第十章 泛型算法

概述

- 标准库并没有给每个容器都定义成员函数来实现各种操作，而是定义了一组**泛型算法**：可以用于不同类型的容器和不同类型的元素
- 大多数算法定义在 `<iostream>` 中；

  标准库还在 `<numeric>` 中定义了一组数值泛型算法
- 一般情况下，这些算法通过遍历**两个迭代器**指定的元素范围来进行操作

  > 指针就像内置数组的迭代器一样，所以也支持对数组进行操作
  >
  > - 可以利用标准库 `begin()` 和 `end()` 函数
  > - 元素范围是左闭右开的
- 迭代器令算法不依赖于容器，但算法依赖于元素类型的操作

  > - 除了对具体元素进行操作外，函数其他操作都可以用迭代器操作来实现
  > - 大多数算法都使用了一个（或多个）元素类型上的操作，依赖元素的类型
  >
  > > **大多数算法支持我们使用自定义的操作来替换默认的运算符**
- 泛型算法本身不会执行容器的操作，它们运行在迭代器之上，执行迭代器的操作

  > 泛型算法永远不会改变容器的大小（插入or删除元素），但可能改变元素的值，位置

初识泛型算法

> 具体有哪些算法，在需要的时候去查一下就好

只读算法

一些算法只会读取输入范围的元素，而从不改变元素

```
1 xxx(begin, end, ...)
```

- `find()`

- `accumulate()`：求和函数
  - 定义在 `<numeric>` 中
  - 第三个参数代表和的初值，决定了函数中使用哪个加法运算符以及返回的类型

    > 这蕴含了一个编程假定：将元素类型加到和的类型上的操作必须是可行的
    >
    > 也就是说，元素的类型 和 和的类型 必须是相容的

- `equal()`：确定两个序列是否保存相同的值
  - 第三个参数代表第二个序列的首元素

    假定比较的两序列一样长

    > 就像数组越界一样，编译器不会保存，需要程序员保证
    >
    > ```
    > vector<int> a{1,2,3};
    > cout<<equal(a.begin(), a.end(), a.begin()+1);        // 编译通过，运行通过，结果是未定义的
    > ```

  - 通过 `==` 比较

写容器元素的算法

一些算法将新值赋予序列中的元素。

- 当我们使用这类算法时，必须保证序列大小不小于我们写入数据的大小。

  > 最多改变给定范围内的序列的元全部元素值

- 算法不检查写操作

```
fill_n(dest, n, val);        // 算法假定写入指定个元素是安全的

// 错误写法
vector<int> a;
fill_n(a.begin(), 10, 0);    // 编译通过，但运行出错
```

介绍 `back_inserter`

一种**保证算法有足够元素空间**来容纳插入数据的方法是使用 **插入迭代器**

- `back_inserter()`：接受一个指向容器的引用，返回一个与该容器绑定的**插入迭代器(*`class back_insert_iterator`*)**

- 定义在 `<iteratro>` 中

```
vector<int> vec;
auto it = back_inserter(vec);
fill_n(it, 10 , 0);      // 这样，即使vec最初没有足够的空间运行也没有问题
cout<<a.capacity();      // 16
```

> 实际上是调用 `push_back()` 添加元素到容器

拷贝算法

向目的位置迭代器指向的序列中的写入数据；

```
/*
copy(xx, xx, xx)     前两个表示输入范围，第三个表示目的序列的起始位置
*/
vector<int> a{1,2,3};
vector<int> b;
copy(a.begin(), a.end(),b.begin());      // 运行出错
```

- copy的目的序列至少要包含与输入序列一样多的元素；

  返回的目的位置迭代器递增后的后一个迭代器；

`replace()`：读入一个序列，将其中所有等于给定值的元素都改为另一个

```
replace(a.begin(), a.end(), 1, 2);
cout<<a[0];      // 2
```

`replace_copy()`：保留原序列不变，将改变的序列保存在另一个新序列中

```
vector<int> a{1,2,3};
vector<int> b(4);
replace_copy(a.begin(), a.end(),b.begin()+1, 1, 2);
cout<<a[0];      // 1
cout<<b[1];      // 2
cout<<b[0];      // 0 值初始化？
```

- 保证目的序列是够的，也可以用 `back_inserter` 来确保容量

重排容器元素的算法

`sort()`：利用`<`运算符来实现大小排序

`unique()`：重排，将不重复的元素出现在容器最开始的部分

> 保持元素的相对次序，返回指向不重复区域之后的一个位置的迭代器

定制操作（重点）

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`<`和`=`运算符来完成操作。

标准库为这些算法还定义了额外的版本，允许提供**自定义的操作来代替默认运算符**（重载运算符）

向算法传递函数

```
1 //sort重载版本，接受三个参数，此参数是一个谓词
2 sort(begin,end,cmp);
3 // 接受一个二元谓词的sort版本，将用该谓词代替<来比较元素
```

谓词：一个可调用的表达式，其返回结果是一个能用作条件的值

> 可调用，表示是一个函数形式

- 一元谓词：接受一个参数；

  二元谓词：接受两个参数
- 接受谓词参数的算法对输入序列中的元素调用谓词

  > 因此元素类型必须和谓词的参数是相容的

Lambda表达式

> 对于那种只在一两个地方使用的简单操作，lambda表达式是好用的。
>
> 如果我们需要在很多地方使用相同的操作，通常应定义一个函数
>
> 如果一个操作需要很多语句，也不建议使用lambda表达式

可调用对象：对于一个对象或表达式，如果可以对其使用调用运算符[1]，则称它为可调用的

- 函数；函数指针；重载了函数调用运算符的类；lambda表达式
- 我们可以向一个算法传递任何类别的可调用对象

lambda表达式：一个lambda表达式表示一个可调用的代码单元

> 可以理解一个未命名的内联函数；
>
> 可以定义在函数内部

- 构成

```
[捕获列表](参数列表) -> 返回类型 {函数体}
```

  - 捕获列表是lambda **所在函数**中定义的**局部非静态变量**的列表；

    > 捕获列表只用于局部非静态变量，可以直接使用局部static变量和它所在函数之外声明的名字

    只有在捕获列表中声明的变量，才能在其函数体中使用；
  - 参数列表不能有默认参数，可以省略

    > - 实参数目永远等于形参数目
    > - 省略参数列表等价指定一个空参数列表

  - 返回类型必须使用尾置返回，可以省略

    > 省略返回类型，会根据函数体中的代码自动推断
    >
    > - 如果只有一个return语句，则返回类型从返回的语句中推断；
    > - 否则，就返回类型被认为是void

- 示例

```
// 调用find_if
find_if(word.begin(), word.end(), [sz](const stirng &a){return a.size>=sz});    // 返回第一个长度大于等于sz的迭代器位置
```

lambda的定义

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。
？？后面解释

> 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。 **？？**

  lambda的数据成员也在lambda对象创建时被初始化

> 类声明时，对象都没有定义

lambda的捕获

- 值捕获

```
[names,...]
```

  - 被捕获的变量的值是在lambda创建时就被拷贝的，而不是调用时拷贝

```
int a = 1;
auto f = [a]{return a;};
a = 2;
cout<<f();  //   输出 1;
```

- 引用捕获

```
[&name,...]
```

  - 必须确保被引用的对象在lambda执行时是存在的
- 隐式捕获

```
[&]     // 所有推断变量采用引用捕获方式
[=]     // 所有推断变量采用值捕获方式
```

  - 让编译其根据lambda函数体中的代码来推断我们要使用哪些变量
- 混合捕获

```
[&, names,...]      // 推断的是引用方式，声明的是值捕获方式
[=, names,...]      // 推断的是值捕获方式，声明的是引用方式
```

  - 推断的声明的必须是不同的类型
- 可变lambda

  默认情况下，对于一个值拷贝的变量，lambda不允许改变其值。

  如果希望在函数体内改变，则要在参数列表和加上 `mutable`

```
int a=1;
auto p = [a]()mutable {return ++a;};
auto p = [a]() {return ++a;};       // 报错
cout<<p();  // 2
cout<<p();  // 3
cout<<a;    // 1
------------------
int a=1;
```

```
auto p = [&a]() {return ++a;};
cout<<p();  // 2
cout<<p();  // 2
cout<<a;    // 2
-----------------
 int a=10;
auto p = [&a]() {return ++a;};
cout<<p();  // 11
a=2;
cout<<p();  // 3
```

参数绑定

C++11 利用标准库函数 `bind()`，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

```
// 必须有头文件 <functional>
auto newCallable = bind(callable, arg_list);
```

- callable，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
  - 使用placeholders名字

```
using namespace std::placeholders;
bool check(int a, int b) {
    return a> b;
}
//main()
vector<int> a{1,2,3};
int value = 2;
auto f = bind(check, _1,value);
bool t = f(3);
auto p = find_if(a.begin(), a.end(), f);        // 注意这里f不能加括号，因为要求的是一个可调用对象，加了括号是使用该可调用对象
cout<<*p;
/*
_1代表该bind生成的函数只有一个参数的，符号find_if的一元谓词要求
*/
```

  - `_1` 是"占位符"，表示的是newCallable的参数，它们占据了传递给newCallable的参数的"位置"

    > 1表示newCallable的第一个参数，以此类推；

> 需要命名空间 `std::placeholders`

- 调用newCallable实际上是去调用callable，并传递给它arg_list中的参数

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

```
auto g = bind(f, a, b, _2, c, _1);
/*
f是一个5个参数的函数，两个占位符表示调用时需要给两个参数
*/
auto g = bind(f,_2,_1);
g(a,b) <==> f(b,a);
```

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

- 有时我们希望以引用的方式传递；
- 有时参数不允许拷贝

如果希望传递给bind一个对象而又不拷贝它，就必须使用标准库 `ref函数`

```
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。** `??`

```
auto in = ref(cin);
auto p = in;
---------------;
auto & in =cin;
auto in2 = in;      // 错误
```

> ref() 函数的返回值 `reference_wrapper` 类类型，reference_wrapper 是个类模板，用来模仿一个类型为T 的对象的引用，使用起来就像是引用一样，
>
> 不过reference_wrapper 对象是可以拷贝构造和赋值构造的。
>
> - `cref()`：生成一个保存const引用的类

再探迭代器

除了容器内定义的迭代器外，标准库在头文件 `<iterator>` 中还定义了额外几种迭代器

- 插入迭代器：这些迭代器被绑定到一个容器中，可用来向容器插入元素

- 流迭代器：这些迭代器被绑定到输入流或输入流上，可用来遍历关联的IO流
- 反向迭代器：这些迭代器向前移动而不是向后移动

  > 除了`forward_list`之外，标准库容器都有反向迭代器

- 移动迭代器：这些专用的迭代器不是拷贝其中的元素，而是移动它们（**后续介绍**）

插入迭代器

插入迭代器一种迭代器适配器[2],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

- `it = t`：在it指定的位置插入值t
  - 插入的位置依赖于插入迭代器的种类

```
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。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。元素被插入到给定迭代器之前的位置
```

  > 注意实现使用的操作，必须对应容器存在该函数，才可创建对应种类的插入迭代器

- `*it,++it,it++`：这些操作尽管存在，但没有任何意义，仍然返回it

iostream迭代器

虽然iostream类型不是容器，但是标准库定义了可以用于这些IO类型对象的迭代器；

这些迭代器将对应的流当作一个特定类型的元素类型序列来处理；

通过流迭代器，可以用泛型算法从流对象读取数据以及向其写入数据

流迭代器不支持`--运算符`

---

istream_iterator：读输入流

操作	含义
istream_iterator in(is)	in从输入流中读取类型为T的值
istream_iterator end	读取类型为T的值的istream_iterator迭代器，表示尾后位置
in1 == in2	必须读取相同类型。判断绑定的对象是否相同
*in	返回从流中读取的值
++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

- 绑定的类型必须定义了`>>`
- 关联到流遇到文件末尾或IO错误或类型不同，迭代器的值就与尾后迭代器相同
- istream_iterator允许使用懒惰求值
  - 不保证迭代器立即从流读取数据，只保证使用时，从流中的读取操作一定完成 `??`

ostream_iterator：写输出流

操作	含义
ostream_iterator out(os)	out将类型为T的值写到输入流os中
ostream_iterator out(os, d)	out将类型为T的值写到输入流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(**不支持string**)
out = val	用`<<`运算符将val写入到out所绑定的ostream中。 val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

- 绑定的类型必须定义了`<<`
- 必须绑定一个流，不允许空的或表示尾后位置

```
ostream_iterator<int> out(cout,' ');
for (auto e : vec) {
    *out_iter++ = e;        // 推荐该形式，保持一致性
}
cout<<endl;          // 刷新流，此时才会输出
```

反向迭代器

- 反向迭代器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，`++`、`--`的含义会倒过来

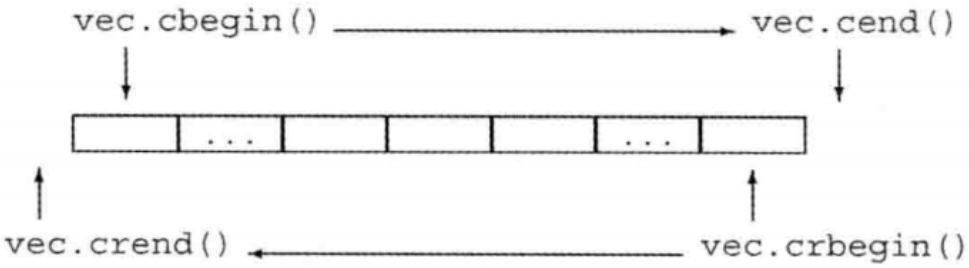

图 10.1：比较 cbegin/cend 和 crbegin/crend

```
1 sort(begin(), end());       // A_1 < A_2 < A_3
2 sort(rbegin(), rend());     // A_3 < A_2 < A_1
```

- 可以利用反向迭代器的成员函数`base()`，将返回迭代器转化为指向相邻一个位置的正向迭代器

> **注意并不是一个相同的位置**
>
> 因为左闭右开形式的限制

泛型算法结构

- **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。**

  可以传递一个更强的迭代器，但是一个更差的迭代器会产生错误。
- **算法还共享一组参数传递规范和一组命名规范**

五类迭代器

迭代器类型（category）	能力	供应者
Input 迭代器	向前读取（*read*）	istream
Output 迭代器	向前写入（*write*）	ostream, inserter
Forward 迭代器	向前读取和写入	
Bidirectional 迭代器	向前（*forward*）和向后（*backward*）读取和写入	list, set, multiset, map, multimap
Random access 迭代器	随机存取，可读取也可写入	vector, deque, string, array

> 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。

- 输出迭代器：只写不读；单遍扫描，只能递增

> Output迭代器和Input迭代器相反，其作用是将元素值一个个写入

- 必须支持的操作

```
*iter
++;
```

- 输入迭代器：只读不写；单遍扫描，只能递增

> 几乎所有迭代器都具备输入迭代器的能力

- 必须支持的操作

```
== !=
*iter, iter->member
++
```

- 前向迭代器：可读写；多遍扫描，只能递增

> =输入迭代器+输出迭代器

- 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
  - 必须支持的操作

```
...
iter[n];
iter += n ; iter -= n; iter +/- n;
iter1 > < >= <= iter2
```

算法形参模式

多数算法的4种调用形式：

```
/*
alg：算法名字
[beg，end)：算法所操作的输入范围
dest：目标位置
[beg2，end2)：第二个范围
other args：额外的、非迭代器的特定参数
*/
alg(beg，end，other args);
alg(beg，end，dest，other args);
alg(beg，end，beg2，other args);        // 假定beg2开始的序列与[beg，end)所表示的范围一样大
alg(beg，end，beg2，end2，other args);
```

算法命名规范

算法遵循的命名和重载规范：

- 如何提供一个操作代替默认的 `<`、`==` 运算符

  > 接受谓词参数或不接受额外参数的算法，通常都是重载的函数；

- 算法是将输出数据写入输入序列还是分离的目标位置

接受一个元素值的算通常都有一个不同名的版本(`_if`)，该版本接受一个谓词代替元素值

```
find(beg，end，val);
find_if(beg，end，val，pred);
```

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会给定的输入序列中。
- `_copy` 这些算法还有另一个版本，将元素写到一个指定的输出目的位置

  ```
  reverse(beg，end);
  reverse_copy(beg，end，dest);
  ```

一些算法同时提供 `_copy 和 _if` 版本

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

特定容器（list/forward_list)算法

定义了独有的成员函数形式的算法

成员函数	含义
lst.merge(lst2) lst.merge(lst2, comp)	将来自lst2的元素合并到lst。 lst1和lst2必须是有序的（按`<`比较）。合并和，lst2为空
lst.remove(val) lst.remove_if(pred)	调用erase删除掉满足`==`或一元谓词的值
lst.reverse()	反转lst中的元素的顺序
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删除掉同一个值的 **连续拷贝**

- 优先使用成员函数而不是泛型函数
- 以上所有函数返回`void`

链表结构独有成员函数算法

`splice()`

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底层的容器**

1. 调用运算符（） ↩

2. 通过封装使用其他的迭代器，来实现特定的操作 ↩