6.2 容器（Container）

容器用来管理一大群元素。为了适应不同需要，STL 提供了不同的容器，如图6.2所示。

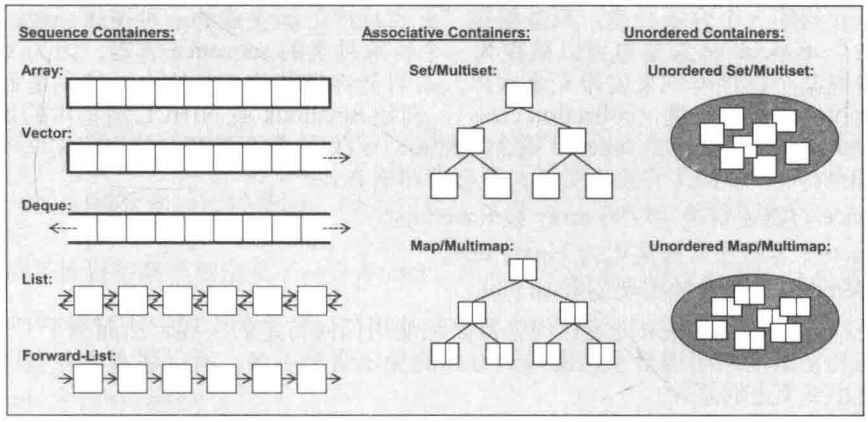

图 6.2　STL 的容器种类

总的来说，容器可分为三大类：

1. **序列式容器（Sequence container）**，这是一种有序（*ordered*）集合，其内每个元素均有确凿的位置——取决于插入时机和地点，与元素值无关。如果你以追加方式对一个集合置入 6 个元素，它们的排列次序将和置入次序一致。STL 提供了 5 个定义好的序列式容器：`array`、`vector`、`deque`、`list` 和 `forward_list`。[1]
2. **关联式容器（Associative container）**，这是一种已排序（*sorted*）集合，元素位置取决于其 value（或 key——如果元素是个 key/value pair）和给定的某个排序准则。如果将 6 个元素置入这样的集合中，它们的值将决定它们的次序，和插入次序无关。STL 提供了 4 个关联式容器：`set`、`multiset`、`map` 和 `multimap`。
3. **无序容器（Unordered (associative) container）**，这是一种无序集合（unordered collection），其内每个元素的位置无关紧要，唯一重要的是某特定元素是否位于此集合内。元素值或其安插顺序，都不影响

[1] Class `array` 自 TR1 才新加入；`forward_list` 自 C++11 才新加入。

元素的位置，而且元素的位置有可能在容器生命中被改变。如果你放 6 个元素到这种集合内，它们的次序不明确，并且可能随时间而改变。 STL 内含 4 个预定义的无序容器：unordered_set、unordered_multiset、unordered_map 和 unordered_multimap。

6.2.6 容器适配器（Container Adapter）

除了以上数个根本的容器类，为满足特殊需求，C++ 标准库还提供了一些所谓的容器适配器，它们也是预定义的容器，提供的是一定限度的接口，用以应付特殊需求。这些容器适配器都是根据基本容器实现而成，包括：

- **Stack** 名字足以说明一切。Stack 容器对元素采取 LIFO（后进先出）管理策略。
- **Queue** 对元素采取 FIFO（先进先出）管理策略。也就是说，它是个寻常的缓冲区（buffer）。
- **Priority queue** 其内的元素拥有各种优先权。所谓优先权乃是基于程序员提供的排序准则（默认为操作符 <）而定义。这种特殊容器的效果相当于这样一个缓冲区："下一元素永远是容器中优先权最高的元素"。如果同时有多个元素具备最高优先权，则其次序无明确定义。

从历史观点来看，容器适配器是 STL 的一部分。从程序员的观点来看，它们只不过是一种特别的容器类，使用"由 STL 提供的容器、迭代器和算法所形成的总体框架"。因此，容器适配器被我当作 STL 内核的外围，直到第 12 章才介绍。

6.3 迭代器（Iterator）

自 C++11 起，我们可以使用一个 range-based `for` 循环来处理所有元素，然而如果只是要找出某元素，并不需要处理所有元素。我们应该迭代所有元素，直到找到目标。此外，我们或许希望将这个（被找到的元素的）位置存放在某处，以便稍后能够继续迭代或进行其他处理。因此我们需要这样的概念：以一个对象表现出容器元素的位置。这样的概念的确存在。实践这个概念的对象就是所谓的**迭代器**（iterator）。事实上我们将会看到，range-based `for` 循环其实就是此概念的一个便捷接口，也就是说，其内部使用迭代器对象迭代（遍历）所有元素。

迭代器是一个"可遍历 STL 容器全部或部分元素"的对象。迭代器用来表现容器中的某一个位置。基本操作如下：

- **Operator** * 返回当前位置上的元素值。如果该元素拥有成员，你可以通过迭代器直接以操作符 -> 取用它们。
- **Operator ++** 令迭代器前进至下一元素。大多数迭代器还可使用 operator -- 退至前一元素。
- **Operators ==**和 **!=** 判断两个迭代器是否指向同一位置。
- **Operator =** 对迭代器赋值（也就是指明迭代器所指向的元素的位置）。

这些操作和 C/C++ "运用 pointer 操作寻常的 array 元素"时的接口一致。差别在于，迭代器是所谓的 *smart pointer*，具有遍历复杂数据结构的能力，其内部运作机制取决于其所遍历的数据结构。因此，每一种容器都必须提供自己的迭代器。事实上每一种容器的确都将其迭代器以嵌套（nested）方式定义于 class 内部。因此各种迭代器的接口虽然相同，类型却各自不同。

这直接引出了泛型程序设计的概念：所有操作都使用相同接口，纵使类型不同。因此，你可以使用 template 将泛型操作公式化，使之得以顺利运作那些“能够满足接口需求”的任何类型。

所有容器类都提供一些基本的成员函数，使我们得以取得迭代器并以之遍历所有元素。这些函数中最重要的是：

- **begin()** 返回一个迭代器，指向容器起点，也就是第一元素（如果有的话）的位置。
- **end()** 返回一个迭代器，指向容器终点。终点位于最末元素的下一位置，这样的迭代器又称作“逾尾（*past-the-end*）”迭代器。

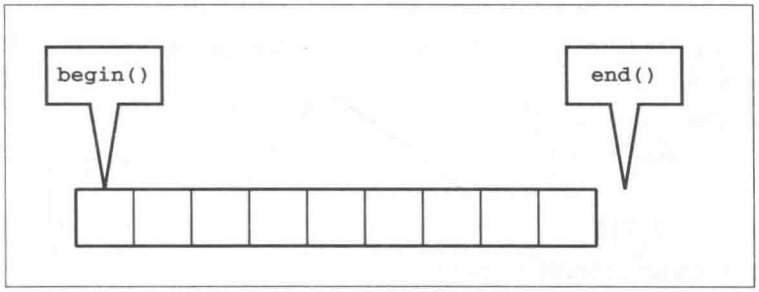

图 6.4 容器的 begin() 和 end()

于是，begin() 和 end() 形成了一个半开区间（half-open range），从第一元素开始，到最末元素的下一位置结束（如图6.4所示）。半开区间有两个优点：

1. 为“遍历元素时的 loop 结束时机”提供一个简单的判断依据。只要尚未到达 end()，loop 就可以继续进行。
2. 不必对空区间（empty range）采取特殊处理手法。空区间的 begin() 就等于 end()。

下面这个例子示范了迭代器的用法，将 list 容器的所有元素打印出来（这是 6.2.1 节第 173 页的 list 实例的变化版本，改用迭代器）。

```
// stl/list1old.cpp

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

int main()
{
    list<char> coll;          // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }
```

```
    // print all elements:
    // - iterate over all elements
    list<char>::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

首先创建一个 list，然后填入字符 'a' 到 'z'，然后打印出所有元素。但这次不是使用 range-base for 循环：

```
for (auto elem : coll) {
    cout << elem << ' ';
}
```

所有元素是被一个寻常的 for 循环打印，使用迭代器走遍容器内的每一个元素：

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

迭代器 pos 被声明于循环之前，其类型是“指向容器内的常量元素”的迭代器：

```
list<char>::const_iterator pos;
```

任何容器都定义有两种迭代器类型：

1. *container*::iterator 以“读/写”模式遍历元素。
2. *container*::const_iterator 以“只读”模式遍历元素。

例如在 class list中，它们的定义可能如下：

```
namespace std {
    template <typename T>
    class list {
      public:
        typedef ... iterator;
        typedef ... const_iterator;
        ...
    };
}
```

至于其中 iterator 和 const_iterator 的确切类型，则由实现（implementation）定义之。

上述循环中，迭代器 pos 以容器的第一元素的位置为初值：

```
pos = coll.begin()
```

循环不断进行，只要 pos 尚未到达容器终点：

```
pos != coll.end()
```

这里的 pos 是与“逾尾（past-the-end）”迭代器做比较。当循环内部执行 ++pos 语句，迭代器 pos 就会前进至下一元素。

总而言之，pos 从第一元素开始，逐次访问每一个元素，直到抵达终点为止（如图6.5所示）。如果容器内没有任何元素，coll.begin() 等于 coll.end()，循环根本不会执行。

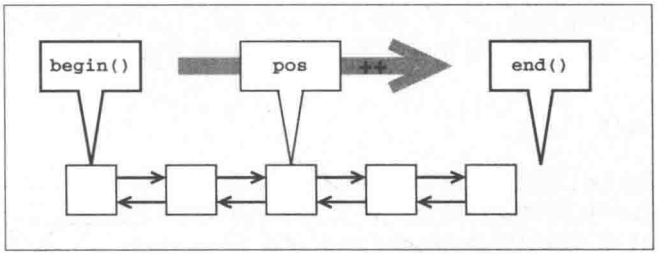

图 6.5 迭代器 pos 遍历 List 的每一个元素

在循环内部，*pos 代表当前（current）元素。本例将它输出至标准输出设备 cout 后，又接着输出了一个空格。你不能改变元素内容，因为 pos 是个 const_iterator，从这样一个迭代器的观点看，元素是常量，不能更改。不过如果你采用非常量（nonconstant）迭代器，而且元素本身的类型也是非常量（nonconstant），就可以通过迭代器来改变元素值。例如：

```
// make all characters in the list uppercase
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    *pos = toupper(*pos);
}
```

如果我们使用迭代器遍历 (unordered) map 和 multimap 的元素，pos 会指向 key/value pair。那么表达式

```
pos->second
```

将取得 key/value pair 的第二成分，也就是元素的 value，而表达式

```
pos->first
```

会取得其 (constant) key。

++pos vs. pos++

注意，这里使用前置式递增（preincrement），因为它比后置式递增（postincrement）效率高。后者内部需要一个临时对象，因为它必须存放迭代器的原本位置并返回之，

所以一般情况下最好使用 ++pos，不要用 pos++。也就是说，你应该避免这么写：

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                              ^^^^^     // OK, but slightly slower
    ...
}
```

这种效率改善几乎总是无关紧要的。所以，不要把这里的推荐解读为“你应该竭尽所能不计代价地做任何事情，只为了如此微小的效率损失”。程序的可读性，以及可维护性，远比效率优化重要。此处的重点是，在本例中你不会因为选用前置式累加而不选用后置式累加而付出代价，那么，当然我们宁可选择前置式累加（和前置式递减）。

cbegin() 和 cend()

自 C++11 开始，我们可以使用关键字 auto（见 3.1.2 节第 14 页）代替迭代器的精确类型（前提是你在迭代器声明期间就初始化，使其类型可以取决于初值）。因此如果我们直接以 begin() 初始化迭代器，就可以使用 auto 声明其类型：

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

如你所见，使用 auto 的优点之一就是，程序比较浓缩精简。如果没有 auto，在循环内声明迭代器的动作应该如下：

```
for (list<char>::const_iterator pos = coll.begin();
     pos != coll.end();
     ++pos) {
    cout << *pos << ' ';
}
```

另一个优点是，采用这种循环写法，万一容器类型有所改变，程序整体仍能保持较佳的强壮性。然而其缺点是，迭代器丧失常量性（constness），可能引发“计划外的赋值”风险。因为

```
auto pos = coll.begin()
```

会使 pos 成为一个非常量迭代器，此乃因为 begin() 返回的是个类型为 *cont*::iterator 的对象。为确保仍可使用常量迭代器，自 C++11 起容器提供 cbegin() 和 cend()，它们返回一个类型为 *cont*::const_iterator 的对象。

现在我来总结改善方案。自 C++11 起，一个允许“迭代容器内所有元素”的循环如果不使用 range-based for 循环，看起来应如下：

```
for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
    ...
}
```

Range-Based for 循环 vs. 迭代器

介绍过迭代器之后，我们可以解释 range-based for 循环的精确行为了。对容器而言，range-based for 循环其实不过是个便捷接口，用来迭代它“所接收到的集合区间”内的每一个元素。在循环体内，真实元素被“当前迭代器所指向的 value”初始化。

因此

```
for (type elem : coll) {
    ...
}
```

被解释为

```
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
    type elem = *pos;
    ...
}
```

现在我们可以了解为什么声明 elem 为一个 constant reference 可以避免非必要复制了。如果不那么做，elem 会被初始化为 *pos 的拷贝。详见 3.1.4 节第 17 页。

6.4 算法（Algorithm）

为了处理容器内的元素，STL 提供了一些标准算法，包括查找、排序、拷贝、重新排序、修改、数值运算等基本而普遍的算法。

算法并非容器类的成员函数，而是一种搭配迭代器使用的全局函数。这么做有一个重要优势：所有算法只需实现一份，就可以对所有容器运作，不必为每一种容器量身定制。算法甚至可以操作不同类型（type）之容器内的元素，也可以与用户自定义的容器搭配。这个概念大幅降低了代码量，提高了程序库的能力和弹性。

注意，这里所阐述的并非面向对象思维（OOP paradigm），而是泛型函数编程思维（generic functional programming paradigm）。在面向对象编程（OOP）概念里，数据与操作合为一体，在这里则被明确划分开来，再通过特定的接口彼此互动。当然这需要付出代价：首先是用法有失直观，其次某些数据结构和算法之间并不兼容。更有甚者，某些容器和算法虽然勉强兼容却毫无用处（也许导致很糟的效率）。因此，深入学习 STL 概念并了解其缺陷，显得十分重要，唯其如此方能取其利而避其害。我将在本章剩余篇幅中，通过一些实例来介绍它们。

让我们从 STL 算法的简单运用入手。以下展现了若干算法的使用形式：

```
// stl/algo1.cpp

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

int main()
{
    // create vector with elements from 1 to 6 in arbitrary order
    vector<int> coll = { 2, 5, 4, 1, 6, 3 };

    // find and print minimum and maximum elements
    auto minpos = min_element(coll.cbegin(),coll.cend());
    cout << "min: "  << *minpos << endl;
    auto maxpos = max_element(coll.cbegin(),coll.cend());
    cout << "max: "  << *maxpos << endl;

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

    // find the first element with value 3
    // - no cbegin()/cend() because later we modify the elements pos3 refers to
    auto pos3 = find (coll.begin(), coll.end(),   // range
                      3);                         // value

    // reverse the order of the found element with value 3 and all following elements
    reverse (pos3, coll.end());

    // print all elements
    for (auto elem : coll) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

为了能够调用算法，首先必须包含头文件 <algorithm>（某些算法需要特别的头文件，见 11.1 节第 505 页）：

```
#include <algorithm>
```

最先出现的算法是 min_element() 和 max_element()。调用它们时必须传入两个实参，定义出欲处理的元素范围。如果想处理容器内所有元素，可使用 cbegin() 和 cend()，或 begin() 和 end()。两个算法都返回一个迭代器，分别指向最小或最大元素。因此，语句

```
auto minpos = min_element(coll.cbegin(),coll.cend());
```

中，算法 min_element() 返回最小元素的位置（如果最小元素不止一个，则返回第一个的位置）。以下语句打印出该元素：

```
cout << "min: " << *minpos << endl;
```

当然，你也可以把上述两个动作合并于单一语句：

```
cout << *min_element(coll.cbegin(),coll.cend()) << endl;
```

接下来出现的算法是 sort()。顾名思义，它将“由两个实参指出来”的区间内的所有元素加以排序。你可以（选择性地）传入一个排序准则，默认使用 operator <。因此，本例容器内的所有元素以递增方式排列。

```
sort (coll.begin(), coll.end());
```

排序后的容器元素次序如下：

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

注意，这里不可使用 cbegin() 和 cend()，因为 sort() 会改动元素的 value，但 const_iterator 不允许如此。

再来便是算法 find()。它在给定范围内查找某值。本例在整个容器内寻找数值为 3 的第一个元素。

```
auto pos3 = find (coll.begin(), coll.end(),   // range
                  3);                         // value
```

如果 find() 成功，返回一个迭代器指向目标元素。如果失败，返回第二实参所指示的区间末端，本例是 coll 的逾尾（past-the-end）迭代器。本例在第三个元素位置上发现数值 3，因此完成后 pos3 指向 coll 的第三个元素。

本例所展示的最后一个算法是reverse()，将区间内的元素反转放置：

```
reverse (pos3, coll.end());
```

这个动作会将第三元素至最末元素的排列次序逆转。由于这是一种改动，我们必须使用一个非常量迭代器，这就是为什么我调用 find() 并传入 begin() 和 end()，而不是传入 cbegin() 和 cend()。否则 pos3 就是个 const_iterator，把它传给 reverse() 会导致错误。

程序输出如下：

```
min: 1
max: 6
1 2 6 5 4 3
```

注意，这个例子运用了若干 C++11 特性。如果你手上的平台不支持 C++11 所有特性，下面是一个等效程序：

```
// stl/algo1old.cpp

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

int main()
{
    // create vector with elements from 1 to 6 in arbitrary order
    vector<int> coll;
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // find and print minimum and maximum elements
    vector<int>::const_iterator minpos = min_element(coll.begin(),
                                                     coll.end());
    cout << "min: "  << *minpos << endl;
    vector<int>::const_iterator maxpos = max_element(coll.begin(),
                                                     coll.end());
    cout << "max: "  << *maxpos << endl;

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

    // find the first element with value 3
    vector<int>::iterator pos3;
    pos3 = find (coll.begin(), coll.end(),  // range
                 3);                        // value

    // reverse the order of the found element with value 3 and all following elements
    reverse (pos3, coll.end());

    // print all elements
    vector<int>::const_iterator pos;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

两个版本之间的区别在于：

- 不能使用初值列（initializer list）来为 vector 设初值。
- 不提供成员函数 `cbegin()` 和 `cend()`，你必须代以 `begin()` 和 `end()`，但还是可以使用 `const_iterator`。
- 不能使用 `auto`，你必须明白声明迭代器。
- 不能使用 range-based `for` 循环，你必须以迭代器输出每一个元素。

6.4.1 区间（Range）

所有算法都是用来处理一或多个区间内的元素。这样的区间可以（但非必须）涵盖容器内的全部元素。为了操作容器元素的某个子集，我们必须将区间首尾当作两个实参（argument）传给算法，而不是一口气把整个容器传递进去。

这样的接口灵活又危险。调用者必须确保经由两实参定义出来的区间是有效的（valid）。所谓有效就是，从起点出发，逐一前进，能够到达终点。也就是说，程序员自己必须确保两个迭代器隶属同一容器，而且前后的放置是正确的，否则结果难料，可能引起无穷循环，也可能访问到内存禁区。就此而言，迭代器就像寻常指针一样危险。不过请注意，所谓“结果难料”（或说行为不明确〔undefined behavior〕）意味着任何 STL 实现均可自由选择合适的方式来处理此类错误。稍后你会发现，确保区间有效并不像听起来那么简单。与此相关的一些细节见 6.12 节第 245 页。

所有算法处理的都是半开区间（half-open range）——包括起始元素的位置但不包括末尾元素的位置。传统的数学表示法是：

[*begin*,*end*)

或

[*begin*,*end*[

本书采用第一种形式。

半开区间的优点主要是单纯，可免除对空集做特殊处理（见 6.3 节第 189 页）。当然，金无足赤，世上没有完美的设计。请看下面的例子：

```
// stl/find1.cpp

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

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

```
    // insert elements from 20 to 40
    for (int i=20; i<=40; ++i) {
        coll.push_back(i);
    }

    // find position of element with value 3
    // - there is none, so pos3 gets coll.end()
    auto pos3 = find (coll.begin(), coll.end(),      // range
                      3);                            // value

    // reverse the order of elements between found element and the end
    // - because pos3 is coll.end() it reverses an empty range
    reverse (pos3, coll.end());

    // find positions of values 25 and 35
    list<int>::iterator pos25, pos35;
    pos25 = find (coll.begin(), coll.end(),  // range
                  25);                       // value
    pos35 = find (coll.begin(), coll.end(),  // range
                  35);                       // value

    // print the maximum of the corresponding range
    // - note: including pos25 but excluding pos35
    cout << "max: " << *max_element (pos25, pos35) << endl;

    // process the elements including the last position
    cout << "max: " << *max_element (pos25, ++pos35) << endl;
}
```

本例首先以 20 至 40 的整数作为容器初值。查找元素值 3 的任务失败后，`find()` 返回区间的结束位置（本例为 `coll.end()`）并赋值给 `pos3`。以此作为稍后调用 `reverse()` 时的区间起点，纯粹是空摆架子，因为其结果相当于：

```
reverse (coll.end(), coll.end());
```

这其实就是反转一个空区间，当然毫无效果了（亦即所谓的“no-op”）。

如果使用 `find()` 获取某个子集的第一和最后元素，你必须考虑一点：半开区间并不包含最后一个元素。所以上述例子中首次调用 `max_element()`：

```
max_element (pos25, pos35)
```

找到的是 34，而不是 35：

```
max: 34
```

为了处理最后一个元素，你必须把该元素的下一位置传给算法：

```
max_element (pos25, ++pos35)
```

这样才能得到正确的结果：

```
max: 35
```

注意，本例用的是 list 容器，所以你只能以 ++ 取得 pos35 的下一个位置。如果面对的是 vector 或 deque 的随机访问迭代器（random-access iterator），你可以写 pos35 + 1。这是因为随机访问迭代器允许迭代器算术运算（参见 6.3.2 节第 198 页和 9.2.5 节第 438 页）。

当然，你可以使用 pos25 和 pos35 来查找其间的任何东西。记住，为了让查找动作含 pos35，必须将 pos35 的下一位置传入，例如：

```
// increment pos35 to search with its value included
++pos35;
pos30 = find(pos25,pos35,     // range
             30);             // value
if (pos30 == pos35) {
    cout << "30 is NOT in the subrange" << endl;
}
else {
    cout << "30 is in the subrange" << endl;
}
```

本节的所有例子都可以正常运作的前提是，你知道 pos25 在 pos35 之前。否则 [pos25,pos35) 就不是个有效区间。如果你对于“哪个元素在前，哪个元素在后”心中没谱，事情就麻烦了，说不定会导致不明确行为。

现在假设你并不知道元素 25 和元素 35 的前后关系，甚至连它们是否存在也心存疑虑。如果你手上是随机访问迭代器（random-access iterator），你可以使用 operator < 进行检查：

```
if (pos25 < pos35) {
    // only [pos25,pos35) is valid
    ...
}
else if (pos35 < pos25) {
    // only [pos35,pos25) is valid
    ...
}
else {
    // both are equal, so both must be end()
    ...
}
```

如果你手上并非随机访问迭代器，那还真没什么直截了当的办法可以确定哪个迭代器在前。你只能在“起点和某个迭代器”之间，以及“该迭代器和终点”之间，寻找另外那个迭代器。此时你的解决方法需要一些变化：

不是一口气在整个区间中查找两值，而是试着了解，哪个值先来到。例如：

```
pos25 = find (coll.begin(), coll.end(),     // range
              25);                          // value
pos35 = find (coll.begin(), pos25,          // range
              35);                          // value
if (pos25 != coll.end() && pos35 != pos25) {
    // pos35 is in front of pos25
    // so, only [pos35,pos25) is valid
    ...
}
else {
    pos35 = find (pos25, coll.end(),        // range
                  35);                      // value
    if (pos35 != coll.end()) {
        // pos25 is in front of pos35
        // so, only [pos25,pos35) is valid
        ...
    }
    else {
        // 25 and/or 35 not found
        ...
    }
}
```

和前例不同的是，本例并非在 coll 的整个区间内查找 35，而是先在起点和 pos25 之间寻找，如果一无所获，再在 pos25 之后的区间寻找。其结果当然使你得以完全掌握哪个位置在前面、哪个子区间有效。

这么做并不是很有效率。当然还有其他高招，可以直接找到 25 或 35 首次出现位置。你可以运用 find_if()，传给它一个 lambda（见 3.1.10 节第 28 页）定义出一个准则，评估 coll 内的每一个元素：

```
pos = find_if (coll.begin(), coll.end(),  // range
               [] (int i) {               // criterion
                   return i == 25 || i == 35;
               });
if (pos == coll.end()) {
    // no element with value 25 or 35 found
    ...
}
else if (*pos == 25) {
```

```
    // element with value 25 comes first
    pos25 = pos;
    pos35 = find (++pos, coll.end(),        // range
                  35);                      // value
    ...
}
else {
    // element with value 35 comes first
    pos35 = pos;
    pos25 = find (++pos, coll.end(),        // range
                  25);                      // value
    ...
}
```

在这里，特殊的 lambda 表达式

```
[] (int i) {
    return i == 25 || i == 35;
}
```

被用作一个准则，允许查找第一个“带有 value 25 或 value 35”的元素。 6.9 节第 229 页已介绍过如何在 STL 中使用 lambda， 10.3 节第 499 页有更详细的讨论。

6.4.2 处理多重区间（Multiple Ranges）

有数个算法可以（或说需要）同时处理多重区间。通常你必须设定第一个区间的起点和终点，至于其他区间，只需设定起点即可，终点通常可由第一区间的元素数量推导出来。例如以下程序片段中，equal() 从头开始逐一比较 coll1 和 coll2 的所有元素：

```
if (equal (coll1.begin(), coll1.end(),  // first range
           coll2.begin())) {            // second range
    ...
}
```

于是，coll2 之中参与比较的元素数量，间接取决于 coll1 内的元素数量（如图6.9所示）。

这使我们收获一个重要心得：**如果某个算法用来处理多重区间，那么当你调用它时，务必确保第二（以及其他）区间所拥有的元素个数至少和第一区间内的元素个数相同。**特别是执行涂写动作时，务必确保目标区间（destination range）够大。

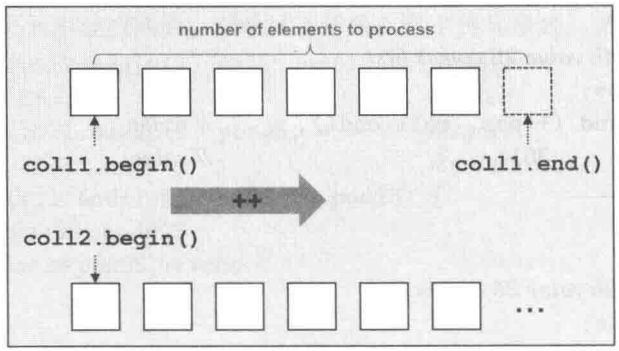

图 6.9 算法在两个区间内迭代（Iterating）

考虑下面这个程序：

```
// stl/copybug.cpp

#include <algorithm>
#include <list>
#include <vector>
using namespace std;

int main()
{
    list<int>   coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    vector<int> coll2;

    // RUNTIME ERROR:
    // - overwrites nonexisting elements in the destination
    copy (coll1.cbegin(), coll1.cend(),    // source
          coll2.begin());                  // destination
    ...
}
```

这里调用了 copy() 算法，将第一区间内的全部元素拷贝至目标区间。如上所述，第一区间的起点和终点都已指定，第二区间只指出起点。然而，由于该算法执行的是覆写动作（overwrite）而非安插动作（insert），所以目标区间必须拥有足够的元素被覆写，否则就会像这个例子一样，导致不明确行为。如果目标区间内没有足够的元素供覆写，通常意味着你会覆写 coll2.end() 之后的任何东西，幸运的话程序立即崩溃——这起码还能让你知道出错了。你可以强制自己获得这种幸运：使用 STL 安全版本。在这样的版本中所有不明确行为都会被导向一个错误处理程序（error-handling procedure）。见 6.12.1 节第 247 页。

想要避免上述错误，你可以 (1) 确认目标区间内有足够的元素空间，或是 (2) 采用 *insert iterator*。Insert iterator 将在 6.5.1 节第 210 页介绍。我首先解释如何修改目标区间使它拥有足够空间。

为了让目标区间够大，你要不一开始就给它一个正确大小，要不就显式变更其大小。这两个办法都只适用于序列式容器（`vector`、`deque`、`list` 和 `forward_list`）。关联式容器根本不会有此问题，因为关联式容器不可能被当作覆写式算法（overwriting algorithm）的操作目标（原因见 6.7.2 节第 221 页）。以下例子展示了如何扩充容器大小：

```
// stl/copy1.cpp

#include <algorithm>
#include <list>
#include <vector>
#include <deque>
using namespace std;

int main()
{
    list<int>   coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    vector<int> coll2;

    // resize destination to have enough room for the overwriting algorithm
    coll2.resize (coll1.size());

    // copy elements from first into second collection
    // - overwrites existing elements in destination
    copy (coll1.cbegin(), coll1.cend(),     // source
          coll2.begin());                   // destination

    // create third collection with enough room
    // - initial size is passed as parameter
    deque<int> coll3(coll1.size());

    // copy elements from first into third collection
    copy (coll1.cbegin(), coll1.cend(),     // source
          coll3.begin());                   // destination
}
```

在这里，`resize()` 的作用是改变 `coll2` 的元素个数：

```
coll2.resize (coll1.size());
```

coll3 则是在初始化时就指明要有足够空间，以容纳 coll1 中的全部元素：

```
deque<int> coll3(coll1.size());
```

注意，这两种方法都会产出新元素并赋予初值。这些元素由 default 构造函数初始化，没有任何实参。你可以传递额外的实参给构造函数和 resize()，这样就可以按你的意愿将新元素初始化。