捕获列表只用于局部非 static 变量，lambda 可以直接使用局部 static 变量和在它所在函数之外声明的名字。

### 完整的 biggies

到目前为止，我们已经解决了程序的所有细节，下面就是完整的程序：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序，删除重复单词
    // 按长度排序，长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(),
                [](const string &a, const string &b)
                  { return a.size() < b.size();});
    // 获取一个迭代器，指向第一个满足 size()>= sz 的元素
    auto wc = find_if(words.begin(), words.end(),
                [sz](const string &a)
                    { return a.size() >= sz; });
    // 计算满足 size >= sz 的元素的数目
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
         << " of length " << sz << " or longer" << endl;
    // 打印长度大于等于给定值的单词，每个单词后面接一个空格
    for_each(wc, words.end(),
             [](const string &s){cout << s << " ";});
    cout << endl;
}
```

## 10.3.2 节练习

392

**练习 10.14**：编写一个 lambda，接受两个 int，返回它们的和。

**练习 10.15**：编写一个 lambda，捕获它所在函数的 int，并接受一个 int 参数。lambda 应该返回捕获的 int 和 int 参数的和。

**练习 10.16**：使用 lambda 编写你自己版本的 biggies。

**练习 10.17**：重写 10.3.1 节练习 10.12(第 345 页)的程序，在对 sort 的调用中使用 lambda 来代替函数 compareIsbn。

**练习 10.18**：重写 biggies，用 partition 代替 find_if。我们在 10.3.1 节练习 10.13（第 345 页）中介绍了 partition 算法。

**练习 10.19**：用 stable_partition 重写前一题的程序，与 stable_sort 类似，在划分后的序列中维持原有元素的顺序。

## 10.3.3 lambda 捕获和返回

当定义一个 lambda 时，编译器生成一个与 lambda 对应的新的（未命名的）类类型。我们将在 14.8.1 节（第 507 页）介绍这种类是如何生成的。目前，可以这样理解，当向一个函数传递一个 lambda 时，同时定义了一个新类型和该类型的一个对象：传递的参数就

是此编译器生成的类类型的未命名对象。类似的，当使用 auto 定义一个用 lambda 初始化的变量时，定义了一个从 lambda 生成的类型的对象。

默认情况下，从 lambda 生成的类都包含一个对应该 lambda 所捕获的变量的数据成员。类似任何普通类的数据成员，lambda 的数据成员也在 lambda 对象创建时被初始化。

### 值捕获

类似参数传递，变量的捕获方式也可以是值或引用。表 10.1（第 352 页）列出了几种不同的构造捕获列表的方式。到目前为止，我们的 lambda 采用值捕获的方式。与传值参数类似，采用值捕获的前提是变量可以拷贝。与参数不同，被捕获的变量的值是在 lambda 创建时拷贝，而不是调用时拷贝：

```
void fcn1()
{
    size_t v1 = 42; // 局部变量
    // 将 v1 拷贝到名为 f 的可调用对象
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j 为 42；f 保存了我们创建它时 v1 的拷贝
}
```

由于被捕获变量的值是在 lambda 创建时拷贝，因此随后对其修改不会影响到 lambda 内对应的值。

393 ### 引用捕获

我们定义 lambda 时可以采用引用方式捕获变量。例如：

```
void fcn2()
{
    size_t v1 = 42; // 局部变量
    // 对象 f2 包含 v1 的引用
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j 为 0；f2 保存 v1 的引用，而非拷贝
}
```

v1 之前的&指出 v1 应该以引用方式捕获。一个以引用方式捕获的变量与其他任何类型的引用的行为类似。当我们在 lambda 函数体内使用此变量时，实际上使用的是引用所绑定的对象。在本例中，当 lambda 返回 v1 时，它返回的是 v1 指向的对象的值。

引用捕获与返回引用（参见 6.3.2 节，第 201 页）有着相同的问题和限制。如果我们采用引用方式捕获一个变量，就必须确保被引用的对象在 lambda 执行的时候是存在的。lambda 捕获的都是局部变量，这些变量在函数结束后就不复存在了。如果 lambda 可能在函数结束后执行，捕获的引用指向的局部变量已经消失。

引用捕获有时是必要的。例如，我们可能希望 biggies 函数接受一个 ostream 的引用，用来输出数据，并接受一个字符作为分隔符：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // 与之前例子一样的重排 words 的代码
```

```
    // 打印 count 的语句改为打印到 os
    for_each(words.begin(), words.end(),
             [&os, c](const string &s) { os << s << c; });
}
```

我们不能拷贝 ostream 对象（参见 8.1.1 节，第 279 页），因此捕获 os 的唯一方法就是捕获其引用（或指向 os 的指针）。

当我们向一个函数传递一个 lambda 时，就像本例中调用 for_each 那样，lambda 会立即执行。在此情况下，以引用方式捕获 os 没有问题，因为当 for_each 执行时，biggies 中的变量是存在的。

我们也可以从一个函数返回 lambda。函数可以直接返回一个可调用对象，或者返回一个类对象，该类含有可调用对象的数据成员。如果函数返回一个 lambda，则与函数不能返回一个局部变量的引用类似，此 lambda 也不能包含引用捕获。


当以引用方式捕获一个变量时，必须保证在 lambda 执行时变量是存在的。

**建议：尽量保持 lambda 的变量捕获简单化** 394

一个 lambda 捕获从 lambda 被创建（即，定义 lambda 的代码执行时）到 lambda 自身执行（可能有多次执行）这段时间内保存的相关信息。确保 lambda 每次执行的时候这些信息都有预期的意义，是程序员的责任。

捕获一个普通变量，如 int、string 或其他非指针类型，通常可以采用简单的值捕获方式。在此情况下，只需关注变量在捕获时是否有我们所需的值就可以了。

如果我们捕获一个指针或迭代器，或采用引用捕获方式，就必须确保在 lambda 执行时，绑定到迭代器、指针或引用的对象仍然存在。而且，需要保证对象具有预期的值。在 lambda 从创建到它执行的这段时间内，可能有代码改变绑定的对象的值。也就是说，在指针（或引用）被捕获的时刻，绑定的对象的值是我们所期望的，但在 lambda 执行时，该对象的值可能已经完全不同了。

一般来说，我们应该尽量减少捕获的数据量，来避免潜在的捕获导致的问题。而且，如果可能的话，应该避免捕获指针或引用。

### 隐式捕获

除了显式列出我们希望使用的来自所在函数的变量之外，还可以让编译器根据 lambda 体中的代码来推断我们要使用哪些变量。为了指示编译器推断捕获列表，应在捕获列表中写一个&或=。&告诉编译器采用捕获引用方式，=则表示采用值捕获方式。例如，我们可以重写传递给 find_if 的 lambda：

```
// sz 为隐式捕获，值捕获方式
wc = find_if(words.begin(), words.end(),
             [=](const string &s)
                { return s.size() >= sz; });
```

如果我们希望对一部分变量采用值捕获，对其他变量采用引用捕获，可以混合使用隐式捕获和显式捕获：

```
void biggies(vector<string> &words,
```

```
                vector<string>::size_type sz,
                ostream &os = cout, char c = ' ')
{
    // 其他处理与前例一样
    // os 隐式捕获，引用捕获方式；c 显式捕获，值捕获方式
    for_each(words.begin(), words.end(),
             [&, c](const string &s) { os << s << c; });
    // os 显式捕获，引用捕获方式；c 隐式捕获，值捕获方式
    for_each(words.begin(), words.end(),
             [=, &os](const string &s) { os << s << c; });
}
```

395 当我们混合使用隐式捕获和显式捕获时，捕获列表中的第一个元素必须是一个&或=。此符号指定了默认捕获方式为引用或值。

当混合使用隐式捕获和显式捕获时，显式捕获的变量必须使用与隐式捕获不同的方式。即，如果隐式捕获是引用方式（使用了&），则显式捕获命名变量必须采用值方式，因此不能在其名字前使用&。类似的，如果隐式捕获采用的是值方式（使用了=），则显式捕获命名变量必须采用引用方式，即，在名字前使用&。

**表 10.1：lambda 捕获列表**

| | |
|---|---|
| [] | 空捕获列表。lambda 不能使用所在函数中的变量。一个 lambda 只有捕获变量后才能使用它们 |
| [*names*] | *names* 是一个逗号分隔的名字列表，这些名字都是 lambda 所在函数的局部变量。默认情况下，捕获列表中的变量都被拷贝。名字前如果使用了&，则采用引用捕获方式 |
| [&] | 隐式捕获列表，采用引用捕获方式。lambda 体中所使用的来自所在函数的实体都采用引用方式使用 |
| [=] | 隐式捕获列表，采用值捕获方式。lambda 体将拷贝所使用的来自所在函数的实体的值 |
| [&, *identifier_list*] | *identifier_list* 是一个逗号分隔的列表，包含 0 个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用方式捕获。*identifier_list* 中的名字前面不能使用& |
| [=, *identifier_list*] | *identifier_list* 中的变量都采用引用方式捕获，而任何隐式捕获的变量都采用值方式捕获。*identifier_list* 中的名字不能包括 this，且这些名字之前必须使用& |

### 可变 lambda

默认情况下，对于一个值被拷贝的变量，lambda 不会改变其值。如果我们希望能改变一个被捕获的变量的值，就必须在参数列表首加上关键字 mutable。因此，可变 lambda 能省略参数列表：

```
void fcn3()
{
    size_t v1 = 42; // 局部变量
    // f 可以改变它所捕获的变量的值
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j 为 43
}
```

一个引用捕获的变量是否（如往常一样）可以修改依赖于此引用指向的是一个 const 类型还是一个非 const 类型：

396

```
void fcn4()
{
    size_t v1 = 42; // 局部变量
    // v1 是一个非 const 变量的引用
    // 可以通过 f2 中的引用来改变它
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j 为 1
}
```

### 指定 lambda 返回类型

到目前为止，我们所编写的 lambda 都只包含单一的 return 语句。因此，我们还未遇到必须指定返回类型的情况。默认情况下，如果一个 lambda 体包含 return 之外的任何语句，则编译器假定此 lambda 返回 void。与其他返回 void 的函数类似，被推断返回 void 的 lambda 不能返回值。

下面给出了一个简单的例子，我们可以使用标准库 transform 算法和一个 lambda 来将一个序列中的每个负数替换为其绝对值：

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) { return i < 0 ? -i : i; });
```

函数 transform 接受三个迭代器和一个可调用对象。前两个迭代器表示输入序列，第三个迭代器表示目的位置。算法对输入序列中每个元素调用可调用对象，并将结果写到目的位置。如本例所示，目的位置迭代器与表示输入序列开始位置的迭代器可以是相同的。当输入迭代器和目的迭代器相同时，transform 将输入序列中每个元素替换为可调用对象操作该元素得到的结果。

在本例中，我们传递给 transform 一个 lambda，它返回其参数的绝对值。lambda 体是单一的 return 语句，返回一个条件表达式的结果。我们无须指定返回类型，因为可以根据条件运算符的类型推断出来。

但是，如果我们将程序改写为看起来是等价的 if 语句，就会产生编译错误：

```
// 错误：不能推断 lambda 的返回类型
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) { if (i < 0) return -i; else return i; });
```

编译器推断这个版本的 lambda 返回类型为 void，但它返回了一个 int 值。

当我们需要为一个 lambda 定义返回类型时，必须使用尾置返回类型（参见 6.3.3 节，第 206 页）：

C++ 11

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) -> int
          { if (i < 0) return -i; else return i; });
```

在此例中，传递给 transform 的第四个参数是一个 lambda，它的捕获列表是空的，接受单一 int 参数，返回一个 int 值。它的函数体是一个返回其参数的绝对值的 if 语句。

397

### 10.3.3 节练习

**练习 10.20**：标准库定义了一个名为 count_if 的算法。类似 find_if，此函数接受一对迭代器，表示一个输入范围，还接受一个谓词，会对输入范围中每个元素执行。count_if 返回一个计数值，表示谓词有多少次为真。使用 count_if 重写我们程序中统计有多少单词长度超过 6 的部分。

**练习 10.21**：编写一个 lambda，捕获一个局部 int 变量，并递减变量值，直至它变为 0。一旦变量变为 0，再调用 lambda 应该不再递减变量。lambda 应该返回一个 bool 值，指出捕获的变量是否为 0。

## 10.3.4 参数绑定

对于那种只在一两个地方使用的简单操作，lambda 表达式是最有用的。如果我们需要在很多地方使用相同的操作，通常应该定义一个函数，而不是多次编写相同的 lambda 表达式。类似的，如果一个操作需要很多语句才能完成，通常使用函数更好。

如果 lambda 的捕获列表为空，通常可以用函数来代替它。如前面章节所示，既可以用一个 lambda，也可以用函数 isShorter 来实现将 vector 中的单词按长度排序。类似的，对于打印 vector 内容的 lambda，编写一个函数来替换它也是很容易的事情，这个函数只需接受一个 string 并在标准输出上打印它即可。

但是，对于捕获局部变量的 lambda，用函数来替换它就不是那么容易了。例如，我们用在 find_if 调用中的 lambda 比较一个 string 和一个给定大小。我们可以很容易地编写一个完成同样工作的函数：

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

但是，我们不能用这个函数作为 find_if 的一个参数。如前文所示，find_if 接受一个一元谓词，因此传递给 find_if 的可调用对象必须接受单一参数。biggies 传递给 find_if 的 lambda 使用捕获列表来保存 sz。为了用 check_size 来代替此 lambda，必须解决如何向 sz 形参传递一个参数的问题。

### 标准库 bind 函数

我们可以解决向 check_size 传递一个长度参数的问题，方法是使用一个新的名为 **bind** 的标准库函数，它定义在头文件 functional 中。可以将 bind 函数看作一个通用的函数适配器（参见 9.6 节，第 329 页），它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。

C++ 11

398

调用 bind 的一般形式为：

```
auto newCallable = bind(callable, arg_list);
```

其中，*newCallable* 本身是一个可调用对象，*arg_list* 是一个逗号分隔的参数列表，对应给定的 *callable* 的参数。即，当我们调用 *newCallable* 时，*newCallable* 会调用 *callable*，并传递给它 *arg_list* 中的参数。

*arg_list* 中的参数可能包含形如 *_n* 的名字，其中 *n* 是一个整数。这些参数是“占位符”，

表示 *newCallable* 的参数，它们占据了传递给 *newCallable* 的参数的“位置”。数值 *n* 表示生成的可调用对象中参数的位置：_1 为 *newCallable* 的第一个参数，_2 为第二个参数，依此类推。

### 绑定 check_size 的 sz 参数

作为一个简单的例子，我们将使用 bind 生成一个调用 check_size 的对象，如下所示，它用一个定值作为其大小参数来调用 check_size：

```
// check6 是一个可调用对象，接受一个 string 类型的参数
// 并用此 string 和值 6 来调用 check_size
auto check6 = bind(check_size, _1, 6);
```

此 bind 调用只有一个占位符，表示 check6 只接受单一参数。占位符出现在 *arg_list* 的第一个位置，表示 check6 的此参数对应 check_size 的第一个参数。此参数是一个 const string&。因此，调用 check6 必须传递给它一个 string 类型的参数，check6 会将此参数传递给 check_size。

```
string s = "hello";
bool b1 = check6(s); // check6(s)会调用 check_size(s, 6)
```

使用 bind，我们可以将原来基于 lambda 的 find_if 调用：

```
auto wc = find_if(words.begin(), words.end(),
                  [sz](const string &a)
```

替换为如下使用 check_size 的版本：

```
auto wc = find_if(words.begin(), words.end(),
                  bind(check_size, _1, sz));
```

此 bind 调用生成一个可调用对象，将 check_size 的第二个参数绑定到 sz 的值。当 find_if 对 words 中的 string 调用这个对象时，这些对象会调用 check_size，将给定的 string 和 sz 传递给它。因此，find_if 可以有效地对输入序列中每个 string 调用 check_size，实现 string 的大小与 sz 的比较。

### 使用 placeholders 名字

399

名字 _*n* 都定义在一个名为 placeholders 的命名空间中，而这个命名空间本身定义在 std 命名空间（参见 3.1 节，第 74 页）中。为了使用这些名字，两个命名空间都要写上。与我们的其他例子类似，对 bind 的调用代码假定之前已经恰当地使用了 using 声明。例如，_1 对应的 using 声明为：

```
using std::placeholders::_1;
```

此声明说明我们要使用的名字 _1 定义在命名空间 placeholders 中，而此命名空间又定义在命名空间 std 中。

对每个占位符名字，我们都必须提供一个单独的 using 声明。编写这样的声明很烦人，也很容易出错。可以使用另外一种不同形式的 using 语句（详细内容将在 18.2.2 节（第 702 页）中介绍），而不是分别声明每个占位符，如下所示：

```
using namespace namespace_name;
```

这种形式说明希望所有来自 *namespace_name* 的名字都可以在我们的程序中直接使用。例如：

```
using namespace std::placeholders;
```

使得由 placeholders 定义的所有名字都可用。与 bind 函数一样，placeholders 命名空间也定义在 functional 头文件中。

### bind 的参数

如前文所述，我们可以用 bind 修正参数的值。更一般的，可以用 bind 绑定给定可调用对象中的参数或重新安排其顺序。例如，假定 f 是一个可调用对象，它有 5 个参数，则下面对 bind 的调用：

```
// g 是一个有两个参数的可调用对象
auto g = bind(f, a, b, _2, c, _1);
```

生成一个新的可调用对象，它有两个参数，分别用占位符_2 和_1 表示。这个新的可调用对象将它自己的参数作为第三个和第五个参数传递给 f。f 的第一个、第二个和第四个参数分别被绑定到给定的值 a、b 和 c 上。

传递给 g 的参数按位置绑定到占位符。即，第一个参数绑定到 _1，第二个参数绑定到 _2。因此，当我们调用 g 时，其第一个参数将被传递给 f 作为最后一个参数，第二个参数将被传递给 f 作为第三个参数。实际上，这个 bind 调用会将

```
g(_1, _2)
```

映射为

```
f(a, b, _2, c, _1)
```

即，对 g 的调用会调用 f，用 g 的参数代替占位符，再加上绑定的参数 a、b 和 c。例如，调用 g(X, Y) 会调用

```
f(a, b, Y, c, X)
```

400

### 用 bind 重排参数顺序

下面是用 bind 重排参数顺序的一个具体例子，我们可以用 bind 颠倒 isShroter 的含义：

```
// 按单词长度由短至长排序
sort(words.begin(), words.end(), isShorter);
// 按单词长度由长至短排序
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

在第一个调用中，当 sort 需要比较两个元素 A 和 B 时，它会调用 isShorter(A, B)。在第二个对 sort 的调用中，传递给 isShorter 的参数被交换过来了。因此，当 sort 比较两个元素时，就好像调用 isShorter(B, A) 一样。

### 绑定引用参数

默认情况下，bind 的那些不是占位符的参数被拷贝到 bind 返回的可调用对象中。但是，与 lambda 类似，有时对有些绑定的参数我们希望以引用方式传递，或是要绑定参数的类型无法拷贝。

例如，为了替换一个引用方式捕获 ostream 的 lambda：

```
// os 是一个局部变量，引用一个输出流
// c 是一个局部变量，类型为 char
for_each(words.begin(), words.end(),
```

```
        [&os, c](const string &s) { os << s << c; });
```

可以很容易地编写一个函数，完成相同的工作：

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

但是，不能直接用 bind 来代替对 os 的捕获：

```
// 错误：不能拷贝 os
for_each(words.begin(), words.end(), bind(print, os, _1, ' '));
```

原因在于 bind 拷贝其参数，而我们不能拷贝一个 ostream。如果我们希望传递给 bind 一个对象而又不拷贝它，就必须使用标准库 **ref** 函数：

```
for_each(words.begin(), words.end(),
         bind(print, ref(os), _1, ' '));
```

函数 ref 返回一个对象，包含给定的引用，此对象是可以拷贝的。标准库中还有一个 **cref** 函数，生成一个保存 const 引用的类。与 bind 一样，函数 ref 和 cref 也定义在头文件 functional 中。

401

**向后兼容：参数绑定**

旧版本 C++提供的绑定函数参数的语言特性限制更多，也更复杂。标准库定义了两个分别名为 bind1st 和 bind2nd 的函数。类似 bind，这两个函数接受一个函数作为参数，生成一个新的可调用对象，该对象调用给定函数，并将绑定的参数传递给它。但是，这些函数分别只能绑定第一个或第二个参数。由于这些函数局限太强，在新标准中已被弃用（deprecated）。所谓被弃用的特性就是在新版本中不再支持的特性。新的 C++程序应该使用 bind。

### 10.3.4 节练习

**练习 10.22**：重写统计长度小于等于 6 的单词数量的程序，使用函数代替 lambda。

**练习 10.23**：bind 接受几个参数？

**练习 10.24**：给定一个 string，使用 bind 和 check_size 在一个 int 的 vector 中查找第一个大于 string 长度的值。

**练习 10.25**：在 10.3.2 节（第 349 页）的练习中，编写了一个使用 partition 的 biggies 版本。使用 check_size 和 bind 重写此函数。

## 10.4 再探迭代器

除了为每个容器定义的迭代器之外，标准库在头文件 iterator 中还定义了额外几种迭代器。这些迭代器包括以下几种。

- **插入迭代器**（insert iterator）：这些迭代器被绑定到一个容器上，可用来向容器插入元素。
- **流迭代器**（stream iterator）：这些迭代器被绑定到输入或输出流上，可用来遍历所

关联的 IO 流。

- **反向迭代器**（reverse iterator）：这些迭代器向后而不是向前移动。除了 forward_list 之外的标准库容器都有反向迭代器。
- **移动迭代器**（move iterator）：这些专用的迭代器不是拷贝其中的元素，而是移动它们。我们将在 13.6.2 节（第 480 页）介绍移动迭代器。

### 10.4.1 插入迭代器

插入器是一种迭代器适配器（参见 9.6 节，第 329 页），它接受一个容器，生成一个迭代器，能实现向给定容器添加元素。当我们通过一个插入迭代器进行赋值时，该迭代器调用容器操作来向给定容器的指定位置插入一个元素。表 10.2 列出了这种迭代器支持的操作。

**表 10.2：插入迭代器操作**

| | |
|---|---|
| `it = t` | 在 it 指定的当前位置插入值 t。假定 c 是 it 绑定的容器，依赖于插入迭代器的不同种类，此赋值会分别调用 c.push_back(t)、c.push_front(t)或 c.insert(t,p)，其中 p 为传递给 inserter 的迭代器位置 |
| `*it,++it,it++` | 这些操作虽然存在，但不会对 it 做任何事情。每个操作都返回 it |

402 插入器有三种类型，差异在于元素插入的位置：

- **back_inserter**（参见 10.2.2 节，第 341 页）创建一个使用 push_back 的迭代器。
- **front_inserter** 创建一个使用 push_front 的迭代器。
- **inserter** 创建一个使用 insert 的迭代器。此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器。元素将被插入到给定迭代器所表示的元素之前。


只有在容器支持 push_front 的情况下，我们才可以使用 front_inserter。类似的，只有在容器支持 push_back 的情况下，我们才能使用 back_inserter。

理解插入器的工作过程是很重要的：当调用 inserter(c, iter)时，我们得到一个迭代器，接下来使用它时，会将元素插入到 iter 原来所指向的元素之前的位置。即，如果 it 是由 inserter 生成的迭代器，则下面这样的赋值语句

```
*it = val;
```

其效果与下面代码一样

```
it = c.insert(it, val); // it 指向新加入的元素
++it; // 递增 it 使它指向原来的元素
```

front_inserter 生成的迭代器的行为与 inserter 生成的迭代器完全不一样。当我们使用 front_inserter 时，元素总是插入到容器第一个元素之前。即使我们传递给 inserter 的位置原来指向第一个元素，只要我们在此元素之前插入一个新元素，此元素就不再是容器的首元素了：

```
list<int> lst = {1,2,3,4};
list<int> lst2, lst3; // 空 list
```

```
// 拷贝完成之后，lst2 包含 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// 拷贝完成之后，lst3 包含 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

当调用 front_inserter(c) 时，我们得到一个插入迭代器，接下来会调用 push_front。当每个元素被插入到容器 c 中时，它变为 c 的新的首元素。因此，front_inserter 生成的迭代器会将插入的元素序列的顺序颠倒过来，而 inserter 和 back_inserter 则不会。

### 10.4.1 节练习

403

**练习 10.26**：解释三种插入迭代器的不同之处。

**练习 10.27**：除了 unique（参见 10.2.3 节，第 343 页）之外，标准库还定义了名为 unique_copy 的函数，它接受第三个迭代器，表示拷贝不重复元素的目的位置。编写一个程序，使用 unique_copy 将一个 vector 中不重复的元素拷贝到一个初始为空的 list 中。

**练习 10.28**：一个 vector 中保存 1 到 9，将其拷贝到三个其他容器中。分别使用 inserter、back_inserter 和 front_inserter 将元素添加到三个容器中。对每种 inserter，估计输出序列是怎样的，运行程序验证你的估计是否正确。

## 10.4.2 iostream 迭代器

虽然 iostream 类型不是容器，但标准库定义了可以用于这些 IO 类型对象的迭代器（参见 8.1 节，第 278 页）。**istream_iterator**（参见表 10.3）读取输入流，**ostream_iterator**（参见表 10.4 节，第 361 页）向一个输出流写数据。这些迭代器将它们对应的流当作一个特定类型的元素序列来处理。通过使用流迭代器，我们可以用泛型算法从流对象读取数据以及向其写入数据。

### istream_iterator 操作

当创建一个流迭代器时，必须指定迭代器将要读写的对象类型。一个 istream_iterator 使用>>来读取流。因此，istream_iterator 要读取的类型必须定义了输入运算符。当创建一个 istream_iterator 时，我们可以将它绑定到一个流。当然，我们还可以默认初始化迭代器，这样就创建了一个可以当作尾后值使用的迭代器。

```
istream_iterator<int> int_it(cin);    // 从 cin 读取 int
istream_iterator<int> int_eof;        // 尾后迭代器
ifstream in("afile");
istream_iterator<string> str_it(in); // 从"afile"读取字符串
```

下面是一个用 istream_iterator 从标准输入读取数据，存入一个 vector 的例子：

```
istream_iterator<int> in_iter(cin);  // 从 cin 读取 int
istream_iterator<int> eof;           // istream 尾后迭代器
while (in_iter != eof)               // 当有数据可供读取时
    // 后置递增运算读取流，返回迭代器的旧值
    // 解引用迭代器，获得从流读取的前一个值
    vec.push_back(*in_iter++);
```

此循环从 cin 读取 int 值，保存在 vec 中。在每个循环步中，循环体代码检查 in_iter 是否等于 eof。eof 被定义为空的 istream_iterator，从而可以当作尾后迭代器来使
404 用。对于一个绑定到流的迭代器，一旦其关联的流遇到文件尾或遇到 IO 错误，迭代器的值就与尾后迭代器相等。

此程序最困难的部分是传递给 push_back 的参数，其中用到了解引用运算符和后置递增运算符。该表达式的计算过程与我们之前写过的其他结合解引用和后置递增运算的表达式一样（参见 4.5 节，第 131 页）。后置递增运算会从流中读取下一个值，向前推进，但返回的是迭代器的旧值。迭代器的旧值包含了从流中读取的前一个值，对迭代器进行解引用就能获得此值。

我们可以将程序重写为如下形式，这体现了 istream_iterator 更有用的地方：

```
istream_iterator<int> in_iter(cin), eof; // 从 cin 读取 int
vector<int> vec(in_iter, eof); // 从迭代器范围构造 vec
```

本例中我们用一对表示元素范围的迭代器来构造 vec。这两个迭代器是 istream_iterator，这意味着元素范围是通过从关联的流中读取数据获得的。这个构造函数从 cin 中读取数据，直至遇到文件尾或者遇到一个不是 int 的数据为止。从流中读取的数据被用来构造 vec。

**表 10.3：istream_iterator 操作**

| | |
|---|---|
| istream_iterator<T> in(is); | in 从输入流 is 读取类型为 T 的值 |
| istream_iterator<T> end; | 读取类型为 T 的值的 istream_iterator 迭代器，表示尾后位置 |
| in1 == in2<br>in1 != in2 | in1 和 in2 必须读取相同类型。如果它们都是尾后迭代器，或绑定到相同的输入，则两者相等 |
| *in | 返回从流中读取的值 |
| in->mem | 与 (*in).mem 的含义相同 |
| ++in, in++ | 使用元素类型所定义的>>运算符从输入流中读取下一个值。与以往一样，前置版本返回一个指向递增后迭代器的引用，后置版本返回旧值 |

### 使用算法操作流迭代器

由于算法使用迭代器操作来处理数据，而流迭代器又至少支持某些迭代器操作，因此我们至少可以用某些算法来操作流迭代器。我们在 10.5.1 节（第 365 页）会看到如何分辨哪些算法可以用于流迭代器。下面是一个例子，我们可以用一对 istream_iterator 来调用 accumulate：

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

此调用会计算出从标准输入读取的值的和。如果输入为：

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

则输出为 664。

### 405 istream_iterator 允许使用懒惰求值

当我们将一个 istream_iterator 绑定到一个流时，标准库并不保证迭代器立即从流读取数据。具体实现可以推迟从流中读取数据，直到我们使用迭代器时才真正读取。标

准库中的实现所保证的是，在我们第一次解引用迭代器之前，从流中读取数据的操作已经完成了。对于大多数程序来说，立即读取还是推迟读取没什么差别。但是，如果我们创建了一个 istream_iterator，没有使用就销毁了，或者我们正在从两个不同的对象同步读取同一个流，那么何时读取可能就很重要了。

### ostream_iterator 操作

我们可以对任何具有输出运算符（<<运算符）的类型定义 ostream_iterator。当创建一个 ostream_iterator 时，我们可以提供（可选的）第二参数，它是一个字符串，在输出每个元素后都会打印此字符串。此字符串必须是一个 C 风格字符串（即，一个字符串字面常量或者一个指向以空字符结尾的字符数组的指针）。必须将 ostream_iterator 绑定到一个指定的流，不允许空的或表示尾后位置的 ostream_iterator。

**表 10.4：ostream_iterator 操作**

| | |
|---|---|
| `ostream_iterator<T> out(os);` | out 将类型为 T 的值写到输出流 os 中 |
| `ostream_iterator<T> out(os,d);` | out 将类型为 T 的值写到输出流 os 中，每个值后面都输出一个 d。d 指向一个空字符结尾的字符数组 |
| `out = val` | 用<<运算符将 val 写入到 out 所绑定的 ostream 中。val 的类型必须与 out 可写的类型兼容 |
| `*out, ++out, out++` | 这些运算符是存在的，但不对 out 做任何事情。每个运算符都返回 out |

我们可以用 ostream_iterator 来输出值的序列：

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // 赋值语句实际上将元素写到 cout
cout << endl;
```

此程序将 vec 中的每个元素写到 cout，每个元素后加一个空格。每次向 out_iter 赋值时，写操作就会被提交。

值得注意的是，当我们向 out_iter 赋值时，可以忽略解引用和递增运算。即，循环可以重写成下面的样子：

```
for (auto e : vec)
    out_iter = e; // 赋值语句将元素写到 cout
cout << endl;
```

运算符*和++实际上对 ostream_iterator 对象不做任何事情，因此忽略它们对我们的程序没有任何影响。但是，推荐第一种形式。在这种写法中，流迭代器的使用与其他迭代器的使用保持一致。如果想将此循环改为操作其他迭代器类型，修改起来非常容易。而且，对于读者来说，此循环的行为也更为清晰。 406

可以通过调用 copy 来打印 vec 中的元素，这比编写循环更为简单：

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

perfect

### 使用流迭代器处理类类型

我们可以为任何定义了输入运算符（>>）的类型创建 istream_iterator 对象。类似的，只要类型有输出运算符（<<），我们就可以为其定义 ostream_iterator。由于 Sales_item 既有输入运算符也有输出运算符，因此可以使用 IO 迭代器重写 1.6 节（第 21 页）中的书店程序：

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// 将第一笔交易记录存在 sum 中，并读取下一条记录
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // 如果当前交易记录（存在 item_iter 中）有着相同的 ISBN 号
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++;        // 将其加到 sum 上并读取下一条记录
    else {
        out_iter = sum;             // 输出 sum 当前值
        sum = *item_iter++;         // 读取下一条记录
    }
}
out_iter = sum;                     // 记得打印最后一组记录的和
```

此程序使用 item_iter 从 cin 读取 Sales_item 交易记录，并将和写入 cout，每个结果后面都跟一个换行符。定义了自己的迭代器后，我们就可以用 item_iter 读取第一条交易记录，用它的值来初始化 sum：

```
// 将第一条交易记录保存在 sum 中，并读取下一条记录
Sales_item sum = *item_iter++;
```

此处，我们对 item_iter 执行后置递增操作，对结果进行解引用操作。这个表达式读取下一条交易记录，并用之前保存在 item_iter 中的值来初始化 sum。

while 循环会反复执行，直至在 cin 上遇到文件尾为止。在 while 循环体中，我们检查 sum 与刚刚读入的记录是否对应同一本书。如果两者的 ISBN 不同，我们将 sum 赋予 out_iter，这将会打印 sum 的当前值，并接着打印一个换行符。在打印了前一本书的交易金额之和后，我们将最近读入的交易记录的副本赋予 sum，并递增迭代器，这将读取下一条交易记录。循环会这样持续下去，直至遇到错误或文件尾。在退出之前，记住要打印输入中最后一本书的交易金额之和。

407

### 10.4.2 节练习

**练习 10.29**：编写程序，使用流迭代器读取一个文本文件，存入一个 vector 中的 string 里。

**练习 10.30**：使用流迭代器、sort 和 copy 从标准输入读取一个整数序列，将其排序，并将结果写到标准输出。

**练习 10.31**：修改前一题的程序，使其只打印不重复的元素。你的程序应使用 unique_copy（参见 10.4.1 节，第 359 页）。

**练习 10.32**：重写 1.6 节（第 21 页）中的书店程序，使用一个 vector 保存交易记录，使用不同算法完成处理。使用 sort 和 10.3.1 节（第 345 页）中的 compareIsbn 函数来排序交易记录，然后使用 find 和 accumulate 求和。

**练习 10.33**：编写程序，接受三个参数：一个输入文件和两个输出文件的文件名。输入文件保存的应该是整数。使用 istream_iterator 读取输入文件。使用 ostream_iterator 将奇数写入第一个输出文件，每个值之后都跟一个空格。将偶数写入第二个输出文件，每个值都独占一行。

### 10.4.3 反向迭代器

反向迭代器就是在容器中从尾元素向首元素反向移动的迭代器。对于反向迭代器，递增（以及递减）操作的含义会颠倒过来。递增一个反向迭代器（++it）会移动到前一个元素；递减一个迭代器（--it）会移动到下一个元素。

除了 forward_list 之外，其他容器都支持反向迭代器。我们可以通过调用 rbegin、rend、crbegin 和 crend 成员函数来获得反向迭代器。这些成员函数返回指向容器尾元素和首元素之前一个位置的迭代器。与普通迭代器一样，反向迭代器也有 const 和非 const 版本。

图 10.1 显示了一个名为 vec 的假设的 vector 上的 4 种迭代器：

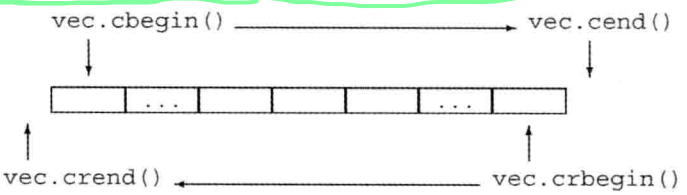


图 10.1：比较 cbegin/cend 和 crbegin/crend

下面的循环是一个使用反向迭代器的例子，它按逆序打印 vec 中的元素：

```
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
// 从尾元素到首元素的反向迭代器                                    408
for (auto r_iter = vec.crbegin();     // 将 r_iter 绑定到尾元素
          r_iter != vec.crend();      // crend 指向首元素之前的位置
          ++r_iter)                   // 实际是递减，移动到前一个元素
    cout << *r_iter << endl;          // 打印 9, 8, 7, ... 0
```

虽然颠倒递增和递减运算符的含义可能看起来令人混淆，但这样做使我们可以用算法透明地向前或向后处理容器。例如，可以通过向 sort 传递一对反向迭代器来将 vector 整理为递减序：

```
sort(vec.begin(), vec.end()); // 按“正常序”排序 vec
// 按逆序排序：将最小元素放在 vec 的末尾
sort(vec.rbegin(), vec.rend());
```

#### 反向迭代器需要递减运算符

不必惊讶，我们只能从既支持++也支持--的迭代器来定义反向迭代器。毕竟反向迭代器的目的是在序列中反向移动。除了 forward_list 之外，标准容器上的其他迭代器都既支持递增运算又支持递减运算。但是，流迭代器不支持递减运算，因为不可能在一个流中反向移动。因此，不可能从一个 forward_list 或一个流迭代器创建反向迭代器。

#### 反向迭代器和其他迭代器间的关系

假定有一个名为 line 的 string，保存着一个逗号分隔的单词列表，我们希望打印

line 中的第一个单词。使用 find 可以很容易地完成这一任务：

```
// 在一个逗号分隔的列表中查找第一个元素
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
```

如果 line 中有逗号，那么 comma 将指向这个逗号；否则，它将等于 line.cend()。当我们打印从 line.cbegin() 到 comma 之间的内容时，将打印到逗号为止的字符，或者打印整个 string（如果其中不含逗号的话）。

如果希望打印最后一个单词，可以改用反向迭代器：

```
// 在一个逗号分隔的列表中查找最后一个元素
auto rcomma = find(line.crbegin(), line.crend(), ',');
```

由于我们将 crbegin() 和 crend() 传递给 find，find 将从 line 的最后一个字符开始向前搜索。当 find 完成后，如果 line 中有逗号，则 rcomma 指向最后一个逗号——即，它指向反向搜索中找到的第一个逗号。如果 line 中没有逗号，则 rcomma 指向 line.crend()。

当我们试图打印找到的单词时，最有意思的部分就来了。看起来下面的代码是显然的方法

```
// 错误：将逆序输出单词的字符
cout << string(line.crbegin(), rcomma) << endl;
```

409 但它会生成错误的输出结果。例如，如果我们的输入是

```
FIRST,MIDDLE,LAST
```

则这条语句会打印 TSAL!

图 10.2 说明了问题所在：我们使用的是反向迭代器，会反向处理 string。因此，上述输出语句从 crbegin 开始反向打印 line 中内容。而我们希望按正常顺序打印从 rcomma 开始到 line 末尾间的字符。但是，我们不能直接使用 rcomma。因为它是一个反向迭代器，意味着它会反向朝着 string 的开始位置移动。需要做的是，将 rcomma 转换回一个普通迭代器，能在 line 中正向移动。我们通过调用 reverse_iterator 的 base 成员函数来完成这一转换，此成员函数会返回其对应的普通迭代器：

```
//正确：得到一个正向迭代器，从逗号开始读取字符直到 line 末尾
cout << string(rcomma.base(), line.cend()) << endl;
```

给定和之前一样的输入，这条语句会如我们的预期打印出 LAST。

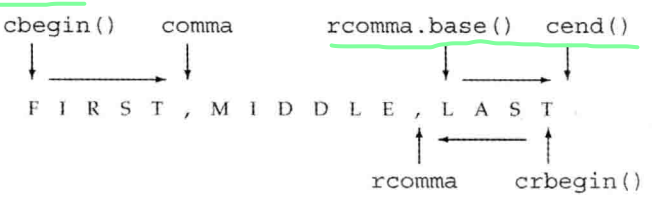


图 10.2：反向迭代器和普通迭代器间的关系

图 10.2 中的对象显示了普通迭代器与反向迭代器之间的关系。例如，rcomma 和 rcomma.base() 指向不同的元素，line.crbegin 和 line.cend() 也是如此。这些不同保证了元素范围无论是正向处理还是反向处理都是相同的。

从技术上讲，普通迭代器与反向迭代器的关系反映了左闭合区间（参见 9.2.1 节，第 296 页）的特性。关键点在于 [line.crbegin(), rcomma) 和 [rcomma.base(), line.cend()) 指向 line 中相同的元素范围。为了实现这一点，rcomma 和 rcomma.base() 必须生成相邻位置而不是相同位置，crbegin() 和 cend() 也是如此。

反向迭代器的目的是表示元素范围，而这些范围是不对称的，这导致一个重要的结果：当我们从一个普通迭代器初始化一个反向迭代器，或是给一个反向迭代器赋值时，结果迭代器与原迭代器指向的并不是相同的元素。

### 10.4.3 节练习

410

**练习 10.34**：使用 reverse_iterator 逆序打印一个 vector。

**练习 10.35**：使用普通迭代器逆序打印一个 vector。

**练习 10.36**：使用 find 在一个 int 的 list 中查找最后一个值为 0 的元素。

**练习 10.37**：给定一个包含 10 个元素的 vector，将位置 3 到 7 之间的元素按逆序拷贝到一个 list 中。

## 10.5 泛型算法结构

任何算法的最基本的特性是它要求其迭代器提供哪些操作。某些算法，如 find，只要求通过迭代器访问元素、递增迭代器以及比较两个迭代器是否相等这些能力。其他一些算法，如 sort，还要求读、写和随机访问元素的能力。算法所要求的迭代器操作可以分为 5 个**迭代器类别**（iterator category），如表 10.5 所示。每个算法都会对它的每个迭代器参数指明须提供哪类迭代器。

表 10.5：迭代器类别

| | |
|---|---|
| 输入迭代器 | 只读，不写；单遍扫描，只能递增 |
| 输出迭代器 | 只写，不读；单遍扫描，只能递增 |
| 前向迭代器 | 可读写；多遍扫描，只能递增 |
| 双向迭代器 | 可读写；多遍扫描，可递增递减 |
| 随机访问迭代器 | 可读写，多遍扫描，支持全部迭代器运算 |

第二种算法分类的方式（如我们在本章开始所做的）是按照是否读、写或是重排序列中的元素来分类。附录 A 按这种分类方法列出了所有算法。

算法还共享一组参数传递规范和一组命名规范，我们在介绍迭代器类别之后将介绍这些内容。

### 10.5.1 5 类迭代器

类似容器，迭代器也定义了一组公共操作。一些操作所有迭代器都支持，另外一些只有特定类别的迭代器才支持。例如，ostream_iterator 只支持递增、解引用和赋值。vector、string 和 deque 的迭代器除了这些操作外，还支持递减、关系和算术运算。

迭代器是按它们所提供的操作来分类的，而这种分类形成了一种层次。除了输出迭代

器之外，一个高层类别的迭代器支持低层类别迭代器的所有操作。

C++标准指明了泛型和数值算法的每个迭代器参数的最小类别。例如，find 算法在
411 一个序列上进行一遍扫描，对元素进行只读操作，因此至少需要输入迭代器。replace 函数需要一对迭代器，至少是前向迭代器。类似的，replace_copy 的前两个迭代器参数也要求至少是前向迭代器。其第三个迭代器表示目的位置，必须至少是输出迭代器。其他的例子类似。对每个迭代器参数来说，其能力必须与规定的最小类别至少相当。向算法传递一个能力更差的迭代器会产生错误。


对于向一个算法传递错误类别的迭代器的问题，很多编译器不会给出任何警告或提示。

### 迭代器类别

**输入迭代器**（input iterator）：可以读取序列中的元素。一个输入迭代器必须支持

- 用于比较两个迭代器的相等和不相等运算符（==、!=）
- 用于推进迭代器的前置和后置递增运算（++）
- 用于读取元素的解引用运算符（*）；解引用只会出现在赋值运算符的右侧
- 箭头运算符（->），等价于(*it).member，即，解引用迭代器，并提取对象的成员

输入迭代器只用于顺序访问。对于一个输入迭代器，*it++保证是有效的，但递增它可能导致所有其他指向流的迭代器失效。其结果就是，不能保证输入迭代器的状态可以保存下来并用来访问元素。因此，输入迭代器只能用于单遍扫描算法。算法 find 和 accumulate 要求输入迭代器；而 istream_iterator 是一种输入迭代器。

**输出迭代器**（output iterator）：可以看作输入迭代器功能上的补集——只写而不读元素。输出迭代器必须支持

- 用于推进迭代器的前置和后置递增运算（++）
- 解引用运算符（*），只出现在赋值运算符的左侧（向一个已经解引用的输出迭代器赋值，就是将值写入它所指向的元素）

我们只能向一个输出迭代器赋值一次。类似输入迭代器，输出迭代器只能用于单遍扫描算法。用作目的位置的迭代器通常都是输出迭代器。例如，copy 函数的第三个参数就是输出迭代器。ostream_iterator 类型也是输出迭代器。

**前向迭代器**（forward iterator）：可以读写元素。这类迭代器只能在序列中沿一个方向移动。前向迭代器支持所有输入和输出迭代器的操作，而且可以多次读写同一个元素。因此，我们可以保存前向迭代器的状态，使用前向迭代器的算法可以对序列进行多遍扫描。算法 replace 要求前向迭代器，forward_list 上的迭代器是前向迭代器。

412 **双向迭代器**（bidirectional iterator）：可以正向/反向读写序列中的元素。除了支持所有前向迭代器的操作之外，双向迭代器还支持前置和后置递减运算符（--）。算法 reverse 要求双向迭代器，除了 forward_list 之外，其他标准库都提供符合双向迭代器要求的迭代器。

**随机访问迭代器**（random-access iterator）：提供在常量时间内访问序列中任意元素的能力。此类迭代器支持双向迭代器的所有功能，此外还支持表 3.7（第 99 页）中的操作：

- 用于比较两个迭代器相对位置的关系运算符（<、<=、>和>=）
- 迭代器和一个整数值的加减运算（+、+=、-和-=），计算结果是迭代器在序列中前进（或后退）给定整数个元素后的位置
- 用于两个迭代器上的减法运算符（-），得到两个迭代器的距离
- 下标运算符(iter[n])，与*(iter[n])等价

算法 sort 要求随机访问迭代器。array、deque、string 和 vector 的迭代器都是随机访问迭代器，用于访问内置数组元素的指针也是。

### 10.5.1 节练习

**练习 10.38**：列出 5 个迭代器类别，以及每类迭代器所支持的操作。

**练习 10.39**：list 上的迭代器属于哪类？vector 呢？

**练习 10.40**：你认为 copy 要求哪类迭代器？reverse 和 unique 呢？

## 10.5.2 算法形参模式

在任何其他算法分类之上，还有一组参数规范。理解这些参数规范对学习新算法很有帮助——通过理解参数的含义，你可以将注意力集中在算法所做的操作上。大多数算法具有如下 4 种形式之一：

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

其中 *alg* 是算法的名字，beg 和 end 表示算法所操作的输入范围。几乎所有算法都接受一个输入范围，是否有其他参数依赖于要执行的操作。这里列出了常见的一种——dest、beg2 和 end2，都是迭代器参数。顾名思义，如果用到了这些迭代器参数，它们分别承担指定目的位置和第二个范围的角色。除了这些迭代器参数，一些算法还接受额外的、非迭代器的特定参数。

413

### 接受单个目标迭代器的算法

dest 参数是一个表示算法可以写入的目的位置的迭代器。算法假定（assume）：按其需要写入数据，不管写入多少个元素都是安全的。


向输出迭代器写入数据的算法都假定目标空间足够容纳写入的数据。

如果 dest 是一个直接指向容器的迭代器，那么算法将输出数据写到容器中已存在的元素内。更常见的情况是，dest 被绑定到一个插入迭代器（参见 10.4.1 节，第 358 页）或是一个 ostream_iterator（参见 10.4.2 节，第 359 页）。插入迭代器会将新元素添加到容器中，因而保证空间是足够的。ostream_iterator 会将数据写入到一个输出流，同样不管要写入多少个元素都没有问题。

### 接受第二个输入序列的算法

接受单独的 beg2 或是接受 beg2 和 end2 的算法用这些迭代器表示第二个输入范围。这些算法通常使用第二个范围中的元素与第一个输入范围结合来进行一些运算。

如果一个算法接受 beg2 和 end2，这两个迭代器表示第二个范围。这类算法接受两个完整指定的范围：[beg, end) 表示的范围和 [beg2 end2) 表示的第二个范围。

只接受单独的 beg2（不接受 end2）的算法将 beg2 作为第二个输入范围中的首元素。此范围的结束位置未指定，这些算法假定从 beg2 开始的范围与 beg 和 end 所表示的范围至少一样大。


接受单独 beg2 的算法假定从 beg2 开始的序列与 beg 和 end 所表示的范围至少一样大。

## 10.5.3 算法命名规范

除了参数规范，算法还遵循一套命名和重载规范。这些规范处理诸如：如何提供一个操作代替默认的<或==运算符以及算法是将输出数据写入输入序列还是一个分离的目的位置等问题。

### 一些算法使用重载形式传递一个谓词

接受谓词参数来代替<或==运算符的算法，以及那些不接受额外参数的算法，通常都是重载的函数。函数的一个版本用元素类型的运算符来比较元素；另一个版本接受一个额外谓词参数，来代替<或==：

414

```
unique(beg, end);          // 使用 == 运算符比较元素
unique(beg, end, comp);    // 使用 comp 比较元素
```

两个调用都重新整理给定序列，将相邻的重复元素删除。第一个调用使用元素类型的==运算符来检查重复元素；第二个则调用 comp 来确定两个元素是否相等。由于两个版本的函数在参数个数上不相等，因此具体应该调用哪个版本不会产生歧义（参见 6.4 节，第 208 页）。

### _if 版本的算法

接受一个元素值的算法通常有另一个不同名的（不是重载的）版本，该版本接受一个谓词（参见 10.3.1 节，第 344 页）代替元素值。接受谓词参数的算法都有附加的 _if 前缀：

```
find(beg, end, val);          // 查找输入范围中 val 第一次出现的位置
find_if(beg, end, pred);      // 查找第一个令 pred 为真的元素
```

这两个算法都在输入范围中查找特定元素第一次出现的位置。算法 find 查找一个指定值；算法 find_if 查找使得 pred 返回非零值的元素。

这两个算法提供了命名上差异的版本，而非重载版本，因为两个版本的算法都接受相同数目的参数。因此可能产生重载歧义，虽然很罕见，但为了避免任何可能的歧义，标准库选择提供不同名字的版本而不是重载。

### 区分拷贝元素的版本和不拷贝的版本

默认情况下，重排元素的算法将重排后的元素写回给定的输入序列中。这些算法还提供另一个版本，将元素写到一个指定的输出目的位置。如我们所见，写到额外目的空间的

算法都在名字后面附加一个_copy（参见 10.2.2 节，第 341 页）：

```
reverse(beg, end);                    // 反转输入范围中元素的顺序
reverse_copy(beg, end, dest);         // 将元素按逆序拷贝到 dest
```

一些算法同时提供_copy 和_if 版本。这些版本接受一个目的位置迭代器和一个谓词：

```
// 从 v1 中删除奇数元素
remove_if(v1.begin(), v1.end(),
                      [](int i) { return i % 2; });
// 将偶数元素从 v1 拷贝到 v2; v1 不变
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
                    [](int i) { return i % 2; });
```

两个算法都调用了 lambda（参见 10.3.2 节，第 346 页）来确定元素是否为奇数。在第一个调用中，我们从输入序列中将奇数元素删除。在第二个调用中，我们将非奇数（亦即偶数）元素从输入范围拷贝到 v2 中。

### 10.5.3 节练习

415

**练习 10.41**：仅根据算法和参数的名字，描述下面每个标准库算法执行什么操作：

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

## 10.6 特定容器算法

与其他容器不同，链表类型 list 和 forward_list 定义了几个成员函数形式的算法，如表 10.6 所示。特别是，它们定义了独有的 sort、merge、remove、reverse 和 unique。通用版本的 sort 要求随机访问迭代器，因此不能用于 list 和 forward_list，因为这两个类型分别提供双向迭代器和前向迭代器。

链表类型定义的其他算法的通用版本可以用于链表，但代价太高。这些算法需要交换输入序列中的元素。一个链表可以通过改变元素间的链接而不是真的交换它们的值来快速“交换”元素。因此，这些链表版本的算法的性能比对应的通用版本好得多。

Best Practices 对于 list 和 forward_list，应该优先使用成员函数版本的算法而不是通用算法。

**表 10.6：list 和 forward_list 成员函数版本的算法**

| 这些操作都返回 void | |
|---|---|
| lst.merge(lst2) | 将来自 lst2 的元素合并入 lst。lst 和 lst2 都必须是有序的。元素将从 lst2 中删除。在合并之后，lst2 变为空。第一个版本使用<运算符；第二个版本使用给定的比较操作 |
| lst.merge(lst2, comp) | |
| lst.remove(val) | 调用 erase 删除掉与给定值相等（==）或令一元谓词为真的每个元素 |
| lst.remove_if(pred) | |
| lst.reverse() | 反转 lst 中元素的顺序 |

续表

| | |
|---|---|
| lst.sort() | 使用<或给定比较操作排序元素 |
| lst.sort(comp) | |
| lst.unique() | 调用 erase 删除同一个值的连续拷贝。第一个版本使用==；第二个版本使用给定的二元谓词 |
| lst.unique(pred) | |

### splice 成员

416 链表类型还定义了 splice 算法，其描述见表 10.7。此算法是链表数据结构所特有的，因此不需要通用版本。

表 10.7：list 和 forward_list 的 splice 成员函数的参数

| lst.splice(*args*)或 flst.splice_after(*args*) | |
|---|---|
| (p, lst2) | p 是一个指向 lst 中元素的迭代器，或一个指向 flst 首前位置的迭代器。函数将 lst2 的所有元素移动到 lst 中 p 之前的位置或是 flst 中 p 之后的位置。将元素从 lst2 中删除。lst2 的类型必须与 lst 或 flst 相同，且不能是同一个链表 |
| (p, lst2, p2) | p2 是一个指向 lst2 中位置的有效的迭代器。将 p2 指向的元素移动到 lst 中，或将 p2 之后的元素移动到 flst 中。lst2 可以是与 lst 或 flst 相同的链表 |
| (p, lst2, b, e) | b 和 e 必须表示 lst2 中的合法范围。将给定范围中的元素从 lst2 移动到 lst 或 flst。lst2 与 lst（或 flst）可以是相同的链表，但 p 不能指向给定范围中元素 |

### 链表特有的操作会改变容器

多数链表特有的算法都与其通用版本很相似，但不完全相同。链表特有版本与通用版本间的一个至关重要的区别是链表版本会改变底层的容器。例如，remove 的链表版本会删除指定的元素。unique 的链表版本会删除第二个和后继的重复元素。

类似的，merge 和 splice 会销毁其参数。例如，通用版本的 merge 将合并的序列写到一个给定的目的迭代器；两个输入序列是不变的。而链表版本的 merge 函数会销毁给定的链表——元素从参数指定的链表中删除，被合并到调用 merge 的链表对象中。在 merge 之后，来自两个链表中的元素仍然存在，但它们都已在同一个链表中。

## 10.6 节练习

**练习 10.42**：使用 list 代替 vector 重新实现 10.2.3 节（第 343 页）中的去除重复单词的程序。

done!

## 小结

417

标准库定义了大约 100 个类型无关的对序列进行操作的算法。序列可以是标准库容器类型中的元素、一个内置数组或者是（例如）通过读写一个流来生成的。算法通过在迭代器上进行操作来实现类型无关。多数算法接受的前两个参数是一对迭代器，表示一个元素范围。额外的迭代器参数可能包括一个表示目的位置的输出迭代器，或是表示第二个输入范围的另一个或另一对迭代器。

根据支持的操作不同，迭代器可分为五类：输入、输出、前向、双向以及随机访问迭代器。如果一个迭代器支持某个迭代器类别所要求的操作，则属于该类别。

如同迭代器根据操作分类一样，传递给算法的迭代器参数也按照所要求的操作进行分类。仅读取序列的算法只要求输入迭代器操作。写入数据到目的位置迭代器的算法只要求输出迭代器操作，依此类推。

算法从不直接改变它们所操作的序列的大小。它们会将元素从一个位置拷贝到另一个位置，但不会直接添加或删除元素。

虽然算法不能向序列添加元素，但插入迭代器可以做到。一个插入迭代器被绑定到一个容器上。当我们将一个容器元素类型的值赋予一个插入迭代器时，迭代器会将该值添加到容器中。

容器 `forward_list` 和 `list` 对一些通用算法定义了自己特有的版本。与通用算法不同，这些链表特有版本会修改给定的链表。

## 术语表

**back_inserter** 这是一个迭代器适配器，它接受一个指向容器的引用，生成一个插入迭代器，该插入迭代器用 `push_back` 向指定容器添加元素。

**双向迭代器（bidirectional iterator）** 支持前向迭代器的所有操作，还具有用--在序列中反向移动的能力。

**二元谓词（binary predicate）** 接受两个参数的谓词。

**bind** 标准库函数，将一个或多个参数绑定到一个可调用表达式。`bind` 定义在头文件 `functional` 中。

**可调用对象（callable object）** 可以出现在调用运算符左边的对象。函数指针、lambda 以及重载了函数调用运算符的类的对象都是可调用对象。

**捕获列表（capture list）** lambda 表达式的一部分，指出 lambda 表达式可以访问所在上下文中哪些变量。

**cref** 标准库函数，返回一个可拷贝的对象，其中保存了一个指向不可拷贝类型的 `const` 对象的引用。

**前向迭代器（forward iterator）** 可以读写元素，但不必支持--的迭代器。 418

**front_inserter** 迭代器适配器，给定一个容器，生成一个用 `push_front` 向容器开始位置添加元素的插入迭代器。

**泛型算法（generic algorithm）** 类型无关的算法。

**输入迭代器（input iterator）** 可以读但不能写序列中元素的迭代器。

**插入迭代器（insert iterator）** 迭代器适配器，生成一个迭代器，该迭代器使用容器操作向给定容器添加元素。

**插入器**（inserter） 迭代器适配器，接受一个迭代器和一个指向容器的引用，生成一个插入迭代器，该插入迭代器用 insert 在给定迭代器指向的元素之前的位置添加元素。

**istream_iterator** 读取输入流的流迭代器。

**迭代器类别**（iterator category）根据所支持的操作对迭代器进行的分类组织。迭代器类别形成一个层次，其中更强大的类别支持更弱类别的所有操作。算法使用迭代器类别来指出迭代器参数必须支持哪些操作。只要迭代器达到所要求的最小类别，它就可以用于算法。例如，一些算法只要求输入迭代器。这类算法可处理除只满足输出迭代器要求的迭代器之外的任何迭代器。而要求随机访问迭代器的算法只能用于支持随机访问操作的迭代器。

**lambda 表达式**（lambda expression） 可调用的代码单元。一个 lambda 类似一个未命名的内联函数。一个 lambda 以一个捕获列表开始，此列表允许 lambda 访问所在函数中的变量。类似函数，lambda 有一个（可能为空的）参数列表、一个返回类型和一个函数体。lambda 可以忽略返回类型。如果函数体是一个单一的 return 语句，返回类型就从返回对象的类型推断。否则，忽略的返回类型默认定为 void。

**移动迭代器**（move iterator） 迭代器适配器，生成一个迭代器，该迭代器移动而不是拷贝元素。移动迭代器将在第 13 章中进行介绍。

**ostream_iterator** 写输出流的迭代器。

**输出迭代器**（output iterator）可以写元素，但不必具有读元素能力的迭代器。

**谓词**（predicate） 返回可以转换为 bool 类型的值的函数。泛型算法通常用来检测元素。标准库使用的谓词是一元（接受一个参数）或二元（接受两个参数）的。

**随机访问迭代器**（random-access iterator）支持双向迭代器的所有操作再加上比较迭代器值的关系运算符、下标运算符和迭代器上的算术运算，因此支持随机访问元素。

**ref** 标准库函数，从一个指向不能拷贝的类型的对象的引用生成一个可拷贝的对象。

**反向迭代器**（reverse iterator） 在序列中反向移动的迭代器。这些迭代器交换了++和--的含义。

**流迭代器**（stream iterator） 可以绑定到一个流的迭代器。

**一元谓词**（unary predicate） 接受一个参数的谓词。

done!

# 第 11 章
# 关联容器

## 内容



关联容器和顺序容器有着根本的不同：关联容器中的元素是按关键字来保存和访问的。与之相对，顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。

虽然关联容器的很多行为与顺序容器相同，但其不同之处反映了关键字的作用。

420 关联容器支持高效的关键字查找和访问。两个主要的**关联容器**（associative-container）类型是 **map** 和 **set**。map 中的元素是一些关键字-值（key-value）对：关键字起到索引的作用，值则表示与索引相关联的数据。set 中每个元素只包含一个关键字；set 支持高效的关键字查询操作——检查一个给定关键字是否在 set 中。例如，在某些文本处理过程中，可以用一个 set 来保存想要忽略的单词。字典则是一个很好的使用 map 的例子：可以将单词作为关键字，将单词释义作为值。

标准库提供 8 个关联容器，如表 11.1 所示。这 8 个容器间的不同体现在三个维度上：每个容器（1）或者是一个 set，或者是一个 map；（2）或者要求不重复的关键字，或者允许重复关键字；（3）按顺序保存元素，或无序保存。允许重复关键字的容器的名字中都包含单词 multi；不保持关键字按顺序存储的容器的名字都以单词 unordered 开头。因此一个 unordered_multi_set 是一个允许重复关键字，元素无序保存的集合，而一个 set 则是一个要求不重复关键字，有序存储的集合。无序容器使用哈希函数来组织元素，我们将在 11.4 节（第 394 页）中详细介绍有关哈希函数的更多内容。

类型 map 和 **multimap** 定义在头文件 map 中；set 和 **multiset** 定义在头文件 set 中；无序容器则定义在头文件 unordered_map 和 unordered_set 中。

**表 11.1：关联容器类型**

| | |
|---|---|
| **按关键字有序保存元素** | |
| map | 关联数组；保存关键字-值对 |
| set | 关键字即值，即只保存关键字的容器 |
| multimap | 关键字可重复出现的 map |
| multiset | 关键字可重复出现的 set |
| **无序集合** | |
| unordered_map | 用哈希函数组织的 map |
| unordered_set | 用哈希函数组织的 set |
| unordered_multimap | 哈希组织的 map；关键字可以重复出现 |
| unordered_multiset | 哈希组织的 set；关键字可以重复出现 |

## 11.1 使用关联容器

虽然大多数程序员都熟悉诸如 vector 和 list 这样的数据结构，但他们中很多人从未使用过关联数据结构。在学习标准库关联容器类型的详细内容之前，我们首先来看一个如何使用这类容器的例子，这对后续学习很有帮助。

map 是关键字-值对的集合。例如，可以将一个人的名字作为关键字，将其电话号码作为值。我们称这样的数据结构为“将名字映射到电话号码”。map 类型通常被称为**关联数组**（associative array）。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。
421 我们通过一个关键字而不是位置来查找值。给定一个名字到电话号码的 map，我们可以使用一个人的名字作为下标来获取此人的电话号码。

与之相对，set 就是关键字的简单集合。当只是想知道一个值是否存在时，set 是最有用的。例如，一个企业可以定义一个名为 bad_checks 的 set 来保存那些曾经开过空头支票的人的名字。在接受一张支票之前，可以查询 bad_checks 来检查顾客的名字是否在其中。

### 使用 map

一个经典的使用关联数组的例子是单词计数程序：

```
// 统计每个单词在输入中出现的次数
map<string, size_t> word_count;  // string 到 size_t 的空 map
string word;
while (cin >> word)
    ++word_count[word];          // 提取 word 的计数器并将其加 1
for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

此程序读取输入，报告每个单词出现多少次。

类似顺序容器，关联容器也是模板（参见 3.3 节，第 86 页）。为了定义一个 map，我们必须指定关键字和值的类型。在此程序中，map 保存的每个元素中，关键字是 string 类型，值是 size_t 类型（参见 3.5.2 节，第 103 页）。当对 word_count 进行下标操作时，我们使用一个 string 作为下标，获得与此 string 相关联的 size_t 类型的计数器。

while 循环每次从标准输入读取一个单词。它使用每个单词对 word_count 进行下标操作。如果 word 还未在 map 中，下标运算符会创建一个新元素，其关键字为 word，值为 0。不管元素是否是新创建的，我们将其值加 1。

一旦读取完所有输入，范围 for 语句（参见 3.2.3 节，第 81 页）就会遍历 map，打印每个单词和对应的计数器。当从 map 中提取一个元素时，会得到一个 pair 类型的对象，我们将在 11.2.3 节（第 379 页）介绍它。简单来说，pair 是一个模板类型，保存两个名为 first 和 second 的（公有）数据成员。map 所使用的 pair 用 first 成员保存关键字，用 second 成员保存对应的值。因此，输出语句的效果是打印每个单词及其关联的计数器。

如果我们对本节第一段中的文本（指英文版中的文本）运行这个程序，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
```

### 使用 set

422

上一个示例程序的一个合理扩展是：忽略常见单词，如"the"、"and"、"or"等。我们可以使用 set 保存想忽略的单词，只对不在集合中的单词统计出现次数：

```
// 统计输入中每个单词出现的次数
map<string, size_t> word_count; // string 到 size_t 的空 map
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an", "a"};
string word;
while (cin >> word)
    // 只统计不在 exclude 中的单词
    if (exclude.find(word) == exclude.end())
        ++word_count[word];     // 获取并递增 word 的计数器
```

与其他容器类似，set 也是模板。为了定义一个 set，必须指定其元素类型，本例中是 string。与顺序容器类似，可以对一个关联容器的元素进行列表初始化（参见 9.2.4 节，第 300 页）。集合 exclude 中保存了 12 个我们想忽略的单词。

此程序与前一个程序的重要不同是，在统计每个单词出现次数之前，我们检查单词是否在忽略集合中，这是在 if 语句中完成的：

```
// 只统计不在 exclude 中的单词
if (exclude.find(word) == exclude.end())
```

find 调用返回一个迭代器。如果给定关键字在 set 中，迭代器指向该关键字。否则，find 返回尾后迭代器。在此程序中，仅当 word 不在 exclude 中时我们才更新 word 的计数器。

如果用此程序处理与之前相同的输入，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
```

### 11.1 节练习

**练习 11.1**：描述 map 和 vector 的不同。

**练习 11.2**：分别给出最适合使用 list、vector、deque、map 以及 set 的例子。

**练习 11.3**：编写你自己的单词计数程序。

**练习 11.4**：扩展你的程序，忽略大小写和标点。例如，"example. "、"example,"和"Example"应该递增相同的计数器。

423

## 11.2 关联容器概述

关联容器（有序的和无序的）都支持 9.2 节（第 294 页）中介绍的普通容器操作（列于表 9.2，第 295 页）。关联容器不支持顺序容器的位置相关的操作，例如 push_front 或 push_back。原因是关联容器中元素是根据关键字存储的，这些操作对关联容器没有意义。而且，关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

除了与顺序容器相同的操作之外，关联容器还支持一些顺序容器不支持的操作（参见表 11.7，第 388 页）和类型别名（参见表 11.3，第 381 页）。此外，无序容器还提供一些用来调整哈希性能的操作，我们将在 11.4 节（第 394 页）中介绍。

关联容器的迭代器都是双向的（参见 10.5.1 节，第 365 页）。

### 11.2.1 定义关联容器

如前所示，当定义一个 map 时，必须既指明关键字类型又指明值类型；而定义一个 set 时，只需指明关键字类型，因为 set 中没有值。每个关联容器都定义了一个默认构

造函数，它创建一个指定类型的空容器。我们也可以将关联容器初始化为另一个同类型容器的拷贝，或是从一个值范围来初始化关联容器，只要这些值可以转化为容器所需类型就可以。在新标准下，我们也可以对关联容器进行值初始化：

C++ 11

```
map<string, size_t> word_count; // 空容器
// 列表初始化
set<string> exclude = {"the", "but", "and", "or", "an", "a",
                       "The", "But", "And", "Or", "An", "A"};
// 三个元素；authors 将姓映射为名
map<string, string> authors = { {"Joyce", "James"},
                                {"Austen", "Jane"},
                                {"Dickens", "Charles"} };
```

与以往一样，初始化器必须能转换为容器中元素的类型。对于 set，元素类型就是关键字类型。

当初始化一个 map 时，必须提供关键字类型和值类型。我们将每个关键字-值对包围在花括号中：

{*key*, *value*}

来指出它们一起构成了 map 中的一个元素。在每个花括号中，关键字是第一个元素，值是第二个。因此，authors 将姓映射到名，初始化后它包含三个元素。

### 初始化 multimap 或 multiset

一个 map 或 set 中的关键字必须是唯一的，即，对于一个给定的关键字，只能有一个元素的关键字等于它。容器 multimap 和 multiset 没有此限制，它们都允许多个元素具有相同的关键字。例如，在我们用来统计单词数量的 map 中，每个单词只能有一个元素。另一方面，在一个词典中，一个特定单词则可具有多个与之关联的词义。

424

下面的例子展示了具有唯一关键字的容器与允许重复关键字的容器之间的区别。首先，我们将创建一个名为 ivec 的保存 int 的 vector，它包含 20 个元素：0 到 9 每个整数有两个拷贝。我们将使用此 vector 初始化一个 set 和一个 multiset：

```
// 定义一个有 20 个元素的 vector，保存 0 到 9 每个整数的两个拷贝
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // 每个数重复保存一次
}
// iset 包含来自 ivec 的不重复的元素；miset 包含所有 20 个元素
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl;      // 打印出 20
cout << iset.size() << endl;      // 打印出 10
cout << miset.size() << endl;     // 打印出 20
```

即使我们用整个 ivec 容器来初始化 iset，它也只含有 10 个元素：对应 ivec 中每个不同的元素。另一方面，miset 有 20 个元素，与 ivec 中的元素数量一样多。

### 11.2.1 节练习

**练习 11.5**：解释 map 和 set 的区别。你如何选择使用哪个？

**练习 11.6**：解释 set 和 list 的区别。你如何选择使用哪个？

**练习 11.7**：定义一个 map，关键字是家庭的姓，值是一个 vector，保存家中孩子（们）的名。编写代码，实现添加新的家庭以及向已有家庭中添加新的孩子。

**练习 11.8**：编写一个程序，在一个 vector 而不是一个 set 中保存不重复的单词。使用 set 的优点是什么？

## 11.2.2 关键字类型的要求

关联容器对其关键字类型有一些限制。对于无序容器中关键字的要求，我们将在 11.4 节（第 396 页）中介绍。对于有序容器——map、multimap、set 以及 multiset，关键字类型必须定义元素比较的方法。默认情况下，标准库使用关键字类型的<运算符来比较两个关键字。在集合类型中，关键字类型就是元素类型；在映射类型中，关键字类型是元素的第一部分的类型。因此，11.2 节（第 377 页）中 word_count 的关键字类型是 string。类似的，exclude 的关键字类型也是 string。

425

传递给排序算法的可调用对象（参见 10.3.1 节，第 344 页）必须满足与关联容器中关键字一样的类型要求。

### 有序容器的关键字类型

可以向一个算法提供我们自己定义的比较操作（参见 10.3 节，第 344 页），与之类似，也可以提供自己定义的操作来代替关键字上的<运算符。所提供的操作必须在关键字类型上定义一个**严格弱序**（strict weak ordering）。可以将严格弱序看作“小于等于”，虽然实际定义的操作可能是一个复杂的函数。无论我们怎样定义比较函数，它必须具备如下基本性质：

- 两个关键字不能同时“小于等于”对方；如果 k1“小于等于”k2，那么 k2 绝不能“小于等于”k1。
- 如果 k1“小于等于”k2，且 k2“小于等于”k3，那么 k1 必须“小于等于”k3。
- 如果存在两个关键字，任何一个都不“小于等于”另一个，那么我们称这两个关键字是“等价”的。如果 k1“等价于”k2，且 k2“等价于”k3，那么 k1 必须“等价于”k3。

如果两个关键字是等价的（即，任何一个都不“小于等于”另一个），那么容器将它们视作相等来处理。当用作 map 的关键字时，只能有一个元素与这两个关键字关联，我们可以用两者中任意一个来访问对应的值。

在实际编程中，重要的是，如果一个类型定义了“行为正常”的<运算符，则它可以用作关键字类型。

### 使用关键字类型的比较函数

用来组织一个容器中元素的操作的类型也是该容器类型的一部分。为了指定使用自定义的操作，必须在定义关联容器类型时提供此操作的类型。如前所述，用尖括号指出要定

义哪种类型的容器，自定义的操作类型必须在尖括号中紧跟着元素类型给出。

在尖括号中出现的每个类型，就仅仅是一个类型而已。当我们创建一个容器（对象）时，才会以构造函数参数的形式提供真正的比较操作（其类型必须与在尖括号中指定的类型相吻合）。

例如，我们不能直接定义一个 Sales_data 的 multiset，因为 Sales_data 没有<运算符。但是，可以用 10.3.1 节练习（第 345 页）中的 compareIsbn 函数来定义一个 multiset。此函数在 Sales_data 对象的 ISBN 成员上定义了一个严格弱序。函数 compareIsbn 应该像下面这样定义

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

426

为了使用自己定义的操作，在定义 multiset 时我们必须提供两个类型：关键字类型 Sales_data，以及比较操作类型——应该是一种函数指针类型（参见 6.7 节，第 221 页），可以指向 compareIsbn。当定义此容器类型的对象时，需要提供想要使用的操作的指针。在本例中，我们提供一个指向 compareIsbn 的指针：

```
// bookstore 中多条记录可以有相同的 ISBN
// bookstore 中的元素以 ISBN 的顺序进行排列
multiset<Sales_data, decltype(compareIsbn)*>
    bookstore(compareIsbn);
```

此处，我们使用 decltype 来指出自定义操作的类型。记住，当用 decltype 来获得一个函数指针类型时，必须加上一个*来指出我们要使用一个给定函数类型的指针（参见 6.7 节，第 223 页）。用 compareIsbn 来初始化 bookstore 对象，这表示当我们向 bookstore 添加元素时，通过调用 compareIsbn 来为这些元素排序。即，bookstore 中的元素将按它们的 ISBN 成员的值排序。可以用 compareIsbn 代替&compareIsbn 作为构造函数的参数，因为当我们使用一个函数的名字时，在需要的情况下它会自动转化为一个指针（参见 6.7 节，第 221 页）。当然，使用&compareIsbn 的效果也是一样的。

### 11.2.2 节练习

**练习 11.9**：定义一个 map，将单词与一个行号的 list 关联，list 中保存的是单词所出现的行号。

**练习 11.10**：可以定义一个 vector<int>::iterator 到 int 的 map 吗？list<int>::iterator 到 int 的 map 呢？对于两种情况，如果不能，解释为什么。

**练习 11.11**：不使用 decltype 重新定义 bookstore。

### 11.2.3 pair 类型

在介绍关联容器操作之前，我们需要了解名为 **pair** 的标准库类型，它定义在头文件 utility 中。

一个 pair 保存两个数据成员。类似容器，pair 是一个用来生成特定类型的模板。当创建一个 pair 时，我们必须提供两个类型名，pair 的数据成员将具有对应的类型。两个类型不要求一样：

```
pair<string, string> anon;          // 保存两个 string
pair<string, size_t> word_count;    // 保存一个 string 和一个 size_t
pair<string, vector<int>> line;     // 保存 string 和 vector<int>
```

427 pair 的默认构造函数对数据成员进行值初始化（参见 3.3.1 节，第 88 页）。因此，anon 是一个包含两个空 string 的 pair，line 保存一个空 string 和一个空 vector。word_count 中的 size_t 成员值为 0，而 string 成员被初始化为空。

我们也可以为每个成员提供初始化器：

```
pair<string, string> author{"James", "Joyce"};
```

这条语句创建一个名为 author 的 pair，两个成员被初始化为"James"和"Joyce"。

与其他标准库类型不同，pair 的数据成员是 public 的（参见 7.2 节，第 240 页）。两个成员分别命名为 first 和 second。我们用普通的成员访问符号（参见 1.5.2 节，第 20 页）来访问它们，例如，在第 375 页的单词计数程序的输出语句中我们就是这么做的：

```
// 打印结果
cout << w.first << " occurs " << w.second
     << ((w.second > 1) ? " times" : " time") << endl;
```

此处，w 是指向 map 中某个元素的引用。map 的元素是 pair。在这条语句中，我们首先打印关键字——元素的 first 成员，接着打印计数器——second 成员。标准库只定义了有限的几个 pair 操作，表 11.2 列出了这些操作。

**表 11.2：pair 上的操作**

| | |
|---|---|
| `pair<T1, T2> p;` | p 是一个 pair，两个类型分别为 T1 和 T2 的成员都进行了值初始化（参见 3.3.1 节，第 88 页） |
| `pair<T1, T2> p(v1, v2)` | p 是一个成员类型为 T1 和 T2 的 pair；first 和 second 成员分别用 v1 和 v2 进行初始化 |
| `pair<T1,T2>p = {v1,v2};` | 等价于 p (v1,v2) |
| `make_pair(v1, v2)` | 返回一个用 v1 和 v2 初始化的 pair。pair 的类型从 v1 和 v2 的类型推断出来 |
| `p.first` | 返回 p 的名为 first 的（公有）数据成员 |
| `p.second` | 返回 p 的名为 second 的（公有）数据成员 |
| `p1` *relop* `p2` | 关系运算符（<、>、<=、>=）按字典序定义：例如，当 p1.first < p2.first 或!(p2.first < p1.first) && p1.second < p2.second 成立时，p1 < p2 为 true。关系运算利用元素的< 运算符来实现 |
| `p1 == p2` | 当 first 和 second 成员分别相等时，两个 pair 相等。相等性判断利用元素的==运算符实现 |
| `p1 != p2` | |

### 创建 pair 对象的函数

C++11 想象有一个函数需要返回一个 pair。在新标准下，我们可以对返回值进行列表初始化（参见 6.3.2 节，第 203 页）

428
```
pair<string, int>
process(vector<string> &v)
{
    // 处理 v
```

```
    if (!v.empty())
        return {v.back(), v.back().size()}; // 列表初始化
    else
        return pair<string, int>(); // 隐式构造返回值
}
```

若 v 不为空，我们返回一个由 v 中最后一个 string 及其大小组成的 pair。否则，隐式构造一个空 pair，并返回它。

在较早的 C++ 版本中，不允许用花括号包围的初始化器来返回 pair 这种类型的对象，必须显式构造返回值：

```
if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());
```

我们还可以用 make_pair 来生成 pair 对象，pair 的两个类型来自于 make_pair 的参数：

```
if (!v.empty())
    return make_pair(v.back(), v.back().size());
```

### 11.2.3 节练习

**练习 11.12**：编写程序，读入 string 和 int 的序列，将每个 string 和 int 存入一个 pair 中，pair 保存在一个 vector 中。

**练习 11.13**：在上一题的程序中，至少有三种创建 pair 的方法。编写此程序的三个版本，分别采用不同的方法创建 pair。解释你认为哪种形式最易于编写和理解，为什么？

**练习 11.14**：扩展你在 11.2.1 节练习（第 378 页）中编写的孩子姓到名的 map，添加一个 pair 的 vector，保存孩子的名和生日。

## 11.3 关联容器操作

除了表 9.2（第 295 页）中列出的类型，关联容器还定义了表 11.3 中列出的类型。这些类型表示容器关键字和值的类型。

**表 11.3：关联容器额外的类型别名**

| | |
|---|---|
| key_type | 此容器类型的关键字类型 |
| mapped_type | 每个关键字关联的类型；只适用于 map |
| value_type | 对于 set，与 key_type 相同<br>对于 map，为 pair<const key_type, mapped_type> |

对于 set 类型，**key_type** 和 **value_type** 是一样的；set 中保存的值就是关键字。在一个 map 中，元素是关键字-值对。即，每个元素是一个 pair 对象，包含一个关键字和一个关联的值。由于我们不能改变一个元素的关键字，因此这些 pair 的关键字部分是 const 的：

```
set<string>::value_type v1;       // v1 是一个 string
set<string>::key_type v2;         // v2 是一个 string
map<string, int>::value_type v3; // v3 是一个 pair<const string, int>
map<string, int>::key_type v4;    // v4 是一个 string
map<string, int>::mapped_type v5;// v5 是一个 int
```

429

与顺序容器一样（参见 9.2.2 节，第 297 页），我们使用作用域运算符来提取一个类型的成员——例如，map<string, int>::key_type。

只有 map 类型（unordered_map、unordered_multimap、multimap 和 map）才定义了 **mapped_type**。

### 11.3.1 关联容器迭代器

当解引用一个关联容器迭代器时，我们会得到一个类型为容器的 value_type 的值的引用。对 map 而言，value_type 是一个 pair 类型，其 first 成员保存 const 的关键字，second 成员保存值：

```
// 获得指向 word_count 中一个元素的迭代器
auto map_it = word_count.begin();
// *map_it 是指向一个 pair<const string, size_t>对象的引用
cout << map_it->first;           // 打印此元素的关键字
cout << " " << map_it->second;   // 打印此元素的值
map_it->first = "new key";       // 错误：关键字是 const 的
++map_it->second; // 正确：我们可以通过迭代器改变元素
```

必须记住，一个 map 的 value_type 是一个 pair，我们可以改变 pair 的值，但不能改变关键字成员的值。

#### set 的迭代器是 const 的

虽然 set 类型同时定义了 iterator 和 const_iterator 类型，但两种类型都只允许只读访问 set 中的元素。与不能改变一个 map 元素的关键字一样，一个 set 中的关键字也是 const 的。可以用一个 set 迭代器来读取元素的值，但不能修改：

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;              // 错误：set 中的关键字是只读的
    cout << *set_it << endl; // 正确：可以读关键字
}
```

430

#### 遍历关联容器

map 和 set 类型都支持表 9.2（第 295 页）中的 begin 和 end 操作。与往常一样，我们可以用这些函数获取迭代器，然后用迭代器来遍历容器。例如，我们可以编写一个循环来打印第 375 页中单词计数程序的结果，如下所示：

```
// 获得一个指向首元素的迭代器
auto map_it = word_count.cbegin();
// 比较当前迭代器和尾后迭代器
while (map_it != word_count.cend()) {
    // 解引用迭代器，打印关键字-值对
    cout << map_it->first << " occurs "
         << map_it->second << " times" << endl;
    ++map_it; // 递增迭代器，移动到下一个元素
}
```

while 的循环条件和循环中的迭代器递增操作看起来很像我们之前编写的打印一个 vector