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曲算法的公共接口，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
//用erase擦除重复的元素
v.erase(end_unique, v.end()); // 1 2 5 6
```

3. 定制操作

很多算法都会比较输入序列中的元素。默认情况下，这类算法使用元素类型的<或--运算符完成比较。标准库还为这些算法定义了额外的版本，允许我们提供自己定义的操作来代替默认运算符。
例如 sort 就是需要<来进行排序，如果排序类型的自定义的类型，或者是没有定义<的类型，就需要重载 sort 默认的<

3.1. 向算法传递函数

运算符是一种返回 bool 值的函数，因此可以定义一种类似运算符的函数传递给算法来重载算法的默认行为。这个传递的参数称作**谓词**

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。标准库算法所使用的谓词分为两类: 一元谓词（unary predicate，意味着它们只接受单一参数）和二元谓词(binary predicate，意味着它们有两个参数)。接受谓词参数的算法对输入序列中的元素调用谓词。因此，元素类型

必须能转换为谓词的参数类型。
接受一个二元谓词参数的 sort 版本用这个谓词代替 < 来比较元素。我们提供给 sort 的谓词必须满足将在 11.2.2 节中所介绍的条件。当前，我们只需知道，此操作必须在输入序列中所有可能的元素值上定义一个一致的序。

```
bool isShorter(const string &s1,const string &s2)
{
  return s1.size()< s2.size();
}
//按长度排序单词
sort(word.begin(),word.end(),isShorter);
```

3.1.2. 稳定排序算法 stable_sort

`stable_sort(first, last, predicate)
为了保持相同长度的单词按字典序排列，可以使用 stable_sort 算法。这种稳定排序算法维持相等元素的原有顺序。

```
vector<string> word{"DD", "B", "CCC", "AA", "E"};
//DD在AA前面
stable_sort(word.begin(), word.end(), isShorter);
//B E DD AA CCC
```

3.1.3. find_if 算法

`find_if(first,last,predicate)`

find_if 算法来查找第一个具有特定大小的元素。find_if 算法接受一对迭代器，表示一个范围，第三个参数是一个一元谓词。find_if 算法对输入序列中的每个元素调用给定的这个谓词。它返回第一个使谓词返回非 0 值的元素，如果不存在这样的元素，则返回尾迭代器。

3.2. lambda 表达式

在以单词长度排序 word 时，我们希望给定任意长度也能排序 word，而不需要每次编写各种长度的判断是否超出给定长度的谓词。

如果一个对象可以使用调用符号()，则这个对象就是可调用对象，是可调用的。

例如函数或者函数指针，此外 lambda 表达式也可以调用的

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，因此也叫做匿名函数或者是闭包。与任何函数类似，一个 lambda 具有一个返回类型、一个参数列表和一个函数体。但与函数不同，lambda 可能定义在函数内部。一个 lambda 表达式具有如下形式

[capture list] (parameter list) ->return type

{

function body

}

- capture list : 捕获类型，是一个 lambda 所在函数中定义的局部变量的列表(通常为空)
- return type : 返回类型，必须尾置返回类型。如果省略返回类型，会根据 return 来推断，如果没有 return，则 void。**如果 return 语句在 if 的块中且省略返回类型，则可能出现无法自动推断返回类型的情况，此时就必须指定返回类型**
- parameter list : 参数列表，可以省略如果省略参数列表，则不接受参数。**形参和实参必须匹配，而且不能有默认实参(因此形参和实参数目必须匹配)**,
- function body : 函数体
- 没有函数名，如果要给这个 lambda 表达式命名，则用 auto 推断

```
auto f=[](return 42;);//定义一个lambda对象，不接受参数，返回42
cout<<f()<<endl;//调用f
```

2.3. 使用捕获

如果 lambda 要使用或者修改 lambda 作用域外的变量，就必须捕获，即在捕获列表中声明捕获变量，以 , 隔开不同的参数，然后在 lambda 表达式的函数体中就可以使用他们。未捕获的变量不能使用(报错: 未捕获)

捕获有值捕获和引用捕获两种，与函数的值传递和引用传递一样。

- 值捕获，就是只有一个名字。值捕获会拷贝一个变量，不能修改 lambda 作用域外的值，只能访问。如果尝试修改，则编译器会报错。
- 引用捕获，在捕获变量前加 **&**，如它创建一个引用指向被捕获的对象，可以修改 lambda 作用域外的值。**iostream 类的变量只能用引用捕获**
- 隐式捕获: 参数列表只有一个=(值捕获)或者&(引用捕获)，则是隐式捕获，默认捕获到的变

量都是指定的类型。也可以隐式显式混合捕获，但是隐式捕获必须在第一个

引用捕获与返回引用有看相同的回题和限制。如 lambda 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。

捕获的其他特性:

- 如果是在类中使用 lambda 表达式，可以用 this 捕获，捕获当前实例的指针
- 如果是在类中使用 lambda 表达式，可以用*this 捕获，捕获当前实例(C++17)
- 可以在捕获列表中定义新的变量并且初始化，该变量直接定义名字，不写类型(由 auto 自动推断)，作用域为 lambda 的函数体。(C++14)
- 参数列表的变量支持 auto 推断(C++14)
- 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。

```
[sz,, &sj](const string &a){ return a.size()>=sz}//第一个是引用捕获，第二个是值捕获
[&,=N,K=0](auto x)
{
  //按值捕获N，其他都是引用捕获
  //定义Lambda内部变量K
  //参数类型自动推断
  cout<<K<<" "<<x<<endl;
  return 0;
};
[=,&x]{return 0;};//除了x是引用捕获全部是值捕获
[&, =f]{return 0;};//除了f是值捕获全部是引用捕获
```

> 实际上在创建 lambda 表达式时，编译器会生成一个与 lambda 表达式对应的未命名的类，使用 auto 初始化 lambda 表达式时，名字就是从这个类中创建的对象
> 当向函数传递 lambda 表达式时，编译器会创建类和他的一个对象
> 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。

```
bool isShorter(const string &x, const string &y)
{
  //用于sort排序的谓词
    return x.size() < y.size();
}

std::string::size_type f(vector<string> &words, std::string::size_type sz)
{
    sort(words.begin(), words.end(), isShorter);//排序为了后面方便擦除重复的和计算长度大于sz的元素
    auto end_unique = unique(words.begin(), words.end());//获得被移到最后的重复元素的迭代器
    words.erase(x, words.end());//擦除
    //获得长度大于sz的第一个元素的迭代器
    auto wc = find_if(words.begin(), words.end(), [sz](const string &a) {
return a.size() >= sz; });//第三个参数就是lambda表达式，可以看做一个未命名的函数，此处为find_if的谓词
    //计算个数
    return words.end() - wc;
}
```

3.2.5. 可变 lambda

一般情况下，值捕获的对象在 lambda 函数体中不能被修改。如果我们想要在 lambda 中修改被值捕获的对象，可以在参数列表后面，尾置返回箭头->(或者是函数体前)加 mutable。

```
auto k = [i, &j, K = 0]() mutable -> double{
  ++i;//i是值捕获的，但是加了mutable后就可以修改
  j += i;
  return i;
};
```

注意: 修改的也是拷贝出来的变量，不是 lambda 外的变量

3.3. 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。

虽然在例子中我们可以写一个 check_size 函数来代替 lambda，但是此时 check_size 需要 string 和 size 两个参数，是一个二元谓词，但是 find_if 只接受一元谓词。此时就需要参数绑定

bind，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，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。
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 allable，并传递给它 arg_list 中的参数。
如果要使用_n作为占位参数，则必须使用名字空间placeholders，如果已经使用了std，那么只需要using std::placeholders，否则就要using namespace placeholders
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 的第个参数，_2 为第二个参数，依此类推

```
// check6是一个可调用对象，接受一个string类型的参数(_1占位的参数)并用此string和值6(直接传递的)来调用check_size
auto check6 = bind (check_size,_1, 6);
string s("OK");
check6(s);//调用check6
//代替lambda的check6版本
//如果要使用占位参数_n
using std::palceholders;//如果前面已经使用了std
using namespace std::placeholders;//如果前面没有使用std
  auto wc = find_if(words.begin(), words.end(), bind(check_size, _1, sz));
```