第10章 泛型算法

10.1 概述

1. 大多数算法都定义在头文件 `algorithm` 中。标准库还在头文件 `numeric` 中定义了一组数值泛型算法。
2. 一般情况下，泛型算法不直接操作容器，而是遍历由两个迭代器指定的一个元素范围来进行操作。通常情况下算法遍历范围，对其中每个元素进行一些处理：

```
std::vector<int> vec = { 0, 1, 2 };
int val = 1;
auto it = find(vec.cbegin(), vec.cend(), val);
```

3. 类似的，由于指针就像内置数组上的迭代器一样，我们可以用 `find` 在数组中查找值：

```
int ia[] = { 0, 1, 2 };
int val = 1;
int * it = find(begin(ia), end(ia), val);
```

4. 迭代器令算法不依赖于容器，但是算法依赖于元素类型的操作。
5. 泛型算法运行于迭代器之上而不会执行容器操作的特性带来了一个令人惊讶但非常必要的编程假定：*算法永远不会改变底层容器的大小*。算法可能改变容器中保存的元素的值，也可能在容器之移动元素，但永远不会直接添加或删除元素。
6. 标准库定义了一类特殊的迭代器称为 *插入器(inserter)*，当给这类迭代器赋值时，它们会在底层的容器上执行插入操作。因此当一个算法操作插入器时，插入器可以完成向容器添加元素的效果，但是算法本身永远不会做这样的操作。

10.2 初识泛型算法

10.2.1 只读算法

1. `accumulate` 将第三个参数作为求和起点，这蕴含这一个编程假定：将元素类型加到和的类型上的操作必须是可行的。即序列中的元素的类型必须于第三个参数匹配，或者能够转换为第三个参数的类型。

```
// vec 中的元素可以时 int，或者 double、long double 或者任何其他可以加到 int 上的类型。
int sum = accumulate(vec.cbegin(), vec.cend(), 0);

// 通过第三个参数显示地创建了一个 string。
string sum = accumulate(v.cbegin(), v.cend(), string(""));

// 将空串当作一个字符串字面值传递给第三个参数是不可以的。
// 因为此时保存和的对象类型将是 const char*，但 const char* 没有 +运算符。
string sum = accumulate(v.cbegin(), v.cend(), "");
```

2. `equal` 算法利用迭代器完成操作，因此可以通过调用 `equal` 来比较两个不同类型的容器中的元素（例如比价 `vector` 和 `list` 两个容器中的元素）。而且，容器中的元素类型也不必一样，只要能用 `==` 运算符来比较这两个元素类型即可（例如比较 `int` 和 `double` 类型）。
3. 那些只接受一个单一迭代器来表示第二个序列的算法，都假定第二个序列至少于第一个序列一样长。

```
// 第三个参数接受一个单一的迭代器，roster2中的元素数目应至少与roster1一样多
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

10.2.2 写容器元素的算法

1. 一些算法从两个序列中读取元素。构成这两个序列的元素可以来自于不同类型的容器。而且两个序列中元素的类型也不要求严格匹配，算法要求的只是能够比较两个序列中的元素。
2. 向目的的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素。
3. 一种保证算法有足够的元素空间来容纳输出数据的方法是使用 *插入迭代器（insert iterator）*。它是一种向容器中添加元素的迭代器，当我们通过它来赋值时，一个与赋值号右侧值相等的元素被添加到容器中。

```
vector<int> vec;

// 使用普通迭代器
fill_n(vec.begin(), 10, 0); // 错误：修改 vec 中的10个（不存在）元素

// 使用插入迭代器
auto it = back_inserter(vec);
*it = 42;                   // 等价于 vec.push_back(42);
fill_n(it, 10, 0);          // 给vec的尾部添加10个0（每次赋值都会在vec上调用push_back）
```

4. 多个算法都提供所谓的 “拷贝” 版本。这些算法计算新元素的值，但是不会将它们放置在输入序列的末尾，而是创建一个新序列保存这些结果。

```
replace(ilst.begin(), ilst.end(), 0, 42);   // 将ilst中所有0替换成42

// 如果希望保留原序列不变，可以调用其“拷贝”版本
// 此调用后ilst并未改变，ivec包含ilst的一份拷贝
// 不过原来在ilst中值为0的元素在ivec中都变为了42
replace(ilst.cbegin(), ilst.cend(), back_inserter(ivec), 0, 42);
```

10.3 定制操作

10.3.1 向算法传递函数

1. *谓词* 是一个可调用的表达式，其返回结果是一个能用作条件的值。*一元谓词 (unary predicate)* 只接受单一参数，*二元谓词 (binary predicate)* 意味着它们有两个参数。

10.3.2 *lambda* 表达式

1. 一个 `lambda` 表达式表示一个可调用的代码单元，我们可以将其理解为一个未命名的内联函数。

```
// 与普通函数不同，lambda必须使用尾置返回
auto func = [capture list](parameter list) -> return type { fuction body }
```

2. 我们可以忽略参数列表和返回类型，但是必须永远包含捕获列表和函数体。
3. 如果 `lambda` 的函数体包含任何单一 `return` 语句之外的内容，且未指定返回类型，则返回 `void` 。
4. 与普通函数不同， `lambda` 不能有默认参数，一个 `lambda` 调用的实参数目永远与形参数目相等。

```
// 忽略括号和参数列表等价于指定一个空参数列表
// 忽略返回类型，lambda根据函数体中的代码推断出返回类型
auto func = [] { return 42; }   // 返回类型为int

// 编译器推断这个版本返回类型为void，但是它返回了一个int类型
auto func1 = [] { int a = 0; return a; }

// 使用了尾置返回类型，可以返回一个int类型
auto func2= [] -> int { int a = 0; return a; }
```

5. 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。

10.3.3 *lambda* 捕获和返回

1. 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。
2. 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。
3. 引用捕获和返回引用有着相同的问题和限制。如果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。
4. `lambda` 捕获的都是局部变量，这些变量在函数结束就不复存在了。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。
5. 如果函数返回一个 `lambda`，则于函数不能返回一个局部变量的引用类似，此 `lambda` 不能包含引用捕获。
6. 当混合使用隐式捕获和显示捕获时，显示捕获的变量必须使用于隐式捕获不同的方式。
7. 默认情况下，对于一个值拷贝的变量， `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` 。

```
int v1 = 42;
auto f = [v1]() mutable { return ++v1; };
```

10.3.4 参数绑定

1. 可以将 `bind` 函数看作一个通用的函数适配器，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。

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

2. `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”，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的位置。
3. 使用 `_n` 之前要先定义命名空间： `using namespace std::placeholders;`

```
bool check_size(const string& s, string::size_type sz) { return s.size() >= sz; }
auto check = bind(check_size, _1, 6);
string s = "hello";
bool b1 = check(s);     // 会调用 check_size(s, 6)
```

4. 使用 `bind` 可以将原来基于 `lambda` 的find_if调用替换为使用 `check_size` 的版本。

```
auto wc = find_if(words.begin(), word.end(), [sz](const string& a){...});
auto wc = find_if(words.begin(), word.end(), bind(check_size, _1, sz));
```

5. 可以使用 `bind` 绑定给定可调用对象中的参数或重新安排其顺序。

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

// 等价调用
g(X, Y);
f(a, b, Y, c, X);
```

6. 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(word.begin(), word.end(), bind(print, os, _1, ' '));   // 错误，不能拷贝ostream
for_each(word.begin(), word.end(), bind(print, ref(os), _1, ' '));
```

10.4 再探迭代器

1. *插入迭代器（insert iterator）*：被绑定到一个容器上，用来向容器插入元素。
2. *流迭代器（stream iterator）*：被绑定到输入或输出流上，可用来遍历所关联的IO流。
3. *反向迭代器（reverse iterator）*：向后而不向前移动的迭代器，除了 `forward_list` 之外的标准库容器都有反向迭代器。
4. *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。

10.4.1 插入迭代器

1. 插入器是一种迭代器适配器，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。
2. 插入器有三种： `back_inserter` 、 `front_inserter` 和 `inserter` 。

10.4.2 *iostream* 迭代器

1. `istream_iterator` 读取输入流， `ostream_iterator` 向一个输出流写数据。我们可以为任何定义了输入运算符（>>）或输出运算符（<<）的类型创建 `istream_iterator` 或 `ostream_iterator` 对象。

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

2. `istream_iterator 允许使用 *懒惰求值*。
3. 对于 `ostream_iterator<T> out(os)` ， `*out, ++out, out++` 这些运算符是存在的，但是不对 `out` 做任何事情。每个运算符都返回 `out` 。

10.4.3 反向迭代器

1. 不能从一个 `forward_list` 或一个流迭代器创建反向迭代器。
2. 通过调用 `reverse_iterator` 的 `base` 成员函数返回其对应的普通迭代器（迭代器指向的位置会往后移动一个位置）。
3. 普通迭代器与反向迭代器的关系反映了左闭合区间。
4. 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，结果迭代器于原迭代器指向的并不是相同的元素，而是相邻的位置。

10.5 泛型算法结构

1. 输入迭代器（input iterator）、输出迭代器（output iterator）、前向迭代器（forward iterator）、双向迭代器（bidirectional iterator）、随机访问迭代器（random-access iterator）。

10.5.1 5类迭代器

1. 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。
2. 对于一个 *输入迭代器*， `*it++` 保证是有效的，但是递增它可能导致所有其他指向流的迭代器失效。其结果是不能保证 *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。
3. 只能向一个 *输出迭代器* 赋值一次，*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*。
4. *前向迭代器* 支持所有的输入和输出迭代器的操作，而且可以多次读写同一个元素，算法可以对序列进行多遍扫描。
5. *随机访问迭代器* 提供在常量时间内访问序列中任意元素的能力。

10.5.2 算法形参模式

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

1. `dest` 参数是一个表示算法可以写入的目的位置的迭代器。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。
2. 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

1. 一些算法使用重载形式传递一个谓词

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

2. _if 版本的算法：接受谓词参数的算法都有附加的 _if 前缀。

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

3. 区分拷贝元素的版本和不拷贝的版本。写到额外目的空间的算法都在名字后面附加一个 _copy

```
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
```

4. 一些算法同时提供 _copy 和 _if 版本。

```
// 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_if(v1.begin(), v1.end(), back_inserter(v2),
            [](int i){ return i % 2; });
```

10.6 特定容器算法

1. 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。
2. 链表特有的操作会改变容器。