泛型算法

- C++ Primer第五版
- 第10章

泛型算法概述

泛型的：可以用于不同类型的容器和不同类型的元素

容器中定义的操作非常有限，其他操作（例如：查找特定元素、替换或删除一个特定元素、排序等）都是通过一组泛型算法来实现的。

大多数算法都定义在头文件algorithm中。头文件numeric中还定义了一组算

一般情况下，这些算法并直接操作容器

```
int val = 42; //我们将查找的值
//如果在vec中找到想要的元素，则返回结果指向它，
//否则返回结果为vec.cend()，即第二个参数
auto result = find(vec.cbegin(), vec.cend(), val);
//报告结果
cout<<“值：”<<val<<(result == vec.cend()?“ 不存在”:“ 存在  ")<<endl;

string val = "a value"; //我们要查找的值
//此调用在lists中查找string元素
auto result = find(list.cbegin(), list.cend(), val);

//由于指针就像内置数组上的迭代器一样，所以也可以用find在数组中查找
int ia[] = {27,210,12,47,109,83};
int val =83;
int *result = find(begin(ia),end(ia),val);
//在从ia[1]开始，直至（但不包含）ia[4]的范围内查找元素
auto result = find(ia+1,ia+4,val);
```

find的执行步骤

1. 访问序列中的首元素。
2. 比较此元素与我们要查找的值。
3. 如果此元素与我们要查找的值匹配，find返回标识此元素的值。
4. 否则，find前进到下一个元素，重复步骤2和3。
5. 如果到达序列尾，find应停止。
6. 如果find到达序列末尾，它可应该返回一个住处元素找到的值。

这些步骤不依赖容器所保存的元素类型。
只要有一个迭代器可用来访问元素，就完全不依赖与容器类型

泛型算法本身不会执行容器的操作，他们只会运用于迭代器上。

初识泛型算法

标准库提供了超过100个算法。

find count accumulate

只读算法：读取其输入范围内的元素，而从不改变元素。

```
//对vec中的元素求和，和的初值为0(第三个参数还决定了返回类型)
int sum = accumulate(vec.cbegin(), vec.cend(),0);

string sum = accumulate(v.cbegin(),v.cend(),string(""));//正确
string sum = accumulate(v.cbegin(),v.cend(),"");//错误：const char* 没有定义+运算符
```

不要求类型　但是需要能够进行==操作

接受单一迭代器来表示第二个序列的算法，都假设第二个序列至少与第一个序列一样长

```
//第三个参数表示第二个序列的首元素
//roster2中的元素数目应该至少与roster1一样多
equal(roster1.cbegin(),roster1.cend(),roster2.cbegin());
//上面的roster1可以是vector<string>
//而roster2是list<const char*>只要能够访问，能够比较即可
```

因此写程序要注意，迭代器是不会帮助我们去检查的

写容器元素的算法

```
fill(vec.begin(),vec.end(),0);//将每个元素重置为0
//将容器的一个子序列设置为10
fill(vec.begin(),vec.begin() + vec.size/2,10);
```

```
vector<int> vec; //空vector
//使用vec,赋予它不同值
fill_n(vec.begin(),vec.size() ,0); //将所有元素重置为0
fill_n(dest,n,val); //dest指向一个元素，而从dest开始至少需要包含n个元素

//下面的代码是错误的
vector<int> vec; //空向量
//灾难：修改vec中的10个（不存在）元素
fill_n(vec.begin(),10,0);
```

向目标的位置迭代器写入数据的算法假定目标位置足够大，能容纳要写入元素，编译器也不会报错，所以写程序要注意！！

插入迭代器：back_inserter，是定义在头文件iterator中的一个函数：
接受指向容器的引用，返回绑定该容器的插入迭代器。

```
vector<int> vec; //空向量
auto it = back_inserter(vec); //通过它赋值会将元素添加到vec中
*it = 42; //vec中现在有一个元素，值为42

vector<int> vec2;
//正确：back_inserter创一个插入迭代器，可用来项vec添加元素
fill_n(back_inserter(vec2),10,0);//添加10个元素到vec
//每次赋值，会在迭代器上调用push_back
```

拷贝算法

```
int a1[]={0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; //a2与a1大小一样
//ret指向拷贝到a2的尾元素之后的位置
auto ret = copy(begin(a1),end(a1),a2);//把a1的内容拷贝给a2

//将所有值为0的元素改为42
replace(list.begin(),list.end(),0,42);

//使用back_insterter按需要增长目标序列
replace_copy(ilist.cbegin(),ilist.cend(),back_inserter(ivec),0,42);
//上面的语句调用后，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中都变成了42
```

重排容器元素的算法

```
//消除重复单词
void elimDups(vector<string>& words)
{
    //按字典顺序排序words，以便查找重复单词
    sort(words.begin(), words.end());
    //unique消除相邻的重复项
    //排列在范围的前部，返回指向不重复区域之后一个位置的迭代器
    auto end_unique = unique(words.begin(), words.end());
    words.erase(end_unique, words.end());
}

int main()
{
    vector<string> words =          { "the","quick","red","fox","jumps","over","the",
    "slow","red","turtle" };

    elimDups(words);

    for ( auto &word : words) {cout << word << " ";}
}
```

fox	jumps	over	quick	red	slow	the	turtle	???	???

end_unique
(最后一个不重复元素之后的位置)

sort默认使用元素类型的<运算符

```
//比较函数，用来按长度排序单词
bool isShorter(const string &s1, const string &s2)
{
	return s1.size() < s2.size();
}
//按长度由短至长排序words，sort可以接受一个二元谓词参数
sort(words.begin(),words.end(),isShorter);
```

第三个参数是一个谓词:

- 谓词是一个可调用的表达式
- 返回结果是一个能用做条件的值
- 谓词分为一元谓词和二元谓词

```
elimDups(words);
stable_sort(words.begin(), words.end(), isShorter);
for ( auto &word : words) {//无需拷贝字符串
	cout << word <<" ";
}
```

stable_sort:稳定排序算法，维持相等元素的原有顺序

sort默认使用元素类型的<运算符

@阿西拜-南昌

```
//比较函数，用来按长度排序单词
bool isShorter(const string &s1, const string &s2)
{
	return s1.size() < s2.size();
}
//按长度由短至长排序words，sort可以接受一个二元谓词参数
sort(words.begin(),words.end(),isShorter);
```

第三个参数是一个谓词:
- 谓词是一个可调用的表达式
- 返回结果是一个能用做条件的值
- 谓词分为一元谓词和二元谓词

定制操作：lambda表达式

可调用的代码单元，一个未命名的内联函数。

尾置返回

```
[capture list] (parameter list) -> return type {function body}
```

局部变量列表

可以忽略参数列表和返回类型

```
#include <iostream>
using namespace std;
int main()
{
	auto f = [] {return 42; };
	f();
}
```

lambda 调用的实参数目 永远与形参数目相等

```
void biggies(vector<string>& words, vector<string>::size_type sz) {
	elimDups(words);//将words按字典顺序排序，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 << "长度大于等于" << sz << "的元素有" << count << "个"<<endl;
	for_each(wc, words.end(), [](const string & s) {cout << s << " "; });
	cout << endl;
}

int main()
{
	vector<string> words = { "the","quick","red","fox",
	"jumps","over","the","slow","red","turtle" };
	biggies(words,4);
}
```

```
Microsoft Visual Studio 调试控制台
长度大于等于4的元素有5个
over slow jumps quick turtle
```

lambda捕获和返回

向函数传递lambda时，同时定义了一个（未命名的）新类型和该类型的一个对象。

默认情况下，新类型包含了捕获的变量，作为数据成员。

```
// 值捕获
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 stored a copy of v1 when we created it
}
// 引用捕获
void fcn2(){
        size_t v1 = 42;  // local variable
        // the object f2 contains a reference to v1
        auto f2 = [&v1] { return v1; };
        v1 = 0;
        auto j = f2(); // j is 0; f2 refers to v1; it doesn't store it
}
```

一个lambda 只有在列表获取一个它所在函数的局部变量，才能在函数中使用该变量

当以引用方式捕获一个变量时，必须保证在lambda执行时变量时存在的

获取列表只用于局部非static变量，因为它可以直接使用局部static变量和它所在函数之外声明的名字1

隐式捕获：让编译器根据lambda体中的代码来推断需要使用哪些变量

```
//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;});
//可以混合使用隐式捕获和显示捕获
void biggies(vector<string> &words,vecotr<string>::size_type sz,
                                                ostream &os = cout, char c=' ')
{
    //其他处理与前例一样
    //偶数隐式捕获，引用捕获方式；c显示捕获，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;});
}
```

混合使用 第一个元素必须是一个 & 或

[隐式，显式]

可变lambda

```
void fcn3(){
    size_t v1 = 42;  // local variable
    // 对于值拷贝的变量，如果需要修改，必须加上关键字mutable
    auto f = [v1]() mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j is 43
}

void fcn4(){
    size_t v1 = 42;  // local variable
    // 对于非const变量的引用，可以通过f2中的引用修改
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j is 1
}
```

参数绑定（bind函数）：定义在functional头文件中

调用bind的一般形式为：

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

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

vector<string> words = { "string1","abcde" };
bool check_size(const string& s, string::size_type sz)
{
	return s.size() >= sz;
}
int main()
{
	//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);

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

	auto wc = find_if(words.begin(), words.end(), bind(check_size, _1, 6));

	auto wc2 = find_if(words.begin(), words.end(), check6);
}
```

名字_n 都定义在placeholders这个命名空间中， 而这个命名空间本身定义在std 命名空间

bind的参数

```
//g是一个有两个参数的可调用对象
auto g = bind(f,a,b,_2,c,_1);
//g(X,Y)的调用会映射到：f(a,b,Y,c,X)
```

用bind重排参数顺序

```
//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));
//当sort需要比较两个元素A和B时，调用isShorter(A,B)
//当sort比较两个元素时，就好像调用了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会被拷贝

```
//错误：不能拷贝os
for_each(words.begin(),words.end(),bind(print,os,_1,' '));
//对于ostream对象，不能拷贝。必须使用标准库ref函数包含给定的引用
for_each(words.begin(),words.end(),bind(print,ref(os),_1,' '));
```

再探迭代器：
插入迭代器、流迭代器、反向迭代器、移动迭代器

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

插入迭代器有三种类型：

- back_inserter，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，插入指定迭代器之前的位置。

```
//it是由inserter生成的迭代器
*it = val;//其效果与下面代码一样

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

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

//拷贝完成后，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()));
```

front_inserter生成的迭代器总是指向容器的第一个元素。这点与inserter生成的迭代器不同。

iostream迭代器　　#include<iterator>

将他们对应的流，当做一个特定类型的元素序列来处理。

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 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

```
istream_iterator<int> int_iter(cin); //绑定一个流，从cin读取int
istream_iterator<int> eof;//默认初始化迭代器，尾后迭代器 遇到文件尾 or IO错误迭代器的值就和尾后迭代器一样
while(in_iter != eof)
//解引用迭代器，获得从流读取的前一个值
      vec.push_back(*in_inter++);//后置递增运算读取流，返回迭代器的旧值

//可以将程序重写为下面的形式
istream_iterator<int> in_iter(cin),eof;
vector<int> vec(in_iter,eof);//从迭代器范围构造vec

ifstream in("afile");
istream_iterator<stirng> str_it(in); //从 “afile”读取字符串
```

使用算法操作流迭代器

```
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1 2 5,则输出为8
```

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

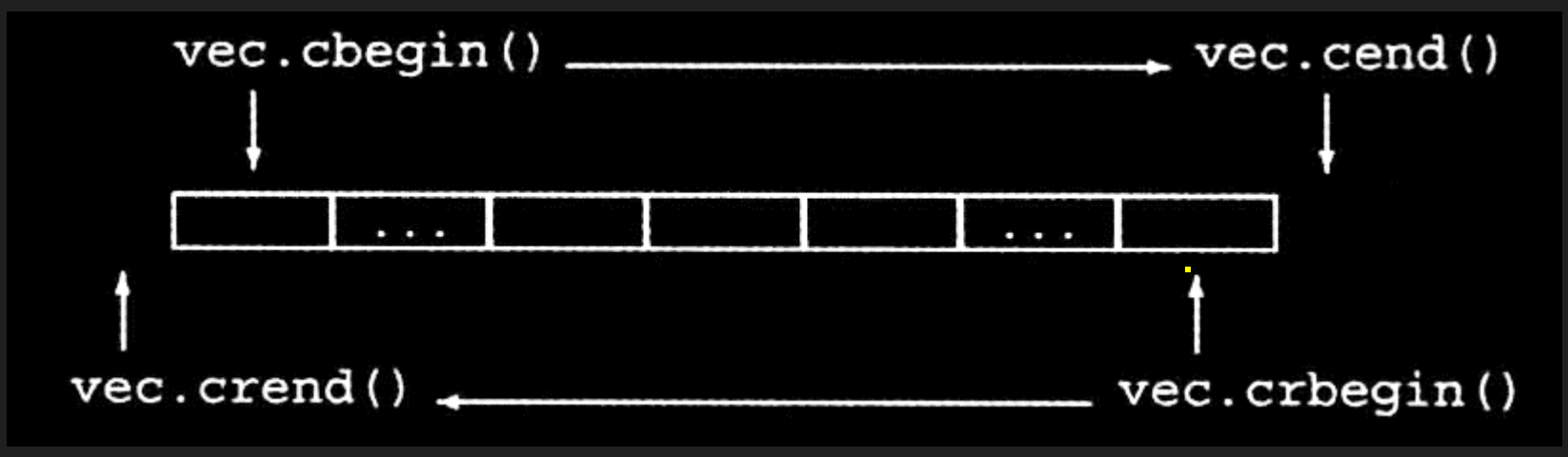
```
//使用ostream_iterator来输出值的序列 :ostream_iterator 的第二个参数
ostream_iterator<int> out_iter(cout, " "); 他是一个字符串  打印每一个元素后
for(auto e:vec)                            就会打印该字符串  “字符常量 ”  or  “空格”
      *out_iter++=e;//赋值语句实际上将元素写到cout
cout<<endl;

//项out_iter赋值时，可以忽略解引用和递增
for(auto e:vec)
      out_iter = e;//赋值语句将元素写到cout
cout<endl;

//通过copy来打印vec中短时
copy(vec.begin,vec.end(),out_iter);
cout<<endl;
```

运算符*和++实际上对ostream_iterator对象不做任何事情
推荐第一种写法，如其他迭代器保持一种，替换方便

使用迭代器处理类类型

```
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_iterm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; //
            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
```

反向迭代器




```
vector<int> vec = { 0,1,2,3,4,5,6,7,8,9 };
//从尾元素到首元素的方向迭代器
for (auto r_iter = vec.crbegin(); r_iter != vec.crend(); ++r_iter)
cout << *r_iter << endl;//打印9,8,7,...,0      如果使用=赋值for
                                                则需要解引用
sort(vec.begin(), vec.end());//按“正常序”排序vec,升序 sort 不能用const 的迭代器
sort(vec.rbegin(), vec.rend());//按逆序排序：将最小元素放在vec的末尾，降序
```

反向迭代器需要递减运算符，所以forward_list或流迭代器不能创建反向迭代器

```
string line = "FIRST,MIDDLE,LAST";
//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;

//在第一逗号分隔的列表中查找最后一个元素
auto rcomma = find(line.crbegin(), line.crend(), ',');
//错误：将逆序输出单词的字符
cout << string(line.crbegin(), rcomma) << endl;
//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;
```

```
Microsoft Visual Studio 调试控制台
FIRST
TSAL
LAST
```

泛型算法结构

算法的最基本的特性是他要求其迭代器提供哪些操作。

算法所要求的5个迭代器类别	
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

类似容器，迭代器的操作也是由层次的。高层类别的迭代器支持所有底层类别迭代器的操作，例如：

- ostream_iterator只支持递增、解引用和赋值。
- vector、string和deque的迭代器还支持递减、关系和算术运算。
- C++标准指明了算法的每个迭代器参数的最小类别。
  - 例如：replace_copy的前两个迭代器至少是向前迭代器。第三个至少是输出迭代器。

算法形参模式：

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);
```

beg end 2个参数的迭代器　dest 一个参数的迭代器　args　值

算法命名和重载规范：　_if _copy

规定如何提供一个操作替代默认的运算符，以及算法将输出数据写入输入序列还是一个分离的目的地等问题。

```
//将相邻重复元素删除
unique(beg,end); //使用 == 运算符比较元素
unique(beg,end,comp); //使用comp比较元素

//在范围内查找特定元素第一次出现的位置。
find(beg,end,val);
find_if(beg,end,pred); //查找第一个令pred为true的元素

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

//从vi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;});
```

重载

通过名字区分

不拷贝

拷贝

特定容器算法

list和forward_list成员函数版本的算法

这些操作都返回 void	
lst.merge(lst2) lst.merge(lst2, comp)	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.remove(val) lst.remove_if(pred)	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.reverse()	反转 lst 中元素的顺序
lst.sort() lst.sort(comp)	使用<或给定比较操作排序元素
lst.unique() lst.unique(pred)	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

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。

list和forward_list的splice成员函数参数

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

splice算法是链表结构所特有的