cout << endl;

    // print each element while each odd element is replaced with 0
    replace_copy_if(coll.begin(), coll.end(),       // source
                    ostream_iterator<int>(cout," "), // destination
                    bind2nd(modulus<int>(),2), // replacement criterion
                    0);                             // new value
    cout << endl;
}
```

程序输出如下：

```
2 3 4 5 6 4 5 6 7 8 9
2 3 4 55 6 4 55 6 7 8 9
42 42 42 5 6 42 5 6 7 8 9
2 0 4 0 6 4 0 6 0 8 0
```

9.7 移除性算法（Removing Algorithms）

本节所列算法系根据元素值或某一准则，在一个区间内移除某些元素。这些算法并不能改变元素的数量，它们只是以逻辑上的思考，将原本置于后面的“不移除元素”向前移动，覆盖那些被移除元素而已。它们都返回新区间的逻辑终点（也就是最后一个“不移除元素”的下一位置）。细节详见 5.6.1 节, p111。

9.7.1 移除某些特定元素

移除某序列内的元素

```
ForwardIterator
remove (ForwardIterator beg, ForwardIterator end,
        const T& value)
ForwardIterator
remove_if (ForwardIterator beg, ForwardIterator end,
           UnaryPredicate op)
```

- remove()会移除区间[beg, end)中每一个“与 value 相等”的元素。
- remove_if()会移除区间[beg, end)中每一个“令以下一元判断式：
  op(elem)
  获得 true”的元素。
- 两个算法都返回变动后的序列的新逻辑终点（也就是最后一个未被移除元素的下一位置）。
- 这些算法会把原本置于后面的未移除元素向前移动，覆盖被移除元素。
- 未被移除的元素在相对次序上保持不变。
- 调用者在调用此算法之后，应保证从此采用返回的新逻辑终点，而不再使用原始终点 end（细节参见 5.6.1 节, p111）。
- op 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 注意，remove_if()通常会在内部复制它所获得的那个一元判断式，然后两次运用它。如果该一元判断式在函数调用过程中改变状态，就可能导致问题。细节见 8.1.4 节，p302。
- 由于会发生元素变动，所以这些算法不可用于关联式容器（参见 5.6.2 节, p115）。关联式容器提供了功能相似的成员函数 erase()（参见 p242）。
- Lists 提供了一个等效成员函数 remove()：不是重新赋值元素，而是重新安排指针，因此具有更佳性能（参见 p242）。
- 复杂度：线性，执行比较动作（或调用 op()）*numberOfElements* 次。

以下程序示范 remove()和 remove_if()的用法：

```
// algo/remove1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    INSERT_ELEMENTS(coll,1,7);
    PRINT_ELEMENTS(coll,"coll:          ");

    // remove all elements with value 5
    vector<int>::iterator pos;
    pos = remove(coll.begin(), coll.end(),     // range
                 5);                           // value to remove
    PRINT_ELEMENTS(coll,"size not changed: ");

    // erase the "removed" elements in the container
    coll.erase(pos, coll.end());
    PRINT_ELEMENTS(coll,"size changed: ");

    // remove all elements less than 4
    coll.erase(remove_if(coll.begin(), coll.end(), // range
                         bind2nd(less<int>(),4)), // remove criterion
                         coll.end());
    PRINT_ELEMENTS(coll,"<4 removed: ");
}
```

程序输出如下：

```
coll:              2 3 4 5 6 4 5 6 7 8 9 1 2 3 4 5 6 7
size not changed:  2 3 4 6 4 6 7 8 9 1 2 3 4 6 7 5 6 7
size changed:      2 3 4 6 4 6 7 8 9 1 2 3 4 6 7
<4 removed:        4 6 4 6 7 8 9 4 6 7
```

复制时一并移除元素

```
OutputIterator
remove_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             const T& value)
OutputIterator
remove_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                OutputIterator destBeg,
                UnaryPredicate op)
```

- remove_copy()是 copy()和 remove()的组合。它将源区间[beg,end)内的所有元素复制到“以 destBeg 为起点”的目标区间去，并在复制过程中移除“与 value 相等”的所有元素。
- remove_copy_if()是 copy()和 remove_if()的组合。它将源区间[beg,end)内的元素复制到“以 destBeg 为起点”的目标区间去，并在复制过程中移除“造成以下一元判断式：

  op(elem)

  结果为 true”的所有元素。
- 两个算法都返回目标区间中最后一个被复制元素的下一位置（也就是第一个未被覆盖的元素）。
- op 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 复杂度：线性，执行比较动作（或调用 op()）*numberOfElements* 次。

以下程序示范 remove_copy()和 remove_copy_if()的用法：

```
// algo/remove2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll1;

    INSERT_ELEMENTS(coll1,1,6);
    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1);
```

```
    // print elements without those having the value 3
    remove_copy(coll1.begin(), coll1.end(),        // source
                ostream_iterator<int>(cout," "),   // destination
                3);                                // removed value
    cout << endl;

    // print elements without those having a value greater than 4
    remove_copy_if(coll1.begin(), coll1.end(),     // source
                   ostream_iterator<int>(cout," "), // destination
                   bind2nd(greater<int>(),4)); // removed elements
    cout << endl;

    // copy all elements greater than 3 into a multiset
    multiset<int> coll2;
    remove_copy_if(coll1.begin(), coll1.end(),     // source
                   inserter(coll2,coll2.end()),    // destination
                   bind2nd(less<int>(),4)); // elements not copied
    PRINT_ELEMENTS(coll2);
}
```

程序输出如下：

```
1 2 3 4 5 6 1 2 3 4 5 6 7 8 9
1 2 4 5 6 1 2 4 5 6 7 8 9
1 2 3 4 1 2 3 4
4 4 5 5 6 6 7 8 9
```

9.7.2 移除重复元素

移除连续重复元素

```
ForwardIterator
unique (ForwardIterator beg, ForwardIterator end)

ForwardIterator
unique (ForwardIterator beg, ForwardIterator end,
        BinaryPredicate op)
```

- 以上两种形式都会移除连续重复元素中的多余元素。
- 第一形式将区间[*beg*,*end*)内所有“与前一元素相等”的元素移除。所以，源序列必须先经过排序，才能使用这个算法移除所有重复元素。

- 第二形式将每一个“位于元素 e 之后并且造成以下二元判断式：

  op(elem,e)

  结果为 **true** 的所有 *elem* 元素移除。换言之此一判断式并非用来将元素和其原本的前一元素比较，而是将它和未被移除的前一元素比较，参见以下实例（译注：换言之，如果序列 {A,B,C,D,E}，A 不符合移除条件，B 符合，轮到 C 时，C 将被拿来和 A 比较，而不是和原本的前一个元素（但已被移除的）B 比较）
- 两种形式都返回被变动后的序列新终点（逻辑终点，也就是最后一个“未被移除的元素”的下一个位置）。
- 这两个算法将“原本位置在后”的未移除元素向前移动，覆盖（overwrite）被移除元素。
- 未被移除的元素在相对次序上保持不变。
- 调用者在调用这些算法之后，应保证从此使用返回的新逻辑终点，不再使用原始终点 *end*（详见 5.6.1 节，p111）。
- op 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- 由于会造成元素变动，所以这些算法不可用于关联式容器（见 5.6.2 节，p115）。
- Lists 提供了一个等效成员函数 unique()，不是重新赋值元素，而是重新安排指针，因此具有更佳性能（参见 p244）。
- 复杂度：线性，执行比较操作（或调用 *op()*）*numberOfElements* 次。

以下程序示范 unique() 的用法：

```
// algo/unique1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    // source data
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                     5, 4, 4 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    list<int> coll;

    // initialize coll with elements from source
    copy (source, source+sourceNum,  // source
          back_inserter(coll));      // destination
    PRINT_ELEMENTS(coll);

    // remove consecutive duplicates
    list<int>::iterator pos;
```

```
    pos = unique (coll.begin(), coll.end());

    /* print elements not removed
    * - use new logical end
    */
    copy (coll.begin(), pos, // source
          ostream_iterator<int>(cout," ")); // destination
    cout << "\n\n";

    // reinitialize coll with elements from source
    copy (source, source+sourceNum,  // source
          coll.begin());             // destination
    PRINT_ELEMENTS(coll);

    // remove elements if there was a previous greater element
    coll.erase (unique (coll.begin(), coll.end(),
                        greater<int>()),
                coll.end());
    PRINT_ELEMENTS(coll);
}
```

程序输出如下：

```
1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4

1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 4 6 6 6 6 7
```

第一次调用 unique()是为了移除连续重复元素。第二次调用则示范 unique()第二形式的行为：将“greater 比较结果为 true”的所有元素移除。例如第一个元素值是 6，大于其后的 1, 2, 2, 3 和 1，所以后面这些元素都被移除。换言之，该判断式不是用来将元素和其直接前趋元素比较，而是将它和未被移除的前趋元素比较（稍后的 unique_copy()另有一个类似例子）。

复制过程中移除重复元素

```
OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             BinaryPredicate op)
```

- 两种形式都是 copy() 和 unique() 的组合。
- 两者都将源区间 [*sourceBeg*, *sourceEnd*) 内的元素复制到“以 *destBeg* 起始的目标区间”，并移除重复元素。
- 两个算法都返回目标区间内“最后一个被复制的元素”的下一位置（也就是第一个未被覆盖的元素）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 复杂度：线性，执行比较动作（或调用 op()）*numberOfElements* 次。

以下程序示范 unique_copy() 的用法：

```
// algo/unique2.cpp

#include "algostuff.hpp"
using namespace std;

bool differenceOne (int elem1, int elem2)
{
    return elem1 + 1 == elem2 || elem1 - 1 == elem2;
}

int main()
{
    // source data
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                     5, 4, 4 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    // initialize coll with elements from source
    list<int> coll;
    copy(source, source+sourceNum,   // source
         back_inserter(coll));       // destination
    PRINT_ELEMENTS(coll);
```

```
    // print element with consecutive duplicates removed
    unique_copy(coll.begin(), coll.end(),          // source
                ostream_iterator<int>(cout," "));  // destination
    cout << endl;

    // print element without consecutive duplicates that
    // differ by one
    unique_copy(coll.begin(), coll.end(),          // source
                ostream_iterator<int>(cout," "),   // destination
                differenceOne());             // duplicates criterion
    cout << endl;
}
```

程序输出如下：

```
1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4
1 4 4 6 1 3 1 6 6 6 4 4
```

注意，第二次调用 unique_copy()后并未移除“与其直接前趋元素相差 1”的所有元素，而是移除“与其未被移除之前趋元素相差 1”的所有元素。例如，出现于三个 6 之后，紧跟着的元素是 5, 7, 5，都与 6 相差 1，所以被移除。然而更后面的两个 4，并非和 6 相差 1，所以被保留下来。

另有一个用来压缩空白序列的例子：

```
// algo/unique3.cpp

#include <iostream>
#include <algorithm>
using namespace std;

bool bothSpaces (char elem1, char elem2)
{
    return elem1 == ' ' && elem2 == ' ';
}

int main()
{
    // don't skip leading whitespaces by default
    cin.unsetf(ios::skipws);

    /* copy standard input to standard output
    * - while compressing spaces
    */
```

```
    unique_copy(istream_iterator<char>(cin),   // beginning of source: cin
                istream_iterator<char>(),  // end of source: end-of-file
                ostream_iterator<char>(cout),  // destination: cout
                bothSpaces);                   // duplicate criterion
}
```

当输入如下：

```
Hello, here are   sometimes more   and sometimes fewer     spaces.
```

输出结果是：

```
Hello, here are sometimes more and sometimes fewer spaces.
```

9.8 变序性算法（Mutating Algorithms）

译注：某些书籍如《*Generic Programming and the STL*》和《STL源码剖析》中，mutating algorithm是指变动性（更易性）算法，也就是本书的modifying algorithm。本书划分更为细腻。至于我将mutating algorithm译为变序性算法，是着眼于其实际效应。C++关键词mutable的意义意思是“即使在const object内仍可变动的”。

变序性算法改变元素的次序，但不改变元素值。这些算法不能用于关联式容器，因为在关联式容器中，元素有一定的次序，不能随意变动。

9.8.1 逆转（Reversing）元素次序

```
void
reverse (BidirectionalIterator beg, BidirectionalIterator end)

OutputIterator
reverse_copy (BidirectionalIterator sourceBeg,
              BidirectionalIterator sourceEnd,
              OutputIterator destBeg)
```

- reverse()会将区间[*beg*, *end*)内的元素全部逆序。
- reverse_copy()会将源区间[*sourceBeg*, *sourceEnd*)内的元素复制到“以destBeg起始的目标区间”，并在复制过程中颠倒安置次序。
- reverse_copy()返回目标区间内最后一个被复制元素的下一位置，也就是第一个未被覆盖（overwritten）的元素。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- Lists提供了一个等效成员函数reverse()，不是重新赋值元素，而是重新安排指针，因此具有更佳性能（参见p246）。
- 复杂度：线性，分别进行 *numberOfElements* / 2 次交换操作或 *numberOfElements* 次赋值（*assign*）操作。

下面这个程序展示 reverse()和 reverse_copy()的用法：

```
// algo/reverse1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order of elements
    reverse (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order from second to last element but one
    reverse (coll.begin()+1, coll.end()-1);
    PRINT_ELEMENTS(coll,"coll: ");

    // print all of them in reverse order
    reverse_copy (coll.begin(), coll.end(),        // source
                  ostream_iterator<int>(cout," ")); // destination
    cout << endl;
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
coll: 9 8 7 6 5 4 3 2 1
coll: 9 2 3 4 5 6 7 8 1
1 8 7 6 5 4 3 2 9
```

9.8.2 旋转（Rotating）元素次序

旋转序列内的元素

```
void
rotate (ForwardIterator beg, ForwardIterator newBeg,
        ForwardIterator end)
```

- 将区间[*beg*, *end*)内的元素进行旋转，执行后 **newBeg* 成为新的第一元素。
- 调用者必须确保 *newBeg* 是区间[*beg*, *end*)内的一个有效位置，否则会引发未定义的行为。
- 复杂度：线性，最多进行 *numberOfElements* 次交换动作。

以下程序示范如何使用 rotate()：

```
// algo/rotate1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // rotate one element to the left
    rotate (coll.begin(),       // beginning of range
            coll.begin() + 1,   // new first element
            coll.end());        // end of range
    PRINT_ELEMENTS(coll,"one left: ");

    // rotate two elements to the right
    rotate (coll.begin(),       // beginning of range
            coll.end() - 2,     // new first element
            coll.end());        // end of range
    PRINT_ELEMENTS(coll,"two right: ");

    // rotate so that element with value 4 is the beginning
    rotate (coll.begin(),                    // beginning of range
            find(coll.begin(),coll.end(),4), // new first element
            coll.end());                     // end of range
    PRINT_ELEMENTS(coll,"4 first: ");
}
```

正如上例所示，你可以使用正偏移量（positive offset）将元素向左起点方向旋转，也可以使用负偏移量（negative offset）将元素向右终点方向旋转。不过请注意，只有在随机存取迭代器身上才能为它加上偏移量。如果不是这类迭代器，你就得使用 advance()（参见 p389, rotate_copy()的例子）。

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
one left: 2 3 4 5 6 7 8 9 1
two right: 9 1 2 3 4 5 6 7 8
4 first: 4 5 6 7 8 9 1 2 3
```

复制并同时旋转元素

```
OutputIterator
rotate_copy (ForwardIterator sourceBeg, ForwardIterator newBeg,
             ForwardIterator sourceEnd,
             OutputIterator destBeg)
```

- 这是 copy()和 rotate()的组合。
- 将源区间[*sourceBeg*, *sourceEnd*)内的元素复制到“以 *destBeg* 起始的目标区间”中，同时旋转元素，使 *newBeg* 成为新的第一元素。
- 返回目标区间内最后一个被复制元素的下一位置。
- 调用者必须确保 *newBeg* 是区间[*beg*, *end*)内的一个有效位置，否则会引发未定义的行为。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 源区间和目标区间两者不可重迭。
- 复杂度：线性，执行 ***numberOfElements*** 次赋值（*assign*）操作。

以下程序示范 rotate_copy()的用法：

```
// algo/rotate2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    set<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);
```

```
    // print elements rotated one element to the left
    set<int>::iterator pos = coll.begin();
    advance(pos,1);
    rotate_copy(coll.begin(),    // beginning of source
                pos,             // new first element
                coll.end(),      // end of source
                ostream_iterator<int>(cout," ")); // destination
    cout << endl;

    // print elements rotated two elements to the right
    pos = coll.end();
    advance(pos,-2);
    rotate_copy(coll.begin(),    // beginning of source
                pos,             // new first element
                coll.end(),      // end of source
                ostream_iterator<int>(cout," ")); // destination
    cout << endl;

    // print elements rotated so that element with value 4
    // is the beginning
    rotate_copy(coll.begin(),    // beginning of source
                coll.find(4),    // new first element
                coll.end(),      // end of source
                ostream_iterator<int>(cout," ")); // destination
    cout << endl;
}
```

与先前的 rotate() 实例不同(参见 p388)，这里用了一个 set 而不是一个 vector。这导致两个结果：

1. 你必须使用 advance()（参见 7.3.1 节，p259）来改变迭代器本身的值，因为双向迭代器不支持 operator+。
2. 你应当使用 find() 成员函数，而非 find() 算法，因为前者有更好的效能。

程序输出如下：

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
8 9 1 2 3 4 5 6 7
4 5 6 7 8 9 1 2 3
```

9.8.3 排列（Permuting）元素

```
bool
next_permutation (BidirectionalIterator beg,
                  BidirectionalIterator end)
bool
prev_permutation (BidirectionalIterator beg,
                  BidirectionalIterator end)
```

- next_permutation()会改变区间[*beg*, *end*)内的元素次序，使它们符合“下一个排列次序”。
- prev_permutation()会改变区间[*beg*, *end*)内的元素次序，使它们符合“上一个排列次序”。
- 如果元素得以排列成（就字典顺序而言的）“正规（normal）”次序，则两个算法都返回 true。所谓正规次序，对 next_permutation()而言为升序，对 prev_permutation()而言为降序。因此，如果要走遍所有排列，你必须先将所有元素（按升序或降序）排序，然后开始以循环方式调用 next_permutation 或 prev_permutation，直到算法返回 false[5]。
- 所谓“字典次序”的排序，p360 有解释。
- 复杂度：线性，最多执行 *numberOfElements* / 2 次交换操作。

下面这个例子展示利用 next_permutation()和 prev_permutation()获得所有元素的所有可能排列的过程：

```
// algo/perm1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,3);
    PRINT_ELEMENTS(coll,"on entry: ");

    /* permute elements until they are sorted
    * - runs through all permutations because the elements are sorted now
    */
```

[5] next_permutation()和 prev_permutation()也可用来在区间内对元素排序。你只需一再调用它们直到它们传回 false。然而这么做的效率很低。

```
    while (next_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }

    PRINT_ELEMENTS(coll,"afterward: ");

    /* permute until descending sorted
    * - this is the next permutation after ascending sorting
    * - so the loop ends immediately
    */
    while (prev_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }

    PRINT_ELEMENTS(coll,"now: ");

    /* permute elements until they are sorted in descending order
    * - runs through all permutations because the elements are sorted
    * in descending order now
    */
    while (prev_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }

    PRINT_ELEMENTS(coll,"afterward: ");
}
```

程序输出如下：

```
on entry: 1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
afterward: 1 2 3
now:       3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
afterward: 3 2 1
```

9.8.4 重排元素（Shuffling，搅乱次序）

```
void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)
void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,
                RandomFunc& op)
```

- 第一形式使用一个均匀分布随机数产生器（uniform distribution random number generator）来打乱区间 [*beg*, *end*) 内的元素次序。
- 第二形式使用 *op* 打乱区间 [*beg*, *end*) 内的元素次序。算法内部会使用一个整数值（其型别为“迭代器所提供之 `difference_type`”）来调用 *op*：

  op(*max*)

  它应该返回一个大于零而小于 *max* 的随机数，不包括 *max* 本身。
- 注意，*op* 是一个 non-const reference。所以你不可以将暂时数值或一般函数传进去。
- 复杂度：线性，执行 *numberOfElements* -1 次交换动作。

你大概会问，为什么 `random_shuffle()` 那个可有可无的参数（代表一个操作）是个 non-const reference 呢？必须如此，因为典型的随机数产生器拥有一个局部状态（local state）。旧式 C 函数如 `rand()` 是将其局部状态存储在某个静态变量中。但是这有一些缺点，例如这种随机数发生器本质上对于多线程（multi-threads）而言就不安全，而且你也不可能拥有两个各自独立的随机数流（streams）。如果使用仿函数，其区域状态被封装为一个或多个成员变量，那么就有了比较好的解决方案。这样一来，随机数产生器就不可能具备常数性，否则何以改变内部状态，何以产生新的随机数呢？不过你还是可以用 by value 方式传递随机数产生器，为什么非要以 by non-const reference 方式传递呢？是这样的，如果这么做，每次调用都会在内部复制一个随机数产生器及其状态，结果，每次你传入随机数产生器，所得的随机数序列都一样，那又有何随机可言？现在你明白为什么要以 by non-const reference 方式传递了吧[6]☺。

如果你需要获得同一个随机数序列两次，复制它就是了。但如果那个随机数产生器的实作手法涉及全局状态（global state），你还是会获得不同的序列。

以下程序示范如何调用 `random_shuffle()` 来打乱元素次序：

```
// algo/random1.cpp

#include <cstdlib>
```

[6] 感谢 Matt Austern 就这个问题所提供的解释。

```
#include "algostuff.hpp"
using namespace std;

class MyRandom {
    public:
    ptrdiff_t operator() (ptrdiff_t max) {
        double tmp;
        tmp = static_cast<double>(rand())
                  / static_cast<double>(RAND_MAX);
        return static_cast<ptrdiff_t>(tmp * max);
    }
};
int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // shuffle all elements randomly
    random_shuffle (coll.begin(), coll.end());

    PRINT_ELEMENTS(coll,"shuffled: ");

    // sort them again
    sort (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"sorted: ");

    /* shuffle elements with self-written random number generator
    * - to pass an lvalue we have to use a temporary object
    */
    MyRandom rd;
    random_shuffle (coll.begin(), coll.end(), // range
                    rd);              // random number generator
    PRINT_ELEMENTS(coll,"shuffled: ");
}
```

第二次调用 random()时用了一个自定义的随机数产生器 rd()。它是根据辅助仿函数 MyRandom 而产生的一个对象，采用一个随机数计算法。它的效果通常比直接调用 rand()好一些[7]。

[7] MyRandom 的随机数产生法在 Bjarne Stroustrup 的《*The C++ Programming Language*》, 3/e 中有介绍和解释。

一个可能（但非必然）的输出如下：

```
coll:       1 2 3 4 5 6 7 8 9
shuffled:   2 6 9 5 4 3 1 7 8
sorted:     1 2 3 4 5 6 7 8 9
shuffled:   2 6 9 3 1 8 7 4 5
```

9.8.5 将元素向前搬移

```
BidirectionalIterator
partition (BidirectionalIterator beg,
           BidirectionalIterator end,
           UnaryPredicate op)

BidirectionalIterator
stable_partition (BidirectionalIterator beg,
                  BidirectionalIterator end,
                  UnaryPredicate op)
```

- 这两种算法将区间[*beg*, *end*)中“造成以下一元判断式：
  op(*elem*)
  结果为 true”的元素向前端移动。
- 这两种算法都返回“令 *op*()结果为 **false**”的第一个元素位置。
- 两者差别是，无论元素是否符合给定的准则，stable_partition()会保持它们之间的相对次序。
- 你可以运用此算法，根据排序准则，将所有元素分割为两部分。nth_element()具有类似能力。至于本算法和 nth_element()之间的区别，参见 p330。
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 复杂度：
  - partition()：线性，总共执行 *op*()操作 ***numberOfElements*** 次，以及最多 ***numberOfElements*** / 2 次的交换操作。
  - stable_partition()：如果系统拥有足够的内存，那么就是线性复杂度，执行 *op*()操作及交换操作共 ***numberOfElements*** 次；如果没有足够内存，则是 n-log-n，执行 *op*()操作 ***numberOfElements*** * log(***numberOfElements***)次。

以下程序示范 partition()和 stable_partition()的用法以及两者的区别：

```
// algo/part1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
int main()
{
  vector<int> coll1;
  vector<int> coll2;

  INSERT_ELEMENTS(coll1,1,9);
  INSERT_ELEMENTS(coll2,1,9);
  PRINT_ELEMENTS(coll1,"coll1: ");
  PRINT_ELEMENTS(coll2,"coll2: ");
  cout << endl;

  // move all even elements to the front
  vector<int>::iterator pos1, pos2;
  pos1 = partition(coll1.begin(), coll1.end(),          // range
                   not1(bind2nd(modulus<int>(),2)));    // criterio
  pos2 = stable_partition(coll2.begin(), coll2.end(),   // range
                          not1(bind2nd(modulus<int>(),2))); // crit

  // print collections and first odd element
  PRINT_ELEMENTS(coll1,"coll1: ");
  cout << "first odd element: " << *pos1 << endl;
  PRINT_ELEMENTS(coll2,"coll2: ");
  cout << "first odd element: " << *pos2 << endl;
}
```

程序输出如下：

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 1 2 3 4 5 6 7 8 9

coll1: 8 2 6 4 5 3 7 1 9
first odd element: 5
coll2: 2 4 6 8 1 3 5 7 9
first odd element: 1
```

正如此例所展示的，stable_partition()保持了奇数元素和偶数元素的相对次序，这一点和 partition()不同。

9.9 排序算法（Sorting Algorithms）

STL 提供了好几种算法来对区间内的元素排序。除了完全排序（full sorting）外，还支持局部排序（partial sorting）的数个变体。如果这些变体的功能对你已经足够，你应该优先使用它们，因为通常它们的性能更佳。

你也可以使用关联式容器，让元素自动排序。然而请注意，对全体元素进行一次性排序，通常比始终维护它们保持已序（*sorted*）状态来得高效一些（细节参见 p228）。

9.9.1 对所有元素排序

```
void
sort (RandomAccessIterator beg, RandomAccessIterator end)

void
sort (RandomAccessIterator beg, RandomAccessIterator end,
      BinaryPredicate op)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end,
             BinaryPredicate op)
```

- sort()和 stable_sort()的上述第一形式，使用 operator< 对区间[*beg*,*end*)内的所有元素进行排序。
- sort()和 stable_sort()的上述第二形式，使用二元判断式 *op*(*elem1*,*elem2*)作为排序准则，对区间[*beg*,*end*)内的所有元素进行排序。
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节，p302。
- sort()和 stable_sort()的区别是，后者保证相等元素的原本相对次序在排序后保持不变。
- 不可以对 lists 调用这些算法，因为 lists 不支持随机存取迭代器。不过 lists 提供了一个成员函数 sort()，可用来对其自身元素排序，参见 p245。
- sort()保证很不错的平均效能 n-log-n。然而如果你必须极力避免可能出现的最差状况，你应该使用 partial_sort()或 stable_sort()。参见 p328 对于排序算法的讨论。
- 复杂度：
  - sort()：平均 n-log-n（平均大约执行比较操作共 *numberOfElements* * log(*numberOfElements*)次）。

— stable_sort()：如果系统拥有足够内存，那么就是 n-log-n，也就是执行比较操作 *numberOfElements* * log(*numberOfElements*) 次；如果没有足够内存，则复杂度是 n-log-n * log-n，亦即执行比较操作 *numberOfElements** *log(numberOfElements)*^{2} 次。

下面这个例子示范 sort()的用法：

```
// algo/sort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"on entry: ");

    // sort elements
    sort (coll.begin(), coll.end());

    PRINT_ELEMENTS(coll,"sorted: ");

    // sorted reverse
    sort (coll.begin(), coll.end(),  // range
          greater<int>());           // sorting criterion

    PRINT_ELEMENTS(coll,"sorted >: ");
}
```

程序输出如下：

```
on entry: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
sorted: 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
sorted >: 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
```

至于如何根据某个 class member 进行排序，请见 p123 实例。

以下程序示范 sort()和 stable_sort()两者间的区别。该程序通过排序准则 lessLength()，将字符串按照字符数量排序：

```
// algo/sort2.cpp

#include "algostuff.hpp"
using namespace std;

bool lessLength (const string& s1, const string& s2)
{
   return s1.length() < s2.length();
}

int main()
{
   vector<string> coll1;
   vector<string> coll2;

   // fill both collections with the same elements
   coll1.push_back ("1xxx");
   coll1.push_back ("2x");
   coll1.push_back ("3x");
   coll1.push_back ("4x");
   coll1.push_back ("5xx");
   coll1.push_back ("6xxxx");
   coll1.push_back ("7xx");
   coll1.push_back ("8xxx");
   coll1.push_back ("9xx");
   coll1.push_back ("10xxx");
   coll1.push_back ("11");
   coll1.push_back ("12");
   coll1.push_back ("13");
   coll1.push_back ("14xx");
   coll1.push_back ("15");
   coll1.push_back ("16");
   coll1.push_back ("17");
   coll2 = coll1;

   PRINT_ELEMENTS(coll1,"on entry:\n ");

   // sort (according to the length of the strings)
   sort (coll1.begin(), coll1.end(),     // range
         lessLength);                    // criterion

   stable_sort (coll2.begin(), coll2.end(),  // range
                lessLength);                 // criterion
   PRINT_ELEMENTS(coll1,"\nwith sort():\n ");
   PRINT_ELEMENTS(coll2,"\nwith stable_sort():\n ");
}
```

程序输出如下：

```
on entry:
1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17

with sort():
17 2x 3x 4x 16 15 13 12 11 9xx 7xx 5xx 8xxx 14xx 1xxx 10xxx 6xxxx

with stable_sort():
2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx
```

只有 stable_sort()保持了元素的相对位置(每个字符串最前头的阿拉伯数字标识出原本的元素顺序)。

9.9.2 局部排序（Partial Sorting）

```
void
partial_sort (RandomAccessIterator beg,
              RandomAccessIterator sortEnd,
              RandomAccessIterator end)

void
partial_sort (RandomAccessIterator beg,
              RandomAccessIterator sortEnd,
              RandomAccessIterator end,
              BinaryPredicate op)
```

- 以上第一形式，以 operator< 对区间[beg,end) 内的元素进行排序，使区间[beg,sortEnd)内的元素处于有序状态（*sorted order*）。
- 以上第二形式，运用二元判断式：

  op(elem1,elem2)

  对区间[beg,end)内的元素进行排序，使区间[beg,sortEnd)内的元素处于有序状态（*sorted order*）。
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- 和 sort()不同的是，partial_sort()并不对全部元素排序：一旦第一个元素至 *sortEnd* 之间的所有元素都排妥次序，就立刻停止。所以如果你只需要前 3 个已序元素，可以使用 partial_sort()来节省时间，因为它不会对剩余的元素进行非必要的排序。

- 如果 *sortEnd* 和 *end* 相等，那么 partial_sort()会对整个序列进行排序。平均而言其效率不及 sort()，不过以最差情况而论则优于 sort()。请参考 p328 关于排序算法的讨论。
- 复杂度：在线性和 n·log·n 之间，大约执行 *numberOfElements* * log(*numberOf-SortedElements*) 次比较操作。

以下程序示范 partial_sort()的用法：

```
// algo/psort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  deque<int> coll;

  INSERT_ELEMENTS(coll,3,7);
  INSERT_ELEMENTS(coll,2,6);
  INSERT_ELEMENTS(coll,1,5);
  PRINT_ELEMENTS(coll);

  // sort until the first five elements are sorted
  partial_sort (coll.begin(),       // beginning of the range
                coll.begin()+5,     // end of sorted range
                coll.end());        // end of full range
  PRINT_ELEMENTS(coll);

  // sort inversely until the first five elements are sorted
  partial_sort (coll.begin(),       // beginning of the range
                coll.begin()+5,     // end of sorted range
                coll.end(),         // end of full range
                greater<int>());    // sorting criterion
  PRINT_ELEMENTS(coll);

  // sort all elements
  partial_sort (coll.begin(),       // beginning of the range
                coll.end(),         // end of sorted range
                coll.end());        // end of full range
  PRINT_ELEMENTS(coll);
}
```

程序输出如下：

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 7 6 5 5 6 4 4 3 4 5
7 6 6 5 5 1 2 2 3 3 4 4 3 4 5
1 2 2 3 3 3 4 4 4 5 5 5 6 6 7
```

```
RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg,
                   InputIterator sourceEnd,
                   RandomAccessIterator destBeg,
                   RandomAccessIterator destEnd)

RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg,
                   InputIterator sourceEnd,
                   RandomAccessIterator destBeg,
                   RandomAccessIterator destEnd,
                   BinaryPredicate op)
```

- 两者都是 copy()和 partial_sort()的组合。
- 它们将元素从源区间 [*sourceBeg*, *sourceEnd*) 复制到目标区间 [*destBeg*, *destEnd*)，同时进行排序。
- “被排序（被复制）的元素数量”是源区间和目标区间两者所含元素数量的较小值。
- 两者都返回目标区间内“最后一个被复制元素”的下一位置（也就是第一个未被覆盖的元素）。
- 如果目标区间 [*destBeg*, *destEnd*) 内的元素数量大于或等于源区间 [*sourceBeg*, *sourceEnd*) 内的元素数量，则所有元素都会被排序并复制，整个行为就相当于 copy()和 sort()的组合。
- 复杂度：在线性和 n-log-n 之间，大约执行 *numberOfElements* * log(*numberOf-SortedElements*) 次比较操作。

以下程序示范 partial_sort_copy()的用法：

```
// algo/psort2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  deque<int> coll1;
  vector<int> coll6(6);    // initialize with 6 elements
  vector<int> coll30(30);  // initialize with 30 elements
```

```
    INSERT_ELEMENTS(coll1,3,7);
    INSERT_ELEMENTS(coll1,2,6);
    INSERT_ELEMENTS(coll1,1,5);
    PRINT_ELEMENTS(coll1);

    // copy elements of coll1 sorted into coll6
    vector<int>::iterator pos6;
    pos6 = partial_sort_copy (coll1.begin(), coll1.end(),
                              coll6.begin(), coll6.end());

    // print all copied elements
    copy (coll6.begin(), pos6,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // copy elements of coll1 sorted into coll30
    vector<int>::iterator pos30;
    pos30 = partial_sort_copy (coll1.begin(), coll1.end(),
                              coll30.begin(), coll30.end(),
                              greater<int>());

    // print all copied elements
    copy (coll30.begin(), pos30,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

程序输出如下：

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 3
7 6 6 5 5 5 4 4 4 3 3 3 2 2 1
```

第一次调用 partial_sort_copy()时，目标区间内只有 6 个元素，所以该算法只复制了 6 个元素，返回 coll6 的终点。第二次调用 parital_sort_copy()时，由于 coll30 有充足的空间，所以 coll1 的所有元素都被复制并排序。

9.9.3 根据第 n 个元素排序

```
void
nth_element (RandomAccessIterator beg,
             RandomAccessIterator nth,
             RandomAccessIterator end)

void
nth_element (RandomAccessIterator beg,
             RandomAccessIterator nth,
             RandomAccessIterator end,
             BinaryPredicate op)
```

- 两种形式都对区间 [*beg*, *end*) 内的元素进行排序，使第 n 个位置上的元素就位，也就是说，所有在位置 n 之前的元素都小于等于它，所有在位置 n 之后的元素都大于等于它。这样，你就得到了“根据 n 位置上的元素”分割开来的两个子序列，第一子序列的元素统统小于第二子序列的元素。如果你只需要 n 个最大或最小元素，但不要求它们必须已序（*sorted*），那么这个算法就很有用。
- 上述第一形式使用 operator< 作为排序准则。
- 上述第二形式使用以下二元判断式作为排序准则：
  op(*elem1*, *elem2*)
- *op* 不应该在函数调用过程中改变自身状态。详见 8.1.4 节, p302。
- patition()算法（参见 p395）也可以根据某个排序准则，将序列中的元素分割成两部分。至于 patition()和 nth_element()间的区别，请参考 p330。
- 复杂度：平均而言为线性。

以下程序示范 nth_element()的用法：

```
// algo/nth1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  deque<int> coll;

  INSERT_ELEMENTS(coll,3,7);
  INSERT_ELEMENTS(coll,2,6);
  INSERT_ELEMENTS(coll,1,5);
  PRINT_ELEMENTS(coll);
```

```
  // extract the four lowest elements
  nth_element (coll.begin(),    // beginning of range
               coll.begin()+3,  // element that should be sorted correctly
               coll.end());     // end of range

  // print them
  cout << "the four lowest elements are: ";
  copy (coll.begin(), coll.begin()+4,
        ostream_iterator<int>(cout," "));
  cout << endl;

  // extract the four highest elements
  nth_element (coll.begin(),    // beginning of range
               coll.end()-4,    // element that should be sorted correctly
               coll.end());     // end of range

  // print them
  cout << "the four highest elements are: ";
  copy (coll.end()-4, coll.end(),
        ostream_iterator<int>(cout," "));
  cout << endl;

  // extract the four highest elements (second version)
  nth_element (coll.begin(),    // beginning of range
               coll.begin()+3,  // element that should be sorted correctly
               coll.end(),      // end of range
               greater<int>()); // sorting criterion

  // print them
  cout << "the four highest elements are: ";
  copy (coll.begin(), coll.begin()+4,
        ostream_iterator<int>(cout," "));
  cout << endl;
}
```

程序输出如下：

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
the four lowest elements are: 2 1 2 3
the four highest elements are: 5 6 7 6
the four highest elements are: 6 7 6 5
```

9.9.4 Heap 算法

就排序而言，heap 是一种特别的元素组织方式，应用于 heap 排序法（heapsort）。heap 可被视为一个以序列式群集（sequential collection）实作而成的二叉树，具有两大性质（译注：细节可参考《STL 源码剖析》4.7 节）：

1. 第一个元素总是最大。
2. 总是能够在对数时间内增加或移除一个元素。

heap 是实作 priority queue（其内元素会自动排序）的一个理想结构。因此，heap 算法也在 priority_queue 容器中有所应用（参见 10.3 节，p453）。为了处理 heap，STL 提供四种算法：

1. `make_heap()` 将某区间内的元素转化成 heap。
2. `push_heap()` 对着 heap 增加一个元素。
3. `pop_heap()` 对着 heap 取出下一个元素。
4. `sort_heap()` 将 heap 转化为一个已序群集（此后它就不再是 heap 了）。

就像以前常见的情况一样，你可以传递一个二元判断式作为排序准则。缺省的排序准则是 `operator<`。

heap 算法细节

```
void
make_heap (RandomAccesIterator beg,
           RandomAccesIterator end)
void
make_heap (RandomAccesIterator beg,
           RandomAccesIterator end,
           BinaryPredicate op)
```

- 两种形式都将区间 [*beg*, *end*) 内的元素转化为 heap。
- *op* 是一个可有可无的（可选的）二元判断式，被视为排序准则：

  op(*elem1*, *elem2*)

- 只有在多于一个元素的情况下，才有必要使用这些函数来处理 heap，如果只有单一元素，那么它自动就形成一个 heap。
- 复杂度：线性，最多执行 *3 * numberOfElements* 次比较动作。

```
void
push_heap (RandomAccesIterator beg, RandomAccesIterator end)

void
push_heap (RandomAccesIterator beg, RandomAccesIterator end,
           BinaryPredicate op)
```

- 两种形式都将 *end* 之前的最后一个元素加入原本就是个 heap 的 [*beg*, *end*-1) 区间内，使整个区间 [*beg*, *end*) 成为一个 heap。

- *op* 是一个可有可无的（可选的）二元判断式，被视为排序准则：
  op(elem1,elem2)
- 调用者必须保证，进入函数时，区间 [*beg*, *end-1*) 内的元素原本便已形成一个 heap（在相同的排序准则下），而新元素紧跟其后。
- 复杂度：对数，最多执行 log(*numberOfElements*)次比较操作。

```
Void
pop_heap (RandomAccesIterator beg, RandomAccesIterator end)

Void
pop_heap (RandomAccesIterator beg, RandomAccesIterator end,
          BinaryPredicate op)
```

- 以上两种形式都将 heap [*beg*, *end*) 内的最高元素，也就是第一个元素，移到最后位置，并将剩余区间 [beg, end-1) 内的元素组织起来，成为一个新的 heap。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
  op(elem1,elem2)
- 调用者必须保证，进入函数时，区间 [*beg*, *end*) 内的元素原本便已形成一个 heap（在相同的排序准则下）。
- 复杂度：对数，最多执行 2*log(*numberOfElements*) 次比较操作。

```
void
sort_heap (RandomAccesIterator beg, RandomAccesIterator end)

void
sort_heap (RandomAccesIterator beg, RandomAccesIterator end,
           BinaryPredicate op)
```

- 以上两种形式都可以将 heap [*beg*, *end*) 转换为一个已序（*sorted*）序列。
- *op* 是个可有可无的二元判断式，被视为排序准则：
  op(elem1,elem2)
- 注意，此算法一旦结束，该区间就不再是个 heap 了。
- 调用者必须保证，进入函数时，区间 [*beg*, *end*) 内的元素原本便已形成一个 heap（在相同的排序准则下）。
- 复杂度：n-log-n，最多执行 *numberOfElements* * log(*numberOfElements*) 次比较动作。

heap 算法使用范例

以下程序示范如何使用各种 heap 算法：

```
// algo/heap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,5,9);
    INSERT_ELEMENTS(coll,1,4);

    PRINT_ELEMENTS (coll, "on entry: ");

    // convert collection into a heap
    make_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after make_heap(): ");

    // pop next element out of the heap
    pop_heap (coll.begin(), coll.end());
    coll.pop_back();

    PRINT_ELEMENTS (coll, "after pop_heap(): ");

    // push new element into the heap
    coll.push_back (17);
    push_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after push_heap(): ");

    /* convert heap into a sorted collection
    * - NOTE: after the call it is no longer a heap
    */
    sort_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after sort_heap(): ");
}
```

程序输出如下：

```
on entry: 3 4 5 6 7 5 6 7 8 9 1 2 3 4
after make_heap(): 9 8 6 7 7 5 5 3 6 4 1 2 3 4
after pop_heap(): 8 7 6 7 4 5 5 3 6 4 1 2 3
after push_heap(): 17 7 8 7 4 5 6 3 6 4 1 2 3 5
after sort_heap(): 1 2 3 3 4 4 5 5 6 6 7 7 8 17
```

调用 make_heap()之后，元素被排序为 heap：

```
9 8 6 7 7 5 5 3 6 4 1 2 3 4
```

如果你把这些元素转换为二叉树结构，你会发现每个节点的值都小于或等于其父节点值（图 9.1）。push_heap()和 pop_heap()虽然会替换元素，但二叉树结构的恒常性质（亦即：每个节点不大于其父节点）不变。

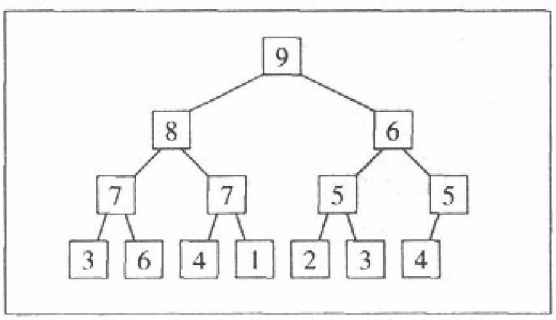


图 9.1 所谓 Heap，其元素形成一个二叉树（Binary Tree）

9.10 已序区间算法（Sorted Range Algorithms）

针对已序区间执行的算法，执行前提是源区间必须在某个排序准则下已序（*sorted*）。较之其对应兄弟（无序区间），它们有着明显的性能优势（通常是对数复杂度，而不再是线性复杂度）。即使迭代器并非随机存取型，也可以使用这些算法——不过如此一来这些算法的复杂度会降为线性，因为迭代器只能一步一步移动。但其比较次数的复杂度仍是对数型。

根据 C++ *Standard*，对着“无序序列”调用这些算法，会导致未定义的行为。然而大部分实作版本中，这些算法对于无序序列仍然有效。只不过，如果你倚仗这个事实，你的程序将不具移植性。

对应于此处所给的算法，关联式容器提供了对应的成员函数。如果要搜寻某个特定的 *key* 或 *value*，你应该使用那些成员函数。

9.10.1 搜寻元素（Searching）

下列算法在已序（*sorted*）区间中搜寻某元素。

检查某个元素是否存在

```
bool
binary_search (ForwardIterator beg, ForwardIterator end,
               const T& value)
bool
binary_search (ForwardIterator beg, ForwardIterator end,
               const T& value,
               BinaryPredicate op)
```

- 两种形式都用来判断已序区间[*beg*, *end*)中是否包含“和 *value* 等值”的元素。
- *op* 是一个可有可无的（可选的）二元判断式，用来作为排序准则：
  op(*elem1*, *elem2*)
- 如果想要获得被搜寻元素的位置，应使用 lower_bound()、upper_bound()或 equal_range()（参见 p413 和 p415）。
- 调用者必须确保进入算法之际，该区间已序（在指定的排序准则作用下）。
- 复杂度：如果搭配随机存取迭代器，则为对数复杂度，否则为线性复杂度（这些算法最多执行 log(*numberOfElements*)+2 次比较操作，但若不是随机存取迭代器，迭代器在元素身上移动的复杂度是线性，于是整体复杂度就是线性了）。

以下示范 binary_search()的用法：

```
// algo/bsearch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // check existence of element with value 5
    if (binary_search(coll.begin(), coll.end(), 5)) {
        cout << "5 is present" << endl;
    }
    else {
        cout << "5 is not present" << endl;
    }
```

```
    // check existence of element with value 42
    if (binary_search(coll.begin(), coll.end(), 42)) {
       cout << "42 is present" << endl;
    }
    else {
       cout << "42 is not present" << endl;
    }
}
```

程序输出如下：

```
1 2 3 4 5 6 7 8 9
5 is present
42 is not present
```

检查若干个值是否存在

```
bool
includes (InputIterator1 beg,
          InputIterator1 end,
          InputIterator2 searchBeg,
          InputIterator2 searchEnd)

bool
includes (InputIterator1 beg,
          InputIterator1 end,
          InputIterator2 searchBeg,
          InputIterator2 searchEnd,
          BinaryPredicate op)
```

- 两种形式都用来判断已序区间 [*beg*, *end*) 是否包含另一个已序区间 [*searchBeg*, *searchEnd*) 的全部元素。也就是说对于 [*searchBeg*, *searchEnd*) 中的每一个元素，如果 [*beg*, *end*) 必有一个对应的相等元素，那么 [*searchBeg*, *searchEnd*) 肯定是 [*beg*, *end*) 的子集。
- *op* 是一个可有可无的（可选的）二元判断式，被用来作为排序准则：
  op(*elem1*, *elem2*)
- 调用者必须确保在进入算法之际，两区间都应该已经按照相同的排序准则排好序了。
- 复杂度：线性，最多执行 2 * (*numberOfElements*+*searchElements*) - 1 次比较操作。

以下程序示范 includes()的用法：

```
// algo/includes.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;
    vector<int> search;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    search.push_back(3);
    search.push_back(4);
    search.push_back(7);
    PRINT_ELEMENTS(search,"search: ");

    // check whether all elements in search are also in coll
    if (includes (coll.begin(), coll.end(),
                  search.begin(), search.end())) {
        cout << "all elements of search are also in coll"
             << endl;
    }
    else {
        cout << "not all elements of search are also in coll"
             << endl;
    }
}
```

程序输出如下：

```
coll: 1 2 3 4 5 6 7 8 9
search: 3 4 7
all elements of search are also in coll
```

搜寻第一个或最后一个可能位置

```
ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- lower_bound()返回第一个“大于等于 *value*”的元素位置。这是可插入“元素值为 *value*”且“不破坏区间[*beg*, *end*)已序性”的第一个位置。
- lower_bound()返回第一个“大于 *value*”的元素位置。这是可插入“元素值为 *value*”且“不破坏区间[*beg*, *end*)已序性”的最后一个位置。
- 如果不存在“其值为 *value*”的元素，上述所有算法都返回 *end*。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
  op(*elem1*, *elem2*)
- 调用者必须确保进入算法之际，所有区间都已按照排序准则排好序了。
- 如要同时获得 lower_bound()和 upper_bound()的结果，请使用 equal_range()（稍后介绍）。
- 关联式容器（set, multiset, map, multimap）分别提供等效成员函数，性能更佳。
- 复杂度：如果搭配随机存取迭代器，则为对数复杂度，否则为线性复杂度（这些算法最多执行 log(*numberOfElements*)+1 次比较动作，但若不是随机存取迭代器，迭代器在元素身上移动的复杂度是线性的，于是整体复杂度就是线性的了）。

以下程序示范 lower_bound()和 upper_bound()[8]的用法：

```
// algo/bounds1.cpp

#include "algostuff.hpp"
using namespace std;
```

[8] 早期的 STL 版本可能需要含入 distance.hpp 文件，参见 p263。

```
int main()
{
  list<int> coll;

  INSERT_ELEMENTS(coll,1,9);
  INSERT_ELEMENTS(coll,1,9);
  coll.sort ();
  PRINT_ELEMENTS(coll);

  // print first and last position 5 could get inserted
  list<int>::iterator pos1, pos2;

  pos1 = lower_bound (coll.begin(), coll.end(),
                      5);

  pos2 = upper_bound (coll.begin(), coll.end(),
                      5);

  cout << "5 could get position "
       << distance(coll.begin(),pos1) + 1
       << " up to "
       << distance(coll.begin(),pos2) + 1
       << " without breaking the sorting" << endl;

  // insert 3 at the first possible position without
  // breaking the sorting
  coll.insert (lower_bound(coll.begin(),coll.end(),
                           3),
               3);

  // insert 7 at the last possible position without
  // breaking the sorting
  coll.insert (upper_bound(coll.begin(),coll.end(),
                           7),
               7);

  PRINT_ELEMENTS(coll);
}
```

程序输出如下：

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9
```

搜寻第一个和最后一个可能位置

```
pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value)

pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- 两种形式都返回“与 *value* 相等”的元素所形成的区间。在此区间内插入“其值为 *value*”的元素，并不会破坏区间 [*beg*, *end*) 的已序性。
- 和下式等效：
  ```
  make_pair(lower_bound(...),upper_bound(...))
  ```
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
  ```
  op(elem1,elem2)
  ```
- 调用者必须确保在进入算法之际，区间已按照排序准则排好序了。
- 关联式容器（set, multiset, map, multimap）都提供有等效成员函数，性能更佳。
- 复杂度：如果搭配随机存取迭代器，则为对数复杂度，否则为线性复杂度（这些算法最多执行 2*log(*numberOfElements*) +1 次比较动作，但若不是随机存取迭代器，迭代器在元素身上移动的复杂度是线性的，于是整体复杂度就是线性的了）。

以下程序展示 equal_range()[9]的用法：

```
// algo/eqrange1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  list<int> coll;

  INSERT_ELEMENTS(coll,1,9);
  INSERT_ELEMENTS(coll,1,9);
  coll.sort ();
  PRINT_ELEMENTS(coll);

  // print first and last position 5 could get inserted
  pair<list<int>::iterator,list<int>::iterator> range;
```

[9] 早期的 STL 版本可能需要 distance.hpp 文件，参见 p263。

```
    range = equal_range (coll.begin(), coll.end(),
                         5);
    cout << "5 could get position "
         << distance(coll.begin(),range.first) + 1
         << " up to "
         << distance(coll.begin(),range.second) + 1
         << " without breaking the sorting" << endl;
}
```

程序输出如下：

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
```

9.10.2 合并元素（Merging）

本节的算法用来将两个区间的元素合并，包括总和（sum）、并集（union）、交集（intersection）等处理。

两个已序集合的总和（Sum）

```
OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg)

OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg, BinaryPredicate op)
```

- 两者都是将源区间 [*source1Beg*, *source1End*) 和 [*source2Beg*, *source2End*) 内的元素合并，使得“以 *destBeg* 起始的目标区间”内含两个源区间的所有元素。假设你对下面两个序列调用 `merge()`：

  1 2 2 4 6 7 7 9

  和

  2 2 2 3 6 6 8 9

  结果将会是：

  1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9

- 目标区间内的所有元素都将按顺序排列。

- 两者都返回目标区间内“最后一个被复制元素”的下一位置（也就是第一个未被覆盖的元素位置）。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
  op(elem1,elem2)
- 源区间没有任何变化。
- 根据标准，调用者应当确保两个源区间一开始都已序。然而在大部分实作版本中，上述算法可以将两个无序的源区间内的元素合并到一个无序的目标区间中。不过如果考虑移植性，这种情况下你应该调用 `copy()` 两次，而不是使用 `merge()`。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重复。
- lists 提供了一个特殊成员函数 `merge()`，用来合并两个 lists（参见 p246）。
- 如果你要确保“两个源区间中都存在的元素”在目标区间中只出现一次，请使用 `set_union()`（参见 p418）。
- 如果你只想获得“同时存在于两个源区间内”的所有元素，请使用 `set_intersection()`（参见 p419）。
- 复杂度：线性，最多执行 *numberOfElements1* + *numberOfElements2* - 1 次比较。

下面这个例子展示 `merge()` 的用法：

```
// algo/merge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll1;
    set<int> coll2;

    // fill both collections with some sorted elements
    INSERT_ELEMENTS(coll1,1,6);
    INSERT_ELEMENTS(coll2,3,8);

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // print merged sequence
    cout << "merged: ";
    merge (coll1.begin(), coll1.end(),
           coll2.begin(), coll2.end(),
           ostream_iterator<int>(cout," "));
```

```
    cout << endl;
}
```

程序输出如下：

```
coll1: 1 2 3 4 5 6
coll2: 3 4 5 6 7 8
merged: 1 2 3 3 4 4 5 5 6 6 7 8
```

p421 另有一个例子，展示“用以合并已序序列”的各种算法之间的不同。

两个已序集合的并集（Union）

```
OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg)

OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg, BinaryPredicate op)
```

- 以上两者都是将已序的源区间 [*source1Beg*, *source1End*) 和 [*source2Beg*, *source2End*) 内的元素合并，得到“以 *destBeg* 起始”的目标区间——这个区间内含的元素要不来自第一源区间，要不就来自第二源区间，或是同时来自两个源区间。例如，对下面两个已序序列调用 set_union()：

  1 2 2 4 6 7 7 9

  和

  2 2 2 3 6 6 8 9

  结果是

  1 2 2 2 3 4 6 6 7 7 8 9
- 目标区间内的所有元素都按顺序排列。
- 同时出现于两个源区间内的元素，在并集区间中将只出现一次。不过如果原来的某个源区间内原本就存在重复元素，则目标区间内也会有重复元素——重复的个数是两个源区间内的重复个数的较大值（译注：可参考《STL 源码剖析》6.5.1 节实例图解）。
- 两者都返回目标区间内“最后一个被复制元素”的下一位置（也就是第一个未被覆盖的元素位置）。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：

  op(elem1, elem2)

- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（*sorted*）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重迭。
- 若想得到两个源区间的全部元素，请用 merge()（参见 p416）。
- 复杂度：线性，最多执行 2*(*numberOfElements1*+*numberOfElements2*) -1 次比较操作。

p421 有一个 set_union()使用范例。该例也展示各种已序序列合并算法的区别。

两个已序集合的交集（Intersection）

```
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg)
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg,
                  BinaryPredicate op)
```

- 以上两者都是将已序源区间 [*source1Beg*, *source1End*) 和 [*source2Beg*, *source2End*) 的元素合并，得到“以 *destBeg* 起始”的目标区间——这个区间内含的元素不但存在于第一源区间，也存在于第二源区间。例如，对下面两个已序序列调用 set_intersection()：

  1 2 2 4 6 7 7 9

  和

  2 2 2 3 6 6 8 9

  结果是：

  2 2 6 9
- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是两个源区间内的重复个数的较小值（译注：可参考《STL 源码剖析》6.5.2 节实例图解）。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- *op* 是个可有可无的二元判断式，被当做排序准则：

  op(*elem1*, *elem2*)

- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（*sorted*）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。

- 目标区间和源区间不得重叠。
- 复杂度：线性，最多执行 2*(*numberOfElements1*+*numberOfElements2*) -1 次比较动作。

p421 有一个 set_intersection() 使用范例。该例也展示各种已序序列合并算法之间的区别。

两个已序集合的差集（Difference）

```
OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg)
OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg,
                BinaryPredicate op)
```

- 以上两者都是将已序源区间 [*source1Beg*,*source1End*) 和 [*source2Beg*, *source2End*) 的元素合并，得到“以 *destBeg* 起始”的目标区间——这个区间内含的元素只存在于第一源区间，不存在于第二源区间。例如，对下面两个已序序列调用 set_difference()：

  1 2 2 4 6 7 7 9

  和

  2 2 2 3 6 6 8 9

  结果是

  1 4 7 7
- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是第一源区间内的重复个数减去第二源区间内的相应重复个数，如果第二源区间内的重复个数大于第一源区间内的相应重复个数，目标区间内的对应重复个数将会是零（译注：可参考《STL 源码剖析》6.5.3 节实例图解）。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：
  op(*elem1*,*elem2*)
- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（*sorted*）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重迭。
- 复杂度：线性，最多执行 2*(*numberOfElements1*+*numberOfElements2*) -1 次比较。

p421 上有一个 set_difference() 使用范例。该例也展示各种已序序列合并算法之间的区别。

```
OutputIterator
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg)

OutputIterator
set_symmetric_difference (InputIterator source1Beg, InputIterator source1End,
                          InputIterator source2Beg, InputIterator source2End,
                          OutputIterator destBeg,
                          BinaryPredicate op)
```

- 两者都是将已序源区间 [*source1Beg*, *source1End*) 和 [*source2Beg*, *source2End*) 的元素合并，得到“以 *destBeg* 起始”的目标区间——这个区间内含的元素或存在于第一源区间，或存在于第二源区间，但不同时存在于两源区间内。例如，对下面两个已序序列调用 set_symmetric_difference()：

  1 2 2 4 6 7 7 9

  和

  2 2 2 3 6 6 8 9

  结果是

  1 2 3 4 6 7 7 8
- 目标区间内的所有元素都按顺序排列。
- 如果某个源区间内原就存在有重复元素，则目标区间内也会有重复元素——重复的个数是两个源区间内的对应重复元素的个数差额（译注：可参考《STL 源码剖析》6.5.4 节实例图解）。
- 两者都返回目标区间内“最后一个被合并元素”的下一位置。
- *op* 是个可有可无的（可选的）二元判断式，被当做排序准则：

  op(*elem1*, *elem2*)
- 源区间没有任何变化。
- 调用者应当确保两个源区间一开始都已序（sorted）。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- 目标区间和源区间不得重叠。
- 复杂度：线性，最多执行 2*(*numberOfElements1*+*numberOfElements2*) -1 次比较动作。

下一小节有个 set_symmetric_difference() 使用范例。该例也展示各种已序序列合并算法之间的区别。

“合并算法”的综合范例

下面这个例子对各个已序序列合并算法做了比较，展示其用法和区别：

```
// algo/setalgos.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    int c1[] = { 1, 2, 2, 4, 6, 7, 7, 9 };
    int num1 = sizeof(c1) / sizeof(int);

    int c2[] = { 2, 2, 2, 3, 6, 6, 8, 9 };
    int num2 = sizeof(c2) / sizeof(int);

    // print source ranges
    cout << "c1: " ;
    copy (c1, c1+num1,
          ostream_iterator<int>(cout," "));
    cout << endl;
    cout << "c2: " ;
    copy (c2, c2+num2,
          ostream_iterator<int>(cout," "));
    cout << '\n' << endl;

    // sum the ranges by using merge()
    cout << "merge(): ";
    merge (c1, c1+num1,
          c2, c2+num2,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // unite the ranges by using set_union()
    cout << "set_union(): ";
    set_union (c1, c1+num1,
               c2, c2+num2,
               ostream_iterator<int>(cout," "));
    cout << endl;

    // intersect the ranges by using set_intersection()
    cout << "set_intersection(): ";
    set_intersection (c1, c1+num1,
                      c2, c2+num2,
                      ostream_iterator<int>(cout," "));
    cout << endl;
```

```
  // determine elements of first range without elements of second range
  // by using set_difference()
  cout << "set_difference(): ";
  set_difference (c1, c1+num1,
                  c2, c2+num2,
                  ostream_iterator<int>(cout," "));
  cout << endl;

  // determine difference the ranges with
  set_symmetric_difference()
  cout << "set_symmetric_difference(): ";
  set_symmetric_difference (c1, c1+num1,
                            c2, c2+num2,
                               ostream_iterator<int>(cout," "));
     cout << endl;
  }
```

程序输出如下：

```
c1:                              1 2 2 4 6 7 7 9
c2:                              2 2 2 3 6 6 8 9

merge():                         1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
set_union():                     1 2 2 2 3 4 6 6 7 7 8 9
set_intersection():              2 2 6 9
set_difference():                1 4 7 7
set_symmetric_difference():      1 2 3 4 6 7 7 8
```

将连贯的（consecutive）已序区间合并

```
void
inplace_merge (BidirectionalIterator beg1,
               BidirectionalIterator end1beg2,
               BidirectionalIterator end2)

void
inplace_merge (BidirectionalIterator beg1,
               BidirectionalIterator end1beg2,
               BidirectionalIterator end2, BinaryPredicate op)
```

- 两者都是将已序源区间[*beg1*, *end1beg2*)和[*end1beg2*, *end2*)的元素合并，使区间[*beg1*, *end2*)成为两者之总和（且形成已序）。
- 复杂度：如有足够的内存则为线性，执行 *numberOfElements*-1 次比较操作，否则为 n-log-n，执行 *numberOfElements* * log(*numberOfElements*) 次比较操作。

以下程序示范 inplace_merge()的用法：

```
// algo/imerge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert two sorted sequences
    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,8);
    PRINT_ELEMENTS(coll);

    // find beginning of second part (element after 7)
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(), // range
                7);                       // value
    ++pos;

    // merge into one sorted range
    inplace_merge (coll.begin(), pos, coll.end());

    PRINT_ELEMENTS(coll);
}
```

程序输出如下：

```
1 2 3 4 5 6 7 1 2 3 4 5 6 7 8
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8
```

9.11 数值算法（Numeric Algorithms）

本节叙述用于数值处理的 STL 算法。当然你也可以用它们来处理非数值元素。例如你可以用 accumulate()求数个字符串和。运用数值算法之前必须先含入头文件 <numeric>[10]：

```
#include <numeric>
```

9.11.1 加工运算后产生结果

对序列进行某种运算

```
T
accumulate (InputIterator beg, InputIterator end,
            T initValue)

T
accumulate (InputIterator beg, InputIterator end,
            T initValue, BinaryFunc op)
```

- 以上第一种形式计算 initValue 和区间[*beg*, *end*)内的所有元素的总和，更具体地说，它针对每一个元素调用以下表达式：
  initValue = *initValue* + *elem*
- 以上第二种形式计算 initValue 和区间[*beg*, *end*)内每一个元素进行 *op* 运算的结果，更具体地说，它针对每一个元素调用以下表达式：
  initValue = *op*(*initValue*, *elem*)
- 因此，对于以下数值序列：
  a1 a2 a3 a4 ...
  上述两个算法分别计算
  initValue + a1 + a2 + a3 + ...
  和
  initValue op a1 *op* a2 *op* a3 *op* ...
- 如果序列为空（beg==end），则两者都返回 *initValue*。
- 复杂度：线性，上述两式分别调用 operator+ 或 *op*() 各 ***numberOfElements*** 次。

下面这个例子展示如何使用 accumulate()得到区间内所有元素的加总和乘积：

---

[10] 早期的 STL 中，数值算法被定义于 <algo.h>。

```
// algo/accu1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  vector<int> coll;

  INSERT_ELEMENTS(coll,1,9);
  PRINT_ELEMENTS(coll);

  // process sum of elements
  cout << "sum: "
       << accumulate (coll.begin(), coll.end(), // range
                      0)                        // initial value
       << endl;

  // process sum of elements less 100
  cout << "sum: "
       << accumulate (coll.begin(), coll.end(), // range
                      -100)                     // initial value
       << endl;

  // process product of elements
  cout << "product: "
       << accumulate (coll.begin(), coll.end(), // range
                      1,                        // initial value
                      multiplies<int>())        // operation
       << endl;

  // process product of elements (use 0 as initial value)
  cout << "product: "
       << accumulate (coll.begin(), coll.end(), // range
                      0,                        // initial value
                      multiplies<int>())        // operation
       << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
product: 0
```

最后一个输出结果是 0，因为任何数乘以 0 都得 0。

计算两序列的内积（Inner Product）

```
T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue)

T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue,
               BinaryFunc op1, BinaryFunc op2)
```

- 以上第一种形式计算并返回 [beg,end) 区间和“以 beg2 为起始的区间”的对应元素组（再加上 initValue）的内积。具体地说也就是针对“两区间内的每一组对应元素”调用以下表达式：

  ```
  initValue = initValue + elem1 * elem2
  ```

- 以上第二种形式将[beg,end)区间和“以 beg2 为起始的区间”内的对应元素组进行 op2 运算，然后再和 initValue 进行 op1 运算，并将结果返回。具体地说也就是针对“两区间内的每一组对应元素”调用以下表达式：

  ```
  initValue = op1(initValue,op2(elem1,elem2))
  ```

- 所以，对于数值序列：

  ```
  a1 a2 a3 ...
  b1 b2 b3 ...
  ```

  上述两个算法分别计算并返回：

  ```
  initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + ...
  ```

  和

  ```
  initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 ...
  ```

- 如果第一区间为空（beg1==end1），则两者都返回 initValue。
- 调用者必须确保“以 beg2 为起始的区间”内含足够元素空间。
- op1 和 op2 都不得变动（修改）其参数内容。
- 复杂度：线性，调用 operator+ 和 operator* 各 *numberOfElement* 次，或是调用 op1() 和 op2() 各 *numberOfElement* 次。

以下程序示范 inner_product()的用法。它计算两个序列的乘积总和（sum of product），以及总和乘积（product of sum）：

```
// algo/inner1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
  list<int> coll;

  INSERT_ELEMENTS(coll,1,6);
  PRINT_ELEMENTS(coll);

  /* process sum of all products
  * (0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
  */
  cout << "inner product: "
       << inner_product (coll.begin(), coll.end(), // first range
                         coll.begin(),        // second range
                         0)                   // initial value
       << endl;

  /* process sum of 1*6 ... 6*1
  * (0 + 1*6 + 2*5 + 3*4 + 4*3 + 5*2 + 6*1)
  */
  cout << "inner reverse product: "
       << inner_product (coll.begin(), coll.end(), // first range
                         coll.rbegin(),       // second range
                         0)                   // initial value
       << endl;

  /* process product of all sums
  * (1 * 1+1 * 2+2 * 3+3 * 4+4 * 5+5 * 6+6)
  */
  cout << "product of sums: "
       << inner_product (coll.begin(), coll.end(), // first range
                         coll.begin(),        // second range
                         1,                   // initial value
                         multiplies<int>(),   // inner operation
                         plus<int>())         // outer operation
       << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6
inner product: 91
inner reverse product: 56
product of sums: 46080
```

9.11.2 相对值和绝对值之间的转换

下面两个算法可以在相对值序列和绝对值序列之间相互转换。

将相对值转换成绝对值

```
OutputIterator
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEnd,
             OutputIterator destBeg, BinaryFunc op)
```

- 第一形式计算源区间[*sourceBeg*, *sourceEnd*)中每个元素的部分和，然后将结果写入以 *destBeg* 为起点的目标区间。
- 第二形式将源区间[*sourceBeg*, *sourceEnd*)中的每个元素和其先前所有元素进行 *op* 运算，并将结果写入以 *destBeg* 为起点的目标区间。
- 因此，对于以下数值序列：

  ```
  a1 a2 a3 ...
  ```

  它们分别计算：

  ```
  a1, a1 + a2, a1 + a2 + a3, ...
  ```

  和

  ```
  a1, a1 op a2, a1 op a2 op a3, ...
  ```

- 两种形式都返回目标区间内“最后一个被写入的值”的下一位置（也就是第一个未被覆盖的元素的位置）。

- 第一形式相当于把一个相对值序列转换为一个绝对值序列。就此而言，partial_sum()正好和 adjacent_difference()互补。
- 源区间和目标区间可以相同。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- *op* 不得变动（更改）传入的参数。
- 复杂度：线性，分别调用 operator+ 或 *op*() ***numberOfElements*** 次。

以下程序示范 partial_sum()的用法：

```
// algo/partsum1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    // print all partial sums
    partial_sum (coll.begin(), coll.end(),          // source range
                 ostream_iterator<int>(cout," ")); // destination
    cout << endl;

    // print all partial products
    partial_sum (coll.begin(), coll.end(),          // source range
                 ostream_iterator<int>(cout," "),  // destination
                 multiplies<int>());               // operation
    cout << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6
1 3 6 10 15 21
1 2 6 24 120 720
```

关于相对值和绝对值之间的互换，请参见 p432 的例子。

将绝对值转换成相对值

```
OutputIterator
adjacent_difference (InputIterator sourceBeg,
                     InputIterator sourceEnd,
                     OutputIterator destBeg)

OutputIterator
adjacent_difference (InputIterator sourceBeg,
                     InputIterator sourceEnd,
                     OutputIterator destBeg, BinaryFunc op)
```

- 第一种形式计算区间[*sourceBeg*, *sourceEnd*)中每一个元素和其紧邻前趋元素的差额，并将结果写入以 *destBeg* 为起点的目标区间。
- 第二种形式针对区间[*sourceBeg*, *sourceEnd*)中的每一个元素和其紧邻前趋元素调用 *op* 操作，并将结果写入以 *destBeg* 为起点的目标区间。
- 第一个元素只是被很单纯地加以复制。
- 因此，对于以下数值序列：

  ```
  a1 a2 a3 a4 ...
  ```

  它们分别计算：

  ```
  a1, a2 - a1, a3 - a2, a4 - a3, ...
  ```

  和：

  ```
  a1, a2 op a1, a3 op a2, a4 op a3, ...
  ```

- 两种形式都返回目标区间内“最后一个被写入的值”的下一位置（也就是第一个未被覆盖的元素的位置）。
- 第一形式相当于把一个绝对值序列转换为一个相对值序列。就此而言，adjacent_difference()正好与patial_sum()互补。
- 源区间和目标区间可以相同。
- 调用者必须确保目标区间够大，要不就得使用插入型迭代器。
- *op* 不得变动（更改）传入的参数。
- 复杂度：线性，分别调用 operator- 或 *op*() ***numberOfElements***-1 次。

以下程序示范 adjacent_difference()的用法：

```
// algo/adjdiff1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;
```

```
    INSERT_ELEMENTS(coll,1,6);
    PRINT_ELEMENTS(coll);

    // print all differences between elements
    adjacent_difference (coll.begin(), coll.end(), // source
                         ostream_iterator<int>(cout," ")); // dest.
    cout << endl;

    // print all sums with the predecessors
    adjacent_difference (coll.begin(), coll.end(), // source
                         ostream_iterator<int>(cout," "), // dest.
                         plus<int>());              // operation
    cout << endl;

    // print all products between elements
    adjacent_difference (coll.begin(), coll.end(), // source
                         ostream_iterator<int>(cout," "), // dest.
                         multiplies<int>());        // operation
    cout << endl;
}
```

程序输出如下：

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

参见下一个例子，其中将绝对值和相对值互相转换。

相对值转换为绝对值，实例解说

下面这个例子展示如何运用 partial_sum()和 adjacent_difference()将一个相对值序列转化为一个绝对值序列，以及如何逆向实施：

```
// algo/relabs.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
```

```
    vector<int> coll;

    coll.push_back(17);
    coll.push_back(-3);
    coll.push_back(22);
    coll.push_back(13);
    coll.push_back(13);
    coll.push_back(-9);
    PRINT_ELEMENTS(coll,"coll: ");

    // convert into relative values
    adjacent_difference (coll.begin(), coll.end(), // source
                         coll.begin());            // destination
    PRINT_ELEMENTS(coll,"relative: ");

    // convert into absolute values
    partial_sum (coll.begin(), coll.end(),         // source
                 coll.begin());                    // destination
    PRINT_ELEMENTS(coll,"absolute: ");
}
```

程序输出如下：

```
coll: 17 -3 22 13 13 -9
relative: 17 -20 25 -9 0 -22
absolute: 17 -3 22 13 13 -9
```