第十章：泛型算法

1. 概述

标准库并未给每个容器都定义成员函数来实现这些操作，而是定义了一组泛型算法(generic algorithm)：称它们为“算法”，是因为它们实现了一些经典算法的公共接口，如排序和搜索；称它们是“泛型的”，是因为它们可以用于不同类型的元素和多种容器类型(不仅包括标准库类型，如 vector 或 list，还包括内置的数组类型)，以及我们将看到的，还能用于其他类型的序列。

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。

一般情况下，这些算法并不直接操作容器，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。

例子：

```
vector<int> v{1, 2, 3, 4, 5};
int target;
cin >> target;
auto result = find(v.begin(), v.end(), target);
if(result==v.end())//判断是否找到
cout << "no" << endl;
else
cout <<target<<"下标为"<< result - v.begin() << endl;
```

find 前两个参数用迭代器（指针）来表示范围，第三个参数表示要搜索的值。如果找到则返回指向该元素的迭代器（如果是内置数组则返回指针），否则返回第二个参数来表示失败.

2. 算法基础

2.1. 算法特性

- 不依赖与容器
  由于泛型算法本身是通过迭代器来操作的，而且大多数算法都需要两个迭代器来表示范围
- 依赖于元素类型
  因此泛型算法不依赖于容器类型而依赖于元素类型（有的容器可能没有定义<等符号）
- 算法不会改变容器
  泛型算法可能会移动或者改变元素但是不会添加或者删除元素
  由于存在插入迭代器（inserter），当用插入迭代器给泛型算法时可以做到算法结束后插入元素，但是算法本身不会这么做
- 假定可以运算
  泛型算法假定输入的类型是可以运算的，就算会有误差。例如 int double long 三者可以加在一起，因为三者都定义了+，甚至可以用 accumulate 把 string 连载一起，只不过初始值为 string("")而不是” “因为 const char*没有定义+
- 不检查写
  例如 fill_n(pos,n,val)从 pos 开始用 n 个 val 代替后面的值，但是 fill_n 不会检查 n 是否超出 pos 所指的序列的范围。

2.2. 只读算法

一些算法只会读取其输入范围内的元素，而从不改变元素。例如 find 和 accumulate（一个用于数组求和的算法，前两个参数用迭代器表示范围，最后一个参数用来表示求和的初始值，返回求和结果）

2.3. 改变值算法

一些算法可以将新值赋给序列中的元素，但是必须确保序列大小至少要大于等于我们要求写入的数目。有些算法需要在运行时输入范围，但是实际写入时最多写到序列结束，因此此类算法还算安全

例如 fill(first,last,val)，用 val 代替 first 到 last 中的值

2.4. 双序列算法

此类算法接受的迭代器数目较多（3 个或者 4 个），用于两个序列的比较等操作

例如 equal(first1,last1,first2) 会前两个参数表示第一个序列的范围，第二个参数表示第二个序列的开始。该算法会把第一个序列的每一个元素与第二个序列对应的元素比较，如果全部相等返回 TRUE，否则返回 FALSE

注意：用一个单一迭代器表示第二个序列的算法都假定第一个序列至少与第一个一样长。确保算法不会试图访问第二个序列中不存在的元素是程序员的责任。例如,算法 equal 会将其第一个序列中的每个元素与第二个序列中的对应元素进行比较。如果第二个序列是第一个序列的一个子集,则程序会产生一个严重错误——equal 会试图访问第二个序列中末尾之后(不存在)的元素。

2.5. back_inserter

一种保证算法有足够元素空间来容纳输出数据的方法是使用插入迭代器(insert_terator)。插入迭代器是一种向容器中添加元素的迭代器。

当我们通过一个插入迭代器赋值时，一个与赋值号右侧值相等的元素被添加到容器中。

我们现在将使用 back_inserter,它是定义在头文件 iterator 中的一个函数。

```
vector<int> v;
auto i=back_inserter(v);
for (int i = 0; i < 10; ++i)
  *i1 = i;//每次循环都在容器末尾添加值为i的元素
for (auto &i : v)
  cout << i << " ";
cout << endl;
```

back_inserter 常常用于创建迭代器来作为泛型算法的参数

```
fill_n(back_inserter(v),10,0)//由于插入迭代器的特殊性，该算法可以在容器后加上10个0
```

2.6. 拷贝算法

`copy(begin1,end1,begin2)`

将迭代器（指针） begin1 到 end1 所指向的元素复制到 begin2 开始的位置，返回目标序列拷贝后最后一个元素的下一个迭代器(或者尾后迭代器)。其中 begin2 的序列长度大于等于输入序列的长度。最多复制到目标序列的末尾

```
vector<int> v{1, 2, 3, 4};
vector<int> x{0};
//最多复制到末尾
copy(v.begin(), v.end(), x.begin());//x:1
x={0,0,0,0, 5};
//result为指向5的迭代器
auto result=copy(v.begin(), v.end(), x.begin());//x:1 2 3 4
```

2.7 replace

`repalce(first,last,old_value,new_value)`

把 first 到 last 的 old_value 替换为 new_value,无返回值

2.8 sort

`sort(first,last)`

sort 算法依赖于元素定义的<把容器内的元素从小到大排序

2.9 unique

`unique(first,last)`

unique 即是独特的，该算法会把重复的元素"删除"，返回最后一个独特元素的下一个迭代器。实际上重复的元素没有被删除，只是被移到了序列最后面。如果不更新 end，range for 依然可以遍历到重复的元素。如果没有重复，则返回end

如果要擦除，可以使用一个变量储存 unique 返回的迭代器，然后调用 erase 从这个迭代器到 end 擦除

```
vector<int> v{1, 5, 1, 6, 2, 6, 8, 1};
sort(v.begin(), v.end()); // 1 1 1 2 5 6 6 8
auto end_unique = unique(v.begin(), v.end());//end_unique指向有序序列的尾后元素
v.erase(end_unique, v.end()); // 1 2 5 6
```