primer学习

第九章 顺序容器

本章是第3章内容的扩展,完成本章的学习后,对标准库顺序容器知识的掌握就完整了。元素在顺序容器中的顺序与其加入容器时的位置相对应。标准库还定义了几种关联容器,关联容器中元素的位置由元素相关联的关键字值决定。我们将在第11章中介绍关联容器特有的操作。

所有容器类都共享公共的接口,不同容器按不同方式对其进行扩展。这个公共接口使容器的学习更加容易——我们基于某种容器所学习的内容也都适用于其他容器。每种容器都提供了不同的性能和功能的权衡。

一个容器就是一些特定类型对象的集合。顺序容器(sequential container)为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值,而是与元素加入容器时的位置相对应。与之相对的,我们将在第11章介绍的有序和无序关联容器,则根据关键字的值来存储元素。

标准库还提供了三种容器适配器,分别为容器操作定义了不同的接口,来与容器类型适配。我们将在本章末尾介绍适配器。

> **Note**
> 本章的内容基于3.2节、3.3节和3.4节中已经介绍的有关容器的知识,我们假定读者已经熟悉了这几节的内容。

9.1 顺序容器概述

表9.1列出了标准库中的顺序容器,所有顺序容器都提供了快速顺序访问元素的能力。但是,这些容器在以下方面都有不同的性能折中:

- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

表9.1:顺序容器类型

类型	描述
vector	可变大小数组。支持快速随机访问。 在尾部之外的位置插入或删除元素肯可能很慢。

类型	描述
deque	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快。
list	双向链表。只支持双向顺序访问。在list中任何位置进行插入/删除操作速度都很快。
forward_list	单项链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快。
array	固定大小数组。支持快速随机访问。不能添加户删除元素。
string	与vector相似的容器,但专门用于保存字符。随机访问快。在尾部插入/删除数度快。

除了固定大小的array外,其他容器都提供高效、灵活的内存管理。我们可以添加和删除元素,扩张和收缩容器的大小。容器保存元素的策略对容器操作的效率有着固有的,有时是重大的影响。在某些情况下,存储策略还会影响特定容器是否支持特定操作。

例如,string和vector将元素保存在连续的内存空间中。由于元素是连续存储的,由元素的下标来计算其地址是非常快速的。但是,在这两种容器的中间位置添加或删除元素就会非常耗时:在一次插入或删除操作后,需要移动插入/删除位置之后的所有元素,来保持连续存储。而且,添加一个元素有时可能还需要分配额外的存储空间。在这种情况下,每个元素都必须移动到新的存储空间中。

list 和forward_list两个容器的设计目的是令容器任何位置的添加和删除操作都很快速。作为代价,这两个容器不支持元素的随机访问:为了访问一个元素,我们只能遍历整个容器。而且,与vector,deque和array相比,这两个容器的额外内存开销也很大。

deque是一个更为复杂的数据结构。与string和vector类似,deque支持快速的随机访问。与string和vector一样,在deque的中间位置添加或删除元素的代价(可能)很高。但是,在deque的两端添加或删除元素都是很快的,与list或forward_list 添加删除元素的速度相当。

forward_list和array是新C++标准增加的类型。与内置数组相比,array是种更安全、更容易使用的数组类型。与内置数组类似,array对象的大小是固定的。因此,array不支持添加和删除元素以及改变容器大小的操作。forward_list的设计目标是达到与最好的手写的单向链表数据结构相当的性能。因此,forward_list没有size操作,因为保存或计算其大小就会比手写链表多出额外的开销。对其他容器而言,size保证是一个快速的常量时间的操作。

> **Note**
> 新标准库的容器比旧版本快得多,原因我们将在13.6节(第470页)解释。新,标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好(通常会更好)。现代C++程序应该使用标准库容器,而不是更原始的数据结构,如内置数组。

确定使用哪种顺序容器

> **Tip**
> 通常,使用vector是最好的选择,除非你有很好的理由选择其他容器。

以下是一些选择容器的基本原则:

- 除非你有很好的理由选择其他容器,否则应使用vector。
- 如果你的程序有很多小的元素,且空间的额外开销很重要,则不要使用list或forward_list。
- 如果程序要求随机访问元素,应使用vector或deque。
- 如果程序要求在容器的中间插入或删除元素,应使用list或forward_list。
- 如果程序需要在头尾位置插入或删除元素,但不会在中间位置进行插入或删除操作,则使用deque。
- 如果程序只有在读取输入时才需要在容器中间位置插入元素,随后需要随机访问元素,则
  - 首先,确定是否真的需要在容器中间位置添加元素。当处理输入数据时,通常可以很容易地向vector追加数据,然后再调用标准库的sort函数(我们将在10.2.3节介绍sort(第343页))来重排容器中的元素,从而避免在中间位置添加元素。
  - 如果必须在中间位置插入元素,考虑在输入阶段使用list,一旦输入完成,将list中的内容拷贝到一个vector中。

如果程序既需要随机访问元素,又需要在容器中间位置插入元素,那该怎么办?答案取决于在list或forward_list中访问元素与vector或deque中插入/删除元素的相对性能。一般来说,应用中占主导地位的操作(执行的访问操作更多还是插入/删除更多)决定了容器类型的选择。在此情况下,对两种容器分别测试应用的性能可能就是必要的了。

> **Best Practiees**
> 如果你不确定应该使用哪种容器,那么可以在程序中只使用vector和list公共的操作:使用选代器,不使用下标操作,避免随机访问。这样,在必要时选择使用vector或list都很方便。

9.2 容器库概览

容器类型上的操作形成了一种层次:

- 某些操作是所有容器类型都提供的(参见表9.2,第295页)。
- 另外一些操作仅针对顺序容器(参见表9.3,第299页)、关联容器(参见表11.7,第388页)或无序容器(参见表11.8,第395页)。
- 还有一些操作只适用于一小部分容器。

在本节中,我们将介绍对所有容器都适用的操作。本章剩余部分将聚焦于仅适用于顺序容器的操作。关联容器特有的操作将在第11章介绍。
般来说,每个容器都定义在一个头文件中,文件名与类型名相同。即,deque定义在头文件deque中,list定义在头文件list中,以此类推。容器均定义为模板类(参见3.3节,第86页)。例如对vector,我们必须提供额外信

息来生成特定的容器类型。
对大多数,但不是所有容器,我们还需要额外提供元素类型信息:

```
list<Sales_data>//保存Sales_data 对象的list
deque<double>//保存 double的deque
```

对容器可以保存的元素类型的限制

顺序容器几乎可以保存任意类型的元素。特别是,我们可以定义一个容器,其元素的类型是另一个容器。这种容器的定义与任何其他容器类型完全一样:在尖括号中指定元素类型(此种情况下,是另一种容器类型):

```
vector<vector<string>> lines;//vectorvector
```

此处lines是一个vector,其元素类型是string的vector。

> **Note**
> 较旧的编译器可能需要在两个尖括号之间键入空格,例如 `vector<vector<string> >`

虽然我们可以在容器中保存几乎任何类型,但某些容器操作对元素类型有其自己的特殊要求。我们可以为不支持特定操作需求的类型定义容器,但这种情况下就只能使用那些没有特殊要求的容器操作了。

例如,顺序容器构造函数的一个版本接受容器大小参数(参见3.3.1节,第88页),它使用了元素类型的默认构造函数。但某些类没有默认构造函数。我们可以定义一个保存这种类型对象的容器,但我们在构造这种容器时不能只传递给它一个元素数目参数:

```
//假定noDefault是一个没有默认构造函数的类型
vector<noDefault>v1(10,init);//正确:提供了元素初始化器
vector<noDefault>v2(10);//错误:必须提供一个元素初始化器
```

当后面介绍容器操作时,我们还会注意到每个容器操作对元素类型的其他限制。

表9.2:容器操作

类型别名:

名称	描述
iterator	此容器类型的迭代器类型
const_interator	可以读取元素,不能修改元素的迭代器类型
size_type	无符号整数类型,足够保存此种容器类型最大可能容器但大小

名称	描述
difference_type	带符号整数类型,足够保存两个迭代器之间的距离
value_type	元素类型
reference	元素的左值类型,与value_type&含义相同
const_reference	元素的const左值类型(即,const value_type&)

构造函数:

名称	描述
C c;	默认构造函数,构造空容器(array)
C c1(c2);	构造c2的拷贝c1
C c(b,e);	构造c,将迭代器b和e指定的范围内的元素拷贝到c(array不支持)
C c{a,b,c,...};	列表初始化c

赋值与swap:

名称	描述
c1=c2	将c1中的元素替换为c2中的元素
c1= {a,b,c...}	将c1中的元素替换为列表中的元素(适用于array, 但列表元素个数必须小于等于c1大小)
a.swap(b)	交换a和b的元素
swap(a,b)	与a.swap(b)等价

大小:

名称	描述
c.size()	c中元素的数目(不支持forward_list)
c.max_size()	c可以保存的最大元素数目
c.empty()	若c中存储了元素,返回false,否则返回true

添加/删除元素(不适用于array)

在不同容器中,这些操作的接口都不同

名称	描述
c.insert(args)	将args中的元素拷贝进c
c.emplace(inits)	使用inits构造c中的一个元素
c.erase(args)	删除args指定的元素
c.clear()	删除c中的所有元素,返回void

关系运算符:

名称	描述
==,!=	所有容器都支持相等(不等)运算符
<,<=,>,>=	关系运算符(无序关联容器不支持)

获取迭代器:

名称	描述
c.begin(),c.begin()	返回指向c的首元素和尾元素之后位置的迭代器
c.cbegin(),c.cbegin()	返回const_iterator

反向容器但额外成员(不支持forward_list)

名称	描述
reverse_iterator	按逆序寻址元素的迭代器
const_reverse_iterator	不能修改元素的逆序迭代器
c.rbegin(),c.rbegin()	返回指向c的尾元素和首元素之前位置的迭代器
c.crbegin(),c.credn()	返回const_reverse_iterator

9.2.1迭代器

与容器一样,迭代器有着公共的接口:如果一个迭代器提供某个操作,那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。例如,标准容器类型上的所有迭代器都允许我们访问容器中的元素,而所有迭代器都是通过解引用运算符来实现这个操作的。类似的,标准库容器的所有迭代器都定义了递增运算符,从当前元素移动到下一个元素。

表3.6(第96页)列出了容器迭代器支持的所有操作,其中有一个例外不符合公共接口特点——forward_list迭代器不支持递减运算符(--)。表3.7(第99页)列出了迭代器支持的算术运算,这些运算只能应用于string,vector、deque和array的迭代器。我们不能将它们用于其他任何容器类型的迭代器。

迭代器范围

> **Note**
> 迭代器范围的概念是标准库的基础。

一个**迭代器范围**(iterator range)由一对迭代器表示,两个迭代器分别指向同一个容器中的元素或者是尾元素之后的位置(one past the last element)。这两个迭代器通常被称为begin和end,或者是first和last(可能有些误导),它们标记了容器中元素的一个范围。

虽然第二个迭代器常常被称为last,但这种叫法有些误导,因为第二个迭代器从来都不会指向范围中的最后一个元素,而是指向尾元素之后的位置。迭代器范围中的元素包含first所表示的元素以及从first开始直至last(但不包含last)之间的所有元素。

这种元素范围被称为左闭合区间(left-inclusive interval),其标准数学描述为

```
[begin,end)
```

表示范围自begin开始,于end之前结束。迭代器begin和end必须指向相同的容器。end可以与begin指向相同的位置,但不能指向begin之前的位置。

> **对构成范围的迭代器的要求**
> 如果满足如下条件,两个迭代器begin和end构成一个迭代器范:
> 它们指向同一个容器中的元素,或者是容器最后一个元素之后的位置,且我们可以通过反复递增begin来到达end。换句话说,end不在begin之前。

> **WARNING**
> 编译器不会强制这些要求。确保程序符合这些约定是程序员的责任

使用左闭合范围蕴含的编程假定

标准库使用左闭合范围是因为这种范围有三种方便的性质。假定begin和end构成个合法的选代器范围,则

- 如果begin与end相等,则范围为空
- 如果begin与end不等,则范围至少包含一个元素,且begin指向该范围中的第一个元素
- 我们可以对begin递增若干次,使得begin==end

这些性质意味着我们可以像下面的代码一样用一个循环来处理一个元素范围,而这是安全的:

```
while(begin! = end){
*begin = val;//正确:范围非空,因此begin指向一个元素
++begin;//移动选代器,获取下一个元素
}
```

给定构成一个合法范围的选代器begin和end,若begin==end,则范围为空。在此情况下,我们应该退出循环。如果范围不为空,begin指向此非空范围的一个元素。因此,在while循环体中,可以安全地解引用begin,因为begin必然指向一个元素。最后,由于每次循环对begin递增一次,我们确定循环最终会结束。

9.2.2 容器类型成员

每个容器都定义了多个类型,如表9.2所示(第295页)。我们已经使用过其中三种:size_type(参见3.2.2节,第79页)、iterator和const_iterator(参见3.4.1节,第97页)。

除了已经使用过的选代器类型,大多数容器还提供反向选代器。简单地说,反向选代器就是一种反向遍历容器的选代器,与正向选代器相比,各种操作的含义也都发生了颠倒。例如,对一个反向选代器执行++操作,会得到上一个元素。我们将在10.4.3节(第363页)介绍更多关于反向选代器的内容。

剩下的就是类型别名了,通过类型别名,我们可以在不了解容器中元素类型的情况下使用它。如果需要元素类型,可以使用容器的value_type。如果需要元素类型的一个引用,可以使用reference或const_reference。这些元素相关的类型别名在泛型编程中非常有用,我们将在16章中介绍相关内容。

为了使用这些类型,我们必须显式使用其类名:

```
//iter是通过list<string>定义的一个迭代器类型
list<string>::iterator iter;
//count是通过vector<int>定义的一个difference_type类型
vector<int>::difference_type count;
```

这些声明语句使用了作用域运算符(参见1.2节,第7页)来说明我们希望使用 `list<string>` 类的iterator成员及 `vector<int>` 类定义的difference_type。

9.2.3 begin和end成员

begin和end操作(参见3.4.1节,第95页)生成指向容器中第一个元素和尾元素之后位置的选代器。这两个选代器最常见的用途是形成一个包含容器中所有元素的选代器范围。

如表9.2(第295页)所示,begin和end有多个版本:带r的版本返回反向选代器(我们将在10.4.3节(第363页)中介绍相关内容);以c开头的版本则返回const迭代器:

```
list<string> a =("Milton","Shakespeare","Austen");
auto itl = a.begin();//list<string>::iterator
auto it2 = a.rbegin();//list<string>::reverse_iterator
auto it3= a.cbegin();//list<string>::const_iterator
auto it4 = a.crbegin();//list<string>::const_reverse_iterator
```

不以c开头的函数都是被重载过的。也就是说,实际上有两个名为begin的成员。一个是const成员(参见7.1.2节,第231页),返回容器的const_iterator类型。另个是非常量成员,返回容器的iterator类型。rbegin,end和rend的情况类似。当我们对一个非常量对象调用这些成员时,得到的是返回iterator的版本。只有在对个const对象调用这些函数时,才会得到一个const版本。与const指针和引用类似,可以将一个普通的iterator转换为对应的const iterator,但反之不行。

以c开头的版本是C++新标准引入的,用以支持auto(参见2.5.2节,第61页)与begin和end函数结合使用。过去,没有其他选择,只能显式声明希望使用哪种类型的选代器:

```
//显式指定类型
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6= a.begin();
//是iterator还是const iterator依赖于a的类型
auto it7 = a.begin();//仅当a是 const 时,it7是const_iterator
auto it8 = a.cbegin();//it8const_iterator
```

当auto与begin或end结合使用时,获得的选代器类型依赖于容器类型,与我们想要如何使用迭代器毫不相干。但以c开头的版本还是可以获得const_iterator的,而不管容器的类型是什么。

9.2.4 容器定义和初始化

每个容器类型都定义了一个默认构造函数(参见7.1.4节,第236页)。除array之外,其他容器的默认构造函数都会创建一个指定类型的空容器,且都可以接受指定容器大小和元素初始值的参数。

表9.3:容器定义和初始化

名称	描述
C c	默认构造函数,如果C是一个array,则c中元素按默认方式初始化,否则为空
C c1(c2)	c1初始化为c2的拷贝。c1和c2必须是相同的类型,

名称	描述
C c1=c2	相同容器,相同元素类型。array类型,还必须相同大小。
C c{a,b,c...}	c初始化为初始化列表中元素的拷贝。列表元素类型必须与C的元素类型相容。
C c={a,b,c...}	对于array类型,列表中元素数目必须等于或小于array的大小。
C c(b,e)	c初始化为迭代器b和e指定范围中的元素的拷贝。 范围中元素的类型必须与CD元素类型相容(array不适用)

只有顺序容器(不包括array)的构造函数才能接受大小参数

名称	描述
C seq(n)	seq包含n个元素,这些元素进行了值初始化,此构造函数是explicit的。string类型不适用
C seq(n,t)	seq包含n个初始化为值t的元素

将一个容器初始化为另一个容器但拷贝

将一个新容器创建为另一个容器的拷贝的方法有两种:可以直接拷贝整个容器,或者(array除外)拷贝由一个选代器对指定的元素范围。

为了创建一个容器为另一个容器的拷贝,两个容器的类型及其元素类型必须匹配。不过,当传递迭代器参数来拷贝一个范围时,就不要求容器类型是相同的了。而且,新容器和原容器中的元素类型也可以不同,只要能将要拷贝的元素转换(参见4.11节,第141页)为要初始化的容器的元素类型即可。

```
//每个容器有三个元素,用给定的初始化器进行初始化
list<string> authors =("Milton","Shakespeare","Austen");
vector<const char*> articles =("a","an","the");

list<string> list2(authors);//正确:类型匹配
deque<string> authList(authors);//错误:容器类型不匹配
vector<string> words(articles);//错误:容器类型必须匹配
//正确:可以将const char*元素转换为string
forward_list<string> words(articles.begin(),articles.end());
```

> **Note**
> 当将一个容器初始化为另一个容器的拷贝时,两个容器的容器类型和元素类型都必须相同。

列表初始化

在新标准中,我们可以对一个容器进行列表初始化(参见3.3.1节,第88页)

```
//每个容器有三个元素,用给定的初始化器进行初始化
list<string> authors =("Milton","Shakespeare","Austen");
vector<const char*> articles =("a","an","the");
```

当这样做时,我们就显式地指定了容器中每个元素的值。对于除array之外的容器类型,初始化列表还隐含地指定了容器的大小:容器将包含与初始值一样多的元素。

与顺序容器大小相关的构造函数

除了与关联容器相同的构造函数外,顺序容器(array除外)还提供另一个构造函数,它接受一个容器大小和一个(可选的)元素初始值。如果我们不提供元素初始值,则标准库会创建一个值初始化器(参见3.3.1节,第88页)

```
vector<int> ivec(10,-1);//10个int元素,每个都初始化为-1
list<string> svec(10,"hi! ");//10个strings;每个都初始化为"hi! "
forward_list<int> ivec(10);//10个元素,每个都初始化为0
deque<string> svec(10)://10个元素,每个都是空string
```

如果元素类型是内置类型或者是具有默认构造函数(参见9.2节,第294页)的类类型,可以只为构造函数提供一个容器大小参数。如果元素类型没有默认构造函数,除了大小参数外,还必须指定一个显式的元素初始值。

> **Note**
> 只有顺序容器的构造函数才接受大小参数,关联容器并不支持。

标准库array具有固定大小

与内置数组一样,标准库array的大小也是类型的一部分。当定义一个array时,除了指定元素类型,还要指定容器大小:

```
array<int,42>1/类型为:保存42个int的数组
array<string,10>//类型为:保存10个string的数组
```

为了使用array类型,我们必须同时指定元素类型和大小:

```
array<int,10>::size_type i;//数组类型包括元素类型和大小
array<int>::size_type j;//错误:array<int>不是一个类型
```

由于大小是array类型的一部分,array不支持普通的容器构造函数。这些构造函数都会确定容器的大小,要么隐式地,要么显式地。而允许用户向一个array构造函数传递大小参数,最好情况下也是多余的,而且容易出错。

array大小固定的特性也影响了它所定义的构造函数的行为。与其他容器不同,一个默认构造的array是非空的:它包含了与其大小一样多的元素。这些元素都被默认初始化(参见2.2.1节,第40页),就像一个内置数组(参见3.5.1节,第102页)中的元素那样。如果我们对array进行列表初始化,初始值的数目必须等于或小于array的大小。如果初始值数目小于array的大小,则它们被用来初始化array中靠前的元素,所有剩余元素都会进行值初始化(参见3.3.1节,第88页)。在这两种情况下,如果元素类型是个类类型,那么该类必须有一个默认构造函数,以使值初始化能够进行:

```
array<int,10> ial;//10个默认初始化的int
array<int,10> ia2 = 10,1,2,3,4,5,6,7,8,9);//列表初始化
array<int,10> ia3 =(42);//ia3[0]为42,剩余元素为0
```

值得注意的是,虽然我们不能对内置数组类型进行拷贝或对象赋值操作(参见3.5.1节,第102页),但array并无此限制:

```
int digs[10]=(0,1,2,3,4,5,6,7,8,9);
int cpy[10]= digs;//错误:内置数组不支持拷贝或赋值
array<int,10> digits =(0,1,2,3,4,5,6,7,8,9);
array<int,10> copy = digits;//正确:只要数组类型匹配即合法
```

与其他容器一样,array也要求初始值的类型必须与要创建的容器类型相同。此外,array还要求元素类型和大小也都一样,因为大小是array类型的一部分。

9.2.5 赋值和swap

表9.4中列出的与赋值相关的运算符可用于所有容器。赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝:

```
cl=c2;//将c1的内容替换为c2中元素的拷贝
c1=(a,b,c);//赋值后,c1大小为3
```

第一个赋值运算后,左边容器将与右边容器相等。如果两个容器原来大小不同,赋值运算后两者的大小都与右边容器的原大小相同。第二个赋值运算后,cl的size变为3,即花括号列表中值的数目。

与内置数组不同,标准库array类型允许赋值。赋值号左右两边的运算对象必须具有相同的类型:

```
array<int,10> al =(0,1,2,3,4,5,6,7,8,9);
array<int,10> a2 =(0);//所有元素值均为0
al=a2;//替换al中的元素
a2 =(0);//错误: 不能将一个花括号列表赋予数组
```

由于右边运算对象的大小可能与左边运算对象的大小不同,因此array类型不支持assign,也不允许用花括号包围的值列表进行赋值。

表9.4:容器赋值运算

运算	描述
c1=c2	将c1中的元素替换为c2中的元素的拷贝。c1和c2必须具有相同的类型。 array还必须是相同的大小,其他容器拷贝后c1,c2大小同c2.
c= {a1,a2,a3...}	将c1中元素替换为初始化列表中元素的拷贝,array适用, 但注意列表元素个数小于等于array元素个数
swap(c1,c2) c1.swap(c2)	交换c1和c2中的元素。c1和c2必须具有相同的类型。 swap通常比从c2向c1拷贝元素快得多

assign操作不适用于关联容器和array

运算	描述
seq.assign(b,e)	将seq中的元素替换为迭代器b和e所表示的范围中的元素。 迭代器b和e不能指向seq中的元素
seq.assign(il)	将seq中的元素替换为初始化列表il中的元素
seq.assign(n,t)	将seq中的元素替换为n个值为t的元素

> **WARNING**
> 赋值相关运算会导致指向左边容器内部的选代器、引用和指针失效。而swap操作将容器内容交换不会导致指向容器的选代器、引用和指针失效(容器类型为array和string的情况除外)。

使用assign(仅顺序容器)

赋值运算符要求左边和右边的运算对象具有相同的类型。它将右边运算对象中所有元素拷贝到左边运算对象中。顺序容器(array除外)还定义了一个名为assign的成员,允许我们从一个不同但相容的类型赋值,或者从容器的一个子序列赋值。assign操作用参数所指定的元素(的拷贝)替换左边容器中的所有元素。例如,我们可以用assgin实现将一个vector中的一段char*值赋予一个list中的string:

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle;//错误：容器类型不匹配
//正确：可以将const char*转换为string
names.assign(oldstyle.cbegin().oldstyle.cend());
```

这段代码中对assign的调用将names中的元素替换为选代器指定的范围中的元素的拷贝。assign的参数决定了容器中将有多少个元素以及它们的值都是什么。

> **WARNING**
> 由于其旧元素被替换,因此传递给assign的选代器不能指向调用assign的容器。

assign的第二个版本接受一个整型值和一个元素值。它用指定数目且具有相同给定值的元素替换容器中原有的元素:

```
//等价于slistl.clear();
//后跟slistl.insert(slistl.begin(),10,"Hiya! ");
list<string> slistl(1);//1个元素,为空string
slistl.assign(10,"Hiya! ");//10个元素,每个都是“Hiya! "
```

使用swap

swap操作交换两个相同类型容器的内容。调用swap之后,两个容器中的元素将会交换:

```
vector<string> svecl(10);//10个元素的vector
vector<string> svec2(24);//24个元素的vector
swap(svecl,svec2);
```

调用swap后,svecl将包含24个string元素,svec2将包含10个string。除array外,交换两个容器内容的操作保证会很快-一元素本身并未交换,swap只是交换了两个容器的内部数据结构。

> **Note**
> 除array外,swap不对任何元素进行拷贝、删除或插入操作,因此可以保证在常数时间内完成。

元素不会被移动的事实意味着,除string外,指向容器的选代器、引用和指针在swap操作之后都不会失效。它们仍指向swap操作之前所指向的那些元素。但是,在swap之后,这些元素已经属于不同的容器了。例如,假定iter在swap之前指向svec1[3]的string,那么在swap之后它指向svec2[3]的元素。与其他容器不同,对一个string调用swap会导致选代器、引用和指针失效。

与其他容器不同,swap两个array会真正交换它们的元素。因此,交换两个array所需的时间与array中元素的数目成正比。

因此,对于array,在swap操作之后,指针、引用和选代器所绑定的元素保持不变,但元素值已经与另一个array中对应元素的值进行了交换。

在新标准库中,容器既提供成员函数版本的swap,也提供非成员版本的swap。而早期标准库版本只提供成员函数版本的swap。非成员版本的swap在泛型编程中是非常重要的。统一使用非成员版本的swap是一个好习惯。

9.2.6 容器大小操作

除了一个例外,每个容器类型都有三个与大小相关的操作。成员函数size(参见3.2.2节,第78页)返回容器中元素的数目;empty当size为0时返回布尔值true,否则返回false;max_size返回一个大于或等于该类型容器所能容纳的最大元素数的值。forward_list支持max_size和empty,但不支持size,原因我们将在下一节解释。

9.2.7 关系运算符

每个容器类型都支持相等运算符(=和！=);除了无序关联容器外的所有容器都支持关系运算符(>、>=、<、<=)。关系运算符左右两边的运算对象必须是相同类型的容器,且必须保存相同类型的元素。即,我们只能将一个 `vector<int>` 与另一个 `vector<int>` 进行比较,而不能将一个 `vector<int>` 与一个 `list<int>` 或一个 `vector<double>` 进行比较。

比较两个容器实际上是进行元素的逐对比较。这些运算符的工作方式与string的关系运算(参见3.2.2节,第79页)类似:

- 如果两个容器具有相同大小且所有元素都两两对应相等,则这两个容器相等：否则两个容器不等。
- 如果两个容器大小不同,但较小容器中每个元素都等于较大容器中的对应元素,则较小容器小于较大容器。
- 如果两个容器都不是另一个容器的前缀子序列,则它们的比较结果取决于第一个不相等的元素的比较结果。

下面的例子展示了这些关系运算符是如何工作的:

```
vector<int> vl =(1,3,5,7,9,12);
vector<int> v2 =| 1,3,9);
vector<int> v3 =(1,3,5,7];
vector<int> v4 =| 1,3,5,7,9,12 1;
vl < v2//true;vl和v2在元素[2]处不问：vl[2]小于等于v2[2]
vl < v3//false;所有元素都相等,但v3中元素数目更少
vl == v4//true;每个元素都相等,且vl和v4大小相同
v1 == v2//false：v2元素数目比v1少
```

容器的关系运算符使用元素的关系运算符完成比较

> **Note**
> 只有当其元素类型也定义了相应的比较运算符时,我们才可以使用关系运算符来比较两个容器。

容器的相等运算符实际上是使用元素的==运算符实现比较的,而其他关系运算符是使用元素的<运算符。如果元素类型不支持所需运算符,那么保存这种元素的容器就不能使用相应的关系运算。例如,我们在第7章中定义的Sales_data类型并未定义=和<运算。
因此,就不能比较两个保存Sales_data元素的容器:

```
vector<Sales data> storeA,storeB;
if(storeA < storeB)//错误: Sales_data 没有<运算符
```

9.3顺序容器操作

顺序容器和关联容器的不同之处在于两者组织元素的方式。这些不同之处直接关系到了元素如何存储、访问、添加以及删除。上一节介绍了所有容器都支持的操作（罗列于表,9.2（第295页））。本章剩余部分将介绍顺序容器所特有的操作。

9.3.1向顺序容器添加元素

除array外,所有标准库容器都提供灵活的内存管理。在运行时可以动态添加或删除元素来改变容器大小。表9.5列出了向顺序容器(非array)添加元素的操作。

表9.5:向顺序容器添加元素的操作

这些操作会改变容器的大小;array不支持这些操作。
forward_list 有自己专有版本的insert和emplace;参见9.3.4节(第312页)。
forward_list 不支持push back和emplace back。
vector和string不支持push_front和emplace front。

操作	描述
c.push_back(t)	在c的尾部创建一个值为t或由args创建的元素。返回void
c.emplace_back(args)	同 c.push_back(t)
c.push_front(t)	在c的头部创建一个值为t或由args创建的元素。返回void
c.emplace_front(args)	同c.push_front(t)
c.insert(p,t)	在迭代器p指向的元素之前创建一个值为t或由args创建的元素。 返回指向新添加的元素的迭代器
c.emplace(p,args)	同c.insert(p,t)
c.insert(p,n,t)	在迭代器p指向的元素之前插入n个值为t的元素。 返回指向新添加的第一个元素的迭代器;若n为0,则返回p
c.insert(p,b,e)	将迭代器b和e指定的范围内的元素插入到迭代器p指向的元素之前。 b和e不能指向c中的元素。返回指向新添加的第一个元素的迭代器; 若范围为空,则返回p

操作	描述
c.insert(p,i1)	il是一个花括号包围的元素值列表。 将这些给定值插入到迭代器p指向的元素之前。 返回指向新添加的第一个元素的迭代器;若列表为空,则返回p

> **WARNING**
> 向一个vector、string或deque插入元素会使所有指向容器的选代器、引用和指针失效。

当我们使用这些操作时,必须记得不同容器使用不同的策略来分配元素空间,而这些策略直接影响性能。在一个vector或string的尾部之外的任何位置,或是一个deque的首尾之外的任何位置添加元素,都需要移动元素。而且,向一个vector或string添加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存,并将元素从旧的空间移动到新的空间中。

使用push_back

在3.3.2节(第90页)中,我们看到push_back将一个元素追加到一个vector的尾部。除array和forward 1ist之外,每个顺序容器(包括string类型)都支持push_back。

例如,下面的循环每次读取一个string到word中,然后追加到容器尾部:

```
//从标准输入读取数据,将每个单词放到容器末尾
string word;while(cin>>word)
container.push_back(word);
```

对push back的调用在container尾部创建了一个新的元素,将container的size增大了1。该元素的值为word的一个拷贝。container的类型可以是list、vector或deque。

由于string是一个字符容器,我们也可以用push back在string末尾添加字符:

```
void pluralize(size_t cnt,string &word){
if(cnt>1)
word.push_back('s');//等价于word+='s'
}
```

> **关键概念:容器元素是拷贝**
> 当我们用一个对象来初始化容器时,或将一个对象插入到容器中时,实际上放入到容器中的是对象值的一个拷贝,而不是对象本身。就像我们将一个对象传递给非引用参数(参见3.2.2节,第79页)一样,容器中的元素与提供值的对象之间没有任何关联。随后对容器中元素的任何改变都不会影响到原始对象,反之亦然。

使用push_front除了push_back,list、forward 1ist和deque容器还支持名为push_front的类似操作。此操作将元素插入到容器头部:

```
1ist<int>ilist;//将元素添加到ilist开头
for(sizet ix=0;ix! =4;++ix)
ilist.push_front(ix);
```

此循环将元素0、1、2、3添加到ilist头部。每个元素都插入到1ist的新的开始位置(new beginning)。即,当我们插入1时,它会被放置在0之前,2被放置在1之前,依此类推。因此,在循环中以这种方式将元素添加到容器中,最终会形成逆序。在循环执行完毕后,ilist保存序列3、2、1、0。

注意,deque像 vector一样提供了随机访问元素的能力,但它提供了vector所不支持的push_front。deque保证在容器首尾进行插入和删除元素的操作都只花费常数时间。与vector一样,在deque首尾之外的位置插入元素会很耗时。

在容器中的特定位置添加元素

push back和push_front操作提供了一种方便地在顺序容器尾部或头部插入单个元素的方法。insert成员提供了更一般的添加功能,它允许我们在容器中任意位置插入0个或多个元素。vector、deque、list和string都支持insert成员。forward_list提供了特殊版本的insert成员,我们将在9.3.4节(第312页)中介绍。

每个insert函数都接受一个迭代器作为其第一个参数。迭代器指出了在容器中什么位置放置新元素。它可以指向容器中任何位置,包括容器尾部之后的下一个位置。由于迭代器可能指向容器尾部之后不存在的元素的位置,而且在容器开始位置插入元素是很有用的功能,所以insert函数将元素插入到迭代器所指定的位置之前。例如,下面的语句

```
slist.insert(iter,"Hello! ");//将“Hel1o! ”添加到iter之前的位置
```

将一个值为“He11o”的string插入到iter指向的元素之前的位置。

虽然某些容器不支持push_front操作,但它们对于insert操作并无类似的限制(插入开始位置)。因此我们可以将元素插入到容器的开始位置,而不必担心容器是否支持push_front:

```
vector<string>svec;list<string>slist;
//等价于调用slist.push_front("Hello! ");
slist.insert(slist.begin(),"Hello! ");
//vector不支持push_front,但我们可以插入到begin()之前
//警告:插入到vector末尾之外的任何位置都可能很慢
svec.insert(svec.begin(),"Hello! ");
```

> **WARNING**
> 将元素插入到vector、deque和string中的任何位置都是合法的。然而,RN色这样做可能很耗时。

插入范围内元素

除了第一个迭代器参数之外,insert函数还可以接受更多的参数,这与容器构造函数类似。其中一个版本接受一个元素数目和一个值,它将指定数量的元素添加到指定位置之前,这些元素都按给定值初始化:

```
svec.insert(svec.end(),10,"Anna");
```

这行代码将10个元素插入到svec的末尾,并将所有元素都初始化为string"Anna”。

接受一对迭代器或一个初始化列表的insert版本将给定范围中的元素插入到指定位置之前:

```
vector<string>v={"quasi","simba","frollo","scar");
//将v的最后两个元素添加到slist的开始位置
slist.insert(slist.begin(),v.end()-2,v.end());
slist.insert(slist.end(),{"these","words","will",
"go","at","the","end"});
//运行时错误：选代器表示要拷贝的范围,不能指向与目的位置相同的容器
slist.insert(slist.begin(),slist.begin(),slist.end());
```

如果我们传递给insert一对选代器,它们不能指向添加元素的目标容器。

C++11

在新标准下,接受元素个数或范围的insert版本返回指向第一个新加入元素的选代t器。（在旧版本的标准库中,这些操作返回void。）如果范围为空,不插入任何元素,insert操作会将第一个参数返回。

使用insert的返回值

通过使用insert的返回值,可以在容器中一个特定位置反复插入元素：

```
list<string> lst;
auto iter = lst.begin();
while(cin >> word)
iter = lst.insert(iter,word);//等价于调用push_front
```

> **Note**
> 理解这个循环是如何工作的非常重要,特别是理解这个循环为什么等价于调用,push_front 尤为重要。

在循环之前,我们将iter初始化为1st.begin（）。第一次调用insert会将我们刚刚读入的string插入到iter所指向的元素之前的位置。insert返回的选代器恰好指向这个新元素。我们将此选代器赋予iter并重复循环,读取下一个单词。只要继续有单词读入,每步while循环就会将一个新元素插入到iter之前,并将iter改变为新加入元素的位置。此元素为（新的）首元素。因此,每步循环将一个新元素插入到list首元素之前的位置。

使用 emplace操作

C++11

新标准引入了三个新成员--emplace_front,emplace和emplace_back,这些操作构造而不是拷贝元素。这些操作分别对应push_front,insert和push_back,允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

当调用push或insert成员函数时,我们将元素类型的对象传递给它们,这些对象被拷贝到容器中。而当我们调用一个emplace成员函数时,则是将参数传递给元素类型的构造函数。emplace成员使用这些参数在容器管理的内存空间中直接构造元素。例如,假定c保存Sales_data（参见7.1.4节,第237页）元素:

```
//在c的末尾构造一个Sales data对象
//使用三个参数的Sales data构造函数
c.emplace_back("978-0590353403",25,15.99);
//错误: 没有接受三个参数的push_back版本
c.push_back("978-0590353403",25,15.99);
//正确: 创建一个临时的Sales_data对象传递给push_back
c.push_back(Sales_data("978-0590353403",25,15.99));
```

其中对emplace_back的调用和第二个push_back调用都会创建新的Sales_data对象。在调用emplace_back时,会在容器管理的内存空间中直接创建对象。而调用push_back则会创建一个局部临时对象,并将其压入容器中。

emplace函数的参数根据元素类型而变化,参数必须与元素类型的构造函数相匹配:

```
//iter指向c中一个元素,其中保存了Sales_data元素
c.emplace_back();//使用Sales_data的默认构造函数
c.emplace(iter,"999-999999999");//使用Sales_data(string)
//使用Sales_data的接受一个ISBN、一个count和一个price的构造函数
c.emplace_front("978-0590353403",25,15.99);
```

> **Note**
> emplace函数在容器中直接构造元素。传递给emplace函数的参数必须与元Nate 素类型的构造函数相匹配.

9.3.2访问元素

表9.6列出了我们可以用来在顺序容器中访问元素的操作。如果容器中没有元素,访问操作的结果是未定义的。

包括array在内的每个顺序容器都有一个front成员函数,而除forward_list之外的所有顺序容器都有一个back成员函数。这两个操作分别返回首元素和尾元素的引

```
//在解引用一个迭代器或调用front或back之前检查是否有元素
if(.c.empty()){
//val和va12是c中第一个元素值的拷贝
auto val=*c.begin(),val2=c.front();
//va13和val4是c中最后一个元素值的拷贝
auto last=c.end();auto val3=*(--last);//不能递减forward_list 迭代器
auto val4=c.back();//forward_list不支持
}
```

此程序用两种不同方式来获取c中的首元素和尾元素的引用。直接的方法是调用front和back。而间接的方法是通过解引用begin返回的迭代器来获得首元素的引用,以及通过递减然后解引用end返回的迭代器来获得尾元素的引用。

这个程序有两点值得注意:迭代器end指向的是容器尾元素之后的(不存在的)元素。为了获取尾元素,必须首先递减此迭代器。另一个重要之处是,在调用front和back(或解引用begin和end返回的迭代器)之前,要确保c非空。如果容器为空,if中操作的行为将是未定义的。

表9.6:在顺序容器中访问元素的操作

at和下标操作只适用于string、vector、deque和array。
back不适用于forward_list。

操作	描述
c.back()	返回c中尾元素的引用。若c为空,函数行为未定义
c.front()	返回c中首元素的引用。若c为空,函数行为未定义
c[n]	返回c中下标为n的元素的引用,n是一个无符号整数。若n>=c.size(),则函数行为未定义
c.at(n)	返回下标为n的元素的引用。如果下标越界,则抛出一out_of_range异常

> **WARNING**
> 对一个空容器调用front和back,就像使用一个越界的下标一样,是一种严重的程序设计错误。

访问成员函数返回的是引用

在容器中访问元素的成员函数(即,front、back、下标和at)返回的都是引用。如果容器是一个const对象,则返回值是const的引用。如果容器不是const的,则返回值是普通引用,我们可以用来改变元素的值:

```
if(! c.empty()){
c.front()=42;//将42赋予c中的第一个元素
auto&v=c.back();//获得指向最后一个元素的引用
v=1024;//改变c中的元素
auto v2=c.back();//v2不是一个引用,它是c.back()的一个拷贝
v2=0;//未改变c中的元素
}
```

与往常一样,如果我们使用auto变量来保存这些函数的返回值,并且希望使用此变量来改变元素的值,必须记得将变量定义为引用类型。

下标操作和安全的随机访问

提供快速随机访问的容器(string、vector、deque和array)也都提供下标运算符(参见3.3.3节,第91页)。就像我们已经看到的那样,下标运算符接受一个下标参数,返回容器中该位置的元素的引用。给定下标必须“在范围内”(即,大于等于0,且小于容器的大小)。保证下标有效是程序员的责任,下标运算符并不检查下标是否在合法范围内。使用越界的下标是一种严重的程序设计错误,而且编译器并不检查这种错误。如果我们希望确保下标是合法的,可以使用at成员函数。at成员函数类似下标运算符,但如果下标越界,at会抛出一个out_of_range异常(参见5.6节,第173页):

```
vector<string>svec;//空vector
cout <<svec[0];//运行时错误:svec中没有元素!
cout<<svec.at(0);//抛出一个out_of_range异常
```

9.3.3删除元素

与添加元素的多种方式类似,(非array)容器也有多种删除元素的方式。表9.7列出了这些成员函数。

表9.7:顺序容器的删除操作

这些操作会改变容器的大小,所以不适用于array。
forward_list 有特殊版本的erase,参见9.3.4节(第312页)。
forward_list不支持pop back;vector和string不支持pop_front。

操作	描述
c.pop_back()	删除c中尾元素。若c为空,则函数行为未定义。函数返回void
c.pop_front()	删除c中首元素。若c为空,则函数行为未定义。函数返回void
c.erase(p)	删除迭代器p所指定的元素,返回一个指向被删元素之后元素的迭代器, 若p指向尾元素,则返回尾后(off-the-end)迭代器。若p是尾后迭代器, 则函数行为未定义

操作	描述
c.erase(b,e)	删除迭代器b和e所指定范围内的元素。 返回一个指向最后一个被删元素之后元素的迭代器,若e本身就是尾后迭代器, 则函数也返回尾后迭代器
c.clear()	删除c中的所有元素。返回void

> **WARNING**
> 删除deque中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向vector或string中删除点之后位置的迭代器、引用和指针都会失

> **WARNING**
> 删除元素的成员函数并不检查其参数。在删除元素之前,程序员必须确保它(们)是存在的。

pop_front和pop_back成员函数

pop_front和pop_back成员函数分别删除首元素和尾元素。与vector和string不支持push_front一样,这些类型也不支持pop_front。类似的,forward_list不支持pop_back。与元素访问成员函数类似,不能对一个空容器执行弹出操作。

这些操作返回void。如果你需要弹出的元素的值,就必须在执行弹出操作之前保存它:

```
while(lilist.empty()){
process(ilist.front());//对ilist的首元素进行一些处理
ilist.pop_front();//完成处理后删除首元素
}
```

从容器内部删除一个元素

成员函数erase从容器中指定位置删除元素。我们可以删除由一个迭代器指定的单个元素,也可以删除由一对迭代器指定的范围内的所有元素。两种形式的erase都返回指向删除的(最后一个)元素之后位置的迭代器。即,若j是i之后的元素,那么erase(i)将返回指向j的迭代器。

例如,下面的循环删除一个1ist中的所有奇数元素:

```
1ist<int>lst={0,1,2,3,4,5,6,7,8,9};
auto it=lst.begin();while(it! =lst.end())
if(*it%2)//若元素为奇数
it=lst.erase(it);//删除此元素
else
++it;
```

每个循环步中,首先检查当前元素是否是奇数。如果是,就删除该元素,并将it设置为我们所删除的元素之后的元素。如果*it为偶数,我们将it递增,从而在下一步循环检查下一个元素。

删除多个元素

接受一对迭代器的erase版本允许我们删除一个范围内的元素:

```
//删除两个迭代器表示的范围内的元素
//返回指向最后一个被删元素之后位置的迭代器
elem1=slist.erase(elem1,elem2);//调用后,elem1==elem2
```

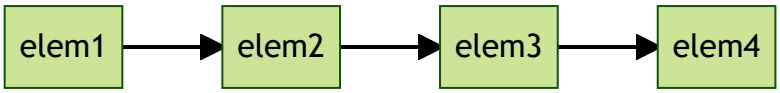
迭代器elem1指向我们要删除的第一个元素,elem2指向我们要删除的最后一个元素之后的位置。

为了删除一个容器中的所有元素,我们既可以调用clear,也可以用begin和end获得的迭代器作为参数调用erase:

```
slist.clear();//删除容器中所有元素
slist.erase(slist.begin(),slist.end());//等价调用
```

9.3.4 特殊的forward_list操作

为了理解 forward_list为什么有特殊版本的添加和删除操作,考虑当我们从一个单向链表中删除一个元素时会发生什么。如图9.1所示,删除一个元素会改变序列中的链接。在此情况下,删除elems会改变elem2,elem2原来指向elem3,但删除elems后,elemz指向了elem4。



删除elem3会改变elem2的值

图9.1：forward list的特殊操作

当添加或删除一个元素时,删除或添加的元素之前的那个元素的后继会发生改变。为了添加或删除一个元素,我们需要访问其前驱,以便改变前驱的链接。但是,forward_list是单向链表。在一个单向链表中,没有简

单的方法来获取一个元素的前驱。出于这个原因,在一个forward_1ist中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。这样,我们总是可以访问到被添加或删除操作所影响的元素。

由于这些操作与其他容器上的操作的实现方式不同,forward_list并未定义insert、emplace和erase,而是定义了名为insert_after、emplace_after和erase_after的操作(参见表9.8)。例如,在我们的例子中,为了删除elem,应该用指向elem,的迭代器调用erase_after。为了支持这些操作,forward_list也定义了before_begin,它返回一个首前(off-the-beginning)迭代器。这个迭代器允许我们在链表首元素之前并不存在的元素“之后”添加或删除元素(亦即在链表首元素之前添加删除元素)。

表9.8:在forward_list中插入或删除元素的操作

操作	描述
lst.before_begin()	返回指向链表首元素之前不存在的元素的迭代器。此迭代器不能解引用。 cbefore_begin()返回一个const iterator
lst.cbefore_begin()	同上
lst.insert_after(p,t)	在迭代器p之后的位置插入元素。 t是一个对象,n是数量,b和e是表示范围的一对迭代器 (b和e不能指向lst内),il是一个花括号列表。 返回一个指向最后一个插入元素的迭代器。如果范围为空,则返回p。 若p为尾后迭代器,则函数行为未定义
lst.insert_after(p,n,t)	同上
lst.insert_after(p,b,e)	同上
lst.insert_after(p,i1)	同上
emplace_after(p,args)	使用args在p指定的位置之后创建一个元素。 返回一个指向这个新元素的迭代器。若p为尾后迭代器,则函数行为未定义
lst.erase_after(p)	删除p指向的位置之后的元素,或删除从b之后直到 (但不包含)e之间的元素。返回一个指向被删元素之后元素的迭代器, 若不存在这样的元素,则返回尾后迭代器。 如果p指向lst的尾元素或者是一个尾后迭代器,则函数行为未定义
lst.erase_after(b,e)	同上

当在forward_1ist中添加或删除元素时,我们必须关注两个迭代器——一个指向我们要处理的元素,另一个指向其前驱。例如,可以改写第312页中从1ist中删除奇数元素的循环程序,将其改为从forward 1ist中删除元素:

```
forward_list<int>flst={0,1,2,3,4,5,6,7,8,9};
auto prev=flst.before_begin();//表示flst的“首前元素”
auto curr=flst.begin();//表示flst中的第一个元素
while(curr! =flst.end()){//仍有元素要处理
if(*curr%2)//若元素为奇数
curr=flst.erase_after(prev);//删除它并移动curr
else{
prev=curr;//移动迭代器curr,指向下一个元素,prev指向
++curr;//curr之前的元素
}
}
```

此例中,curr表示我们要处理的元素,prev表示curr的前驱。调用begin来初始化curr,这样第一步循环就会检查第一个元素是否是奇数。我们用before_begin来初始化prev,它返回指向curr之前不存在的元素的迭代器。

当找到奇数元素后,我们将prev 传递给erase_after。此调用将prev之后的元素删除,即,删除curr指向的元素。然后我们将curr重置为erase_after的返回值,使得curr指向序列中下一个元素,prev保持不变,仍指向(新)curr之前的元素。如果curr指向的元素不是奇数,在else中我们将两个迭代器都向前移动。

9.3.5改变容器大小

如表9.9所描述,我们可以用resize来增大或缩小容器,与往常一样,array不支持resize。如果当前大小大于所要求的大小,容器后部的元素会被删除;如果当前大小小于新大小,会将新元素添加到容器后部:

```
list<int>ilist(10,42);//10个int:每个的值都是42
ilist.resize(15);//将5个值为0的元素添加到ilist的末尾
ilist.resize(25,-1);//将10个值为-1的元素添加到ilist的末尾
ilist.resize(5);//从ilist末尾删除20个元素
```

resize操作接受一个可选的元素值参数,用来初始化添加到容器中的元素。如果调用者未提供此参数,新元素进行值初始化(参见3.3.1节,第88页)。如果容器保存的是类类型元素,且resize向容器添加新元素,则我们必须提供初始值,或者元素类型必须提供一个默认构造函数。

表9.9:顺序容器大小操作

resize 不适用于array

操作	描述
c.resize(n)	调整c的大小为n个元素。若 n < c.size(),则多出的元素被丢弃。若必须添加新元素,对新元素进行值初始化
c.resize(n,t)	调整c的大小为n个元素。任何新添加的元素都初始化为值t

> **WARNING**
> 如果resize缩小容器,则指向被删除元素的迭代器、引用和指针都会失效;对vector、string 或deque进行resize 可能导致迭代器、指针和引用失效。

9.3.6容器操作可能使迭代器失效

向容器中添加元素和从容器中删除元素的操作可能会使指向容器元素的指针、引用或迭代器失效。一个失效的指针、引用或迭代器将不再表示任何元素。使用失效的指针、引用或迭代器是一种严重的程序设计错误,很可能引起与使用未初始化指针一样的问题(参见2.3.2节,第49页)

在向容器添加元素后:

- 如果容器是vector或string,且存储空间被重新分配,则指向容器的迭代器、指针和引用都会失效。如果存储空间未重新分配,指向插入位置之前的元素的迭代器、指针和引用仍有效,但指向插入位置之后元素的迭代器、指针和引用将会失效。·对于deque,插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素,迭代器会失效,但指向存在的元素的引用和指针不会失效。
- 对于1ist和forward_list,指向容器的迭代器(包括尾后迭代器和首前迭代器)、指针和引用仍有效。
当我们从一个容器中删除元素后,指向被删除元素的迭代器、指针和引用会失效,这应该不会令人惊讶。毕竟,这些元素都已经被销毁了。当我们删除一个元素后:
- 对于1ist和forward_list,指向容器其他位置的迭代器(包括尾后迭代器和首前迭代器)、引用和指针仍有效。
- 对于deque,如果在首尾之外的任何位置删除元素,那么指向被删除元素外其他元素的迭代器、引用或指针也会失效。如果是删除deque的尾元素,则尾后迭代器也会失效,但其他迭代器、引用和指针不受影响;如果是删除首元素,这些也不会受影响。
- 对于vector和string,指向被删元素之前元素的迭代器、引用和指针仍有效。

注意:当我们删除元素时,尾后迭代器总是会失效。

> **WARNING**
> 使用失效的迭代器、指针或引用是严重的运行时错误。

> **建议:管理迭代器**
> 当你使用迭代器(或指向容器元素的引用或指针)时,最小化要求迭代器必须保持有效的程序片段是一个好的方法。
> 由于向迭代器添加元素和从迭代器删除元素的代码可能会使迭代器失效,因此必须保证每次改变容器的操作之后都正确地重新定位送代器。这个建议对vector、string和deque尤为重要。

编写改变容器的循环程序

添加/删除 vector、string 或deque元素的循环程序必须考虑迭代器、引用和指针可能失效的问题。程序必须保证每个循环步中都更新迭代器、引用或指针。如果循环中调用的是insert或erase,那么更新迭代器很容易。这些操作都返回迭代器,我们可以用来更新:

```
//傻瓜循环,删除偶数元素,复制每个奇数元素
vector<int>vi={0,1,2,3,4,5,6,7,8,9};
auto iter=vi.begin();//调用begin而不是cbegin,因为我们要改变vi
while(iter! =vi.end()){
if(*iter%2){
iter=vi.insert(iter,*iter);//复制当前元素
iter+=2;//向前移动迭代器,跳过当前元素以及插入到它之前的元素
}else iter=vi.erase(iter);//删除偶数元素
//不应向前移动迭代器,iter指向我们删除的元素之后的元素
}
```

此程序删除 vector中的偶数值元素,并复制每个奇数值元素。我们在调用insert和erase后都更新迭代器,因为两者都会使迭代器失效。

在调用erase后,不必递增迭代器,因为erase返回的迭代器已经指向序列中下一个元素。调用insert后,需要递增迭代器两次。记住,insert在给定位置之前插入新元素,然后返回指向新插入元素的迭代器。因此,在调用insert后,iter指向新插入元素,位于我们正在处理的元素之前。我们将迭代器递增两次,恰好越过了新添加的元素和正在处理的元素,指向下一个未处理的元素。

不要保存end返回的迭代器

当我们添加/删除 vector或string的元素后,或在deque中首元素之外任何位置添加/删除元素后,原来end返回的迭代器总是会失效。因此,添加或删除元素的循环程序必须反复调用end,而不能在循环之前保存end返回的迭代器,一直当作容器末尾使用。通常C++标准库的实现中end()操作都很快,部分就是因为这个原因。

例如,考虑这样一个循环,它处理容器中的每个元素,在其后添加一个新元素。我们希望循环能跳过新添加的元素,只处理原有元素。在每步循环之后,我们将定位迭代器,使其指向下一个原有元素。如果我们试图“优化”这个循环,在循环之前保存end()返回的迭代器,一直用作容器末尾,就会导致一场灾难:

```
//灾难:此循环的行为是未定义的
auto begin=v.begin(),end=v.end();//保存尾迭代器的值是一个坏主意
while(begin! =end){
//做一些处理
//插入新值,对begin重新赋值,否则的话它就会失效
++begin;//向前移动begin,因为我们想在此元素之后插入元素
begin=v.insert(begin,42);//插入新值
++begin;//向前移动begin跳过我们刚刚加入的元素
}
```

此代码的行为是未定义的。在很多标准库实现上,此代码会导致无限循环。问题在于我们将end操作返回的迭代器保存在一个名为end的局部变量中。在循环体中,我们向容器中添加了一个元素,这个操作使保存在end中的迭代器失效了。这个迭代器不再指向v中任何元素,或是v中尾元素之后的位置。

> **Tip**
> 如果在一个循环中插入/删除deque、string或vector中的元素,不要缓Tn/存end返回的迭代器。

必须在每次插入操作后重新调用end(),而不能在循环开始前保存它返回的迭代器:

```
//更安全的方法:在每个循环步添加/删除元素后都重新计算end
while(begin! =v.end()){
//做一些处理
++begin;//向前移动begin,因为我们想在此元素之后插入元素
begin=v.insert(begin,42);//插入新值
++begin;//向前移动begin,跳过我们刚刚加入的元素
}
```

9.4 vector对象是如何增长的

为了支持快速随机访问,vector将元素连续存储一—每个元素紧挨着前一个元素存储。通常情况下,我们不必关心一个标准库类型是如何实现的,而只需关心它如何使用。然而,对于vector和string,其部分实现渗透到了接口中。

假定容器中元素是连续存储的,且容器的大小是可变的,考虑向vector或string中添加元素会发生什么:如果没有空间容纳新元素,容器不可能简单地将它添加到内存中其他位置一—因为元素必须连续存储。容器必须分配新的内存空间来保存已有元素和新元素,将已有元素从旧位置移动到新空间中,然后添加新元素,释放旧存储空间。如果我们每添加一个新元素,vector就执行一次这样的内存分配和释放操作,性能会慢到不可接受。

为了避免这种代价,标准库实现者采用了可以减少容器空间重新分配次数的策略。当不得不获取新的内存空间时,vector和string的实现通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用,可用来保存更多的新元素。这样,就不需要每次添加新元素都重新分配容器的内存空间了。

这种分配策略比每次添加新元素时都重新分配容器内存空间的策略要高效得多。其实际性能也表现得足够好——虽然vector在每次重新分配内存空间时都要移动所有元素,但使用此策略后,其扩张操作通常比1ist和deque还要快。

管理容量的成员函数

如表9.10所示,vector和string类型提供了一些成员函数,允许我们与它的实现中内存分配部分互动。capacity操作告诉我们容器在不扩张内存空间的情况下可以容纳多少个元素。reserve操作允许我们通知容器它应该准备保存多少个元素。

表9.10:容器大小管理操作

shrink_to_fit只适用于vector、string和deque。
capacity和 reserve 只适用于vector和string。

操作	描述
c.shrink_to_fit()	请将 capacity()减少为与size()相同大小
c.capacity()	不重新分配内存空间的话,c可以保存多少元素
c.reserve(n)	分配至少能容纳n个元素的内存空间

> **Note**
> reserve 并不改变容器中元素的数量,它仅影响vector预先分配多大的内存空间。

只有当需要的内存空间超过当前容量时,reserve 调用才会改变 vector的容量。如果需求大小大于当前容量,reserve至少分配与需求一样大的内存空间(可能更大)。

如果需求大小小于或等于当前容量,reserve什么也不做。特别是,当需求大小小于当前容量时,容器不会退回内存空间。因此,在调用reserve之后,capacity将会大于或等于传递给 reserve的参数。

这样,调用reserve永远也不会减少容器占用的内存空间。类似的,resize成员函数(参见9.3.5节,第314页)只改变容器中元素的数目,而不是容器的容量。我们同样不能使用resize来减少容器预留的内存空间。

在新标准库中,我们可以调用shrink_to_fit 来要求deque、vector或string退回不需要的内存空间。此函数指出我们不再需要任何多余的内存空间。但是,具体的实现可以选择忽略此请求。也就是说,调用shrink_to_fit也并不保证一定退回内存空间。

capacity 和size

理解 capacity和size的区别非常重要。容器的size是指它已经保存的元素的数目;而capacity则是在不分配新的内存空间的前提下它最多可以保存多少元素。

下面的代码展示了size和capacity之间的相互作用:

```
vector<int>ivec;
//size应该为0;capacity的值依赖于具体实现
cout<<"ivec:size:"<<ivec.size()
<<"capacity:"<< ivec.capacity()<<endl;
//向ivec添加24个元素
for(vector<int>::size_type ix=0;ix! =24;++ix)
ivec.push_back(ix);

//size应该为24;capacity应该大于等于24,具体值依赖于标准库实现
cout <<"ivec:size:"<< ivec.size()
<<"capacity:"<<ivec.capacity()<<end1
```

当在我们的系统上运行时,这段程序得到如下输出:

```
ivec:size:0 capacity:0
ivec:size:24 capacity:32
```

我们知道一个空vector的size为0,显然在我们的标准库实现中一个空vector的capacity也为0。当向vector中添加元素时,我们知道size与添加的元素数目相等。而 capacity至少与size一样大,具体会分配多少额外空间则视标准库具体实现而定。在我们的标准库实现中,每次添加1个元素,共添加24个元素,会使capacity变为32。

可以想象ivec的当前状态如下图所示:

0	1	2	...	23	保留空间	-
					ivec.size()	ivec.capacity()

现在可以预分配一些额外空间:

```
ivec.reserve(50);//将capacity至少设定为50,可能会更大
//size应该为24;capacity应该大于等于50,具体值依赖于标准库实现
cout<<"ivec:size:"<<ivec.size()
<<"capacity:"<< ivec.capacity()<<endl;
```

程序的输出表明reserve严格按照我们需求的大小分配了新的空间:

```
ivec:size:24 capacity:50
```

接下来可以用光这些预留空间:

```
//添加元素用光多余容量
while(ivec.size()! =ivec.capacity())
ivec.push back(0);
//capacity应该未改变,size和capacity不相等
cout <<"ivec:size:"<< ivec.size()
<<"capacity:"<<ivec.capacity()<<endl;
```

程序输出表明此时我们确实用光了预留空间,size和capacity相等:

```
vec:size:50 capacity:50
```

由于我们只使用了预留空间,因此没有必要为vector分配新的空间。实际上,只要没有操作需求超出vector的容量,vector 就不能重新分配内存空间。

如果我们现在再添加一个新元素,vector就不得不重新分配空间:

```
ivec.push back(42);//再添加一个元素
//size应该为51;capacity应该大于等于51,具体值依赖于标准库实现
cout<<"ivec:size:"<< ivec.size()
<<"capacity:"<<ivec.capacity()<<endl;
```

这段程序的输出为

```
ivec:size:51 capacity:100
```

这表明 vector的实现采用的策略似乎是在每次需要分配新内存空间时将当前容量翻倍。

可以调用shrink to_fit 来要求vector将超出当前大小的多余内存退回给系统:

```
ivec.shrink to fit();//要求归还内存
//size应该未改变;capacity的值依赖于具体实现
cout<<"ivec:size:"<<ivec.size()
<<"capacity:"<< ivec.capacity()<< endl;
```

调用shrink_to_fit只是一个请求,标准库并不保证退还内存。

> **Note**
> 每个vector实现都可以选择自己的内存分配策略。但是必须遵守的一条原则是:只有当迫不得已时才可以分配新的内存空间。

只有在执行insert 操作时size与capacity相等,或者调用resize或reserve时给定的大小超过当前capacity,vector才可能重新分配内存空间。会分配多少超过给定容量的额外空间,取决于具体实现。

虽然不同的实现可以采用不同的分配策略,但所有实现都应遵循一个原则:确保用push_back向vector添加元素的操作有高效率。从技术角度说,就是通过在一个初始为空的vector 上调用n次push back来创建一个n个元素的vector,所花费的时间不能超过n的常数倍。

9.5额外的string操作

除了顺序容器共同的操作之外,string类型还提供了一些额外的操作。这些操作中的大部分要么是提供string类和C风格字符数组之间的相互转换,要么是增加了允许我们用下标代替迭代器的版本。

标准库string类型定义了大量函数。幸运的是,这些函数使用了重复的模式。由于函数过多,本节初次阅读可能令人心烦,因此读者可能希望快速浏览本节。当你了解string支持哪些类型的操作后,就可以在需要使用一个特定操作时回过头来仔细阅读。

9.5.1构造string的其他方法

除了我们在3.2.1节(第76页)已经介绍过的构造函数,以及与其他顺序容器相同的构造函数(参见表9.3,第299页)外,string类型还支持另外三个构造函数,如表9.11所示。

表9.11:构造string的其他方法

n、len2和pos2都是无符号值

方法	描述
string s(cp,n)	s是cp指向的数组中前n个字符的拷贝。此数组至少应该包含n个字符
strings(s2,pos2)	s是strings2从下标pos2开始的字符的拷贝。若pos2>s2.size(), 构造函数的行为未定义
string s(s2,pos2,1en2)	s是strings2从下标pos2开始1en2个字符的拷贝。若pos2>s2.size(), 构造函数的行为未定义。不管1en2的值是多少,构造函数至多拷贝s2.size()-pos2个字符

这些构造函数接受一个string或一个const char*参数,还接受(可选的)指定拷贝多少个字符的参数。当我们传递给它们的是一个string时,还可以给定一个下标来指出从哪里开始拷贝:

```
const char*cp="Hel1o World! ! ! ";//以空字符结束的数组
char noNull[]={'H','i'};//不是以空字符结束
string s1(cp);//拷贝cp中的字符直到遇到空字符;s1=="Hello World! ! ! "
string s2(noNul1,2);//从noNul1拷贝两个字符;s2=="Hi"
string s3(noNul1);//未定义:noNu11不是以空字符结束
string s4(cp+6,5);//从cp[6]开始拷贝5个字符;s4=="World"
string s5(s1,6,5);//从s1[6]开始拷贝5个字符;s5=="World"
string s6(s1,6);//从s1[6]开始拷贝,直至s1末尾;s6=="World! ! ! "
string s7(s1,6,20);//正确,只拷贝到s1末尾;s7=="World! ! ! "
string s8(s1,16);//抛出一个out_of range异常
```

通常当我们从一个const char*创建string时,指针指向的数组必须以空字符结尾,拷贝操作遇到空字符时停止。如果我们还传递给构造函数一个计数值,数组就不必以空字符结尾。如果我们未传递计数值且数组也未以空字符结尾,或者给定计数值大于数组大小,则构造函数的行为是未定义的。

当从一个string拷贝字符时,我们可以提供一个可选的开始位置和一个计数值。开始位置必须小于或等于给定的string的大小。如果位置大于size,则构造函数抛出一个out_of_range异常(参见5.6节,第173页)。如果我们传递了一个计数值,则从给定位置开始拷贝这么多个字符。不管我们要求拷贝多少个字符,标准库最多拷贝到string结尾,不会更多。

substr操作

substr操作(参见表9.12)返回一个string,它是原始string的一部分或全部的拷贝。可以传递给substr一个可选的开始位置和计数值:

```
string s("hello world");
string s2=s.substr(0,5);//s2=hel1o
string s3=s.substr(6);//s3=world
string s4=s.substr(6,11);//s3=world
string s5=s.substr(12);//抛出一个out_of_range异常
```

如果开始位置超过了string的大小,则substr 函数抛出一个out_of_range异常(参见5.6节,第173页)。如果开始位置加上计数值大于string的大小,则substr会调整计数值,只拷贝到string的末尾。

表9.12:子字符串操作

操作	描述
s.substr(pos,n)	返回一个string,包含s中从pos开始的n个字符的拷贝。pos的默认值为0。 n的默认值为s.size()-pos,即拷贝从pos开始的所有字符

9.5.2改变string的其他方法

string类型支持顺序容器的赋值运算符以及assign、insert 和erase操作(参见9.2.5节,第302页;9.3.1节,第306页;9.3.3节,第311页)。除此之外,它还定义了额外的insert和erase版本。

除了接受迭代器的insert和erase版本外,string还提供了接受下标的版本。下标指出了开始删除的位置,或是insert到给定值之前的位置:

```
s.insert(s.size(),5,'!');//在s末尾插入5个感叹号
s.erase(s.size()-5,5);//从s删除最后5个字符
```

标准库string类型还提供了接受C风格字符数组的insert和assign版本。例如,我们可以将以空字符结尾的字符数组insert到或assign给一个string:

```
const char*cp="Stately,plump Buck";
s.assign(cp,7);//s=="Stately"
s.insert(s.size(),cp+7);//s=="stately,plump Buck"
```

此处我们首先通过调用assign替换s的内容。我们赋予s的是从cp指向的地址开始的7个字符。要求赋值的字符数必须小于或等于cp指向的数组中的字符数(不包括结尾的空字符)。

接下来在s上调用insert,我们的意图是将字符插入到s[size()]处(不存在的)元素之前的位置。在此例中,我们将cp开始的7个字符(至多到结尾空字符之前)拷贝到s中。

我们也可以指定将来自其他string或子字符串的字符插入到当前string中或赋予当前string:

```
strings="some string",s2="some other string";
s.insert(0,s2);//在s中位置0之前插入s2的拷贝
//在s[0]之前插入s2中s2[0]开始的s2.size()个字符
s.insert(0,s2,0,s2.size());a
```

ppend和replace函数

string类定义了两个额外的成员函数:append和replace,这两个函数可以改变string的内容。表9.13描述了这两个函数的功能。append操作是在string末尾进行插入操作的一种简写形式:

```
strings("C++ Primer"),s2=s;//将s和s2初始化为"C++Primer"
s.insert(s.size(),"4th Ed.");//s=="C++ Primer 4th Ed."
s2.append("4th Ed.");//等价方法:将"4th Ed.”追加到s2;s==s2
```

replace 操作是调用erase和insert的一种简写形式:

```
//将“4th“替换为“5th”的等价方法
s.erase(11,3);//s=="C++ Primer Ed."
s.insert(11,"5th");//s=="C++ Primer 5th Ed."
//从位置11开始,删除3个字符并插入"5th"
s2.replace(11,3,"5th");//等价方法:s==s2
```

此例中调用replace时,插入的文本恰好与删除的文本一样长。这不是必须的,可以插入一个更长或更短的string:

```
s.replace(11,3,"Fifth");//s=="C++ Primer Fifth Ed."
```

在此调用中,删除了3个字符,但在其位置插入了5个新字符。

表9.13:修改string的操作

操作	描述
s.insert(pos,args)	在pos之前插入args指定的字符。pos可以是一个下标或一个迭代器。 接受下标的版本返回一个指向s的引用; 接受迭代器的版本返回指向第一个插入字符的迭代器
s.erase(pos,len)	删除从位置pos开始的1en个字符。如果1en被省略, 则删除从pos开始直至s末尾的所有字符。返回一个指向s的引用
s.assign(args)	将s中的字符替换为args指定的字符。返回一个指向s的引用
s.append(args)	将args追加到s。返回一个指向s的引用
s.replace(range,args)	删除s中范围range内的字符,替换为args指定的字符。 range或者是一个下标和一个长度,或者是一对指向s的迭代器。 返回一个指向s的引用

args可以是下列形式之一;append和assign可以使用所有形式。
str不能与s相同,迭代器b和e不能指向s。

操作	描述
str	字符串str
str,pos,1en	str中从pos开始最多1en个字cp,1en从cp指向的字符数组的前(最多)1en个字符
cp	cp指向的以空字符结尾的字符数组
b,e	迭代器b和e指定的范围内的字符

操作	描述
初始化列表	花括号包围的,以逗号分隔的字符列表

replace和insert所允许的args形式依赖于range和pos是如何指定的。

replace	replace	insert	insert	args可以是
(pos,len,args)	(b,e,args)	(pos,args)	(iter,args)	-
是	是	是	否	str
是	否	是	否	str,pos,len
是	是	是	否	cp,len
是	是	否	否	cp
是	是	是	是	n,c
否	是	否	是	b2,e2
否	是	否	是	初始化列表

改变string的多种重载函数

表9.13列出的append、assign、insert 和replace函数有多个重载版本。根据我们如何指定要添加的字符和string中被替换的部分,这些函数的参数有不同版本。幸运的是,这些函数有共同的接口。

assign和append 函数无须指定要替换string中哪个部分:assign总是替换string中的所有内容,append总是将新字符追加到string末尾。

replace函数提供了两种指定删除元素范围的方式。可以通过一个位置和一个长度来指定范围,也可以通过一个迭代器范围来指定。insert函数允许我们用两种方式指定插入点:用一个下标或一个迭代器。在两种情况下,新元素都会插入到给定下标(或迭代器)之前的位置。

可以用好几种方式来指定要添加到string中的字符。新字符可以来自于另一个string,来自于一个字符指针(指向的字符数组),来自于一个花括号包围的字符列表,或者是一个字符和一个计数值。当字符来自于一个string或一个字符指针时,我们可以传递一个额外的参数来控制是拷贝部分还是全部字符。

并不是每个函数都支持所有形式的参数。例如,insert就不支持下标和初始化列表参数。类似的,如果我们希望用迭代器指定插入点,就不能用字符指针指定新字符的来源。

9.5.3 string搜索操作

string类提供了6个不同的搜索函数,每个函数都有4个重载版本。表9.14描述了这些搜索成员函数及其参数。每个搜索操作都返回一个string:size_type值,表示匹配发生位置的下标。如果搜索失败,则返回一个名为string:npos的static成员(参见7.6节,第268页)。标准库将npos定义为一个const string:size_type类型,并初始化为值-1。由于npos是一个unsigned类型,此初始值意味着npos等于任何string最大的可能大小(参见2.1.2节,第32页)。

> **WARNING**
> string 搜索函数返回string::size_type值,该类型是一个unsigned类型。因此,用一个int或其他带符号类型来保存这些函数的返回值不是一个好主意(参见2.1.2节,第33页)。

find函数完成最简单的搜索。它查找参数指定的字符串,若找到,则返回第一个匹配位置的下标,否则返回npos:

```
string name("AnnaBelle");
auto posl=name.find("Anna");//posl==0
```

这段程序返回0,即子字符串"Anna"在"AnnaBe11e"中第一次出现的下标。

搜索(以及其他string操作)是大小写敏感的。当在string中查找子字符串时,要注意大小写:

```
string lowercase("annabelle");
pos1=lowercase.find("Anna");//pos1==npos
```

这段代码会将pos1置为npos,因为Anna与anna不匹配。

一个更复杂一些的问题是查找与给定字符串中任何一个字符匹配的位置。例如,下面代码定位name中的第一个数字:

```
string numbers("0123456789"),name("r2d2");
//返回1,即,name中第一个数字的下标
auto pos=name.find_first_of(numbers);
```

如果是要搜索第一个不在参数中的字符,我们应该调用find_first_not_of。例如,为了搜索一个string中第一个非数字字符,可以这样做:

```
string dept("03714p3");
//返回5—一字符'p'的下标
auto pos=dept.find first_not of(numbers);
```

表9.14:string搜索操作

搜索操作返回指定字符出现的下标,如果未找到则返回npos。

操作	描述
s.find(args)	查找s中args第一次出现的位置
s.rfind(args)	查找s中args最后一次出现的位置
s.find_first_of(args)	在s中查找args中任何一个字符第一次出现的位置。
s.find_last_of(args)	在s中查找args中任何一个字符最后一次出现的位置
s.find_first_not_of(args)	在s中查找第一个不在args中的字符
s.find_last_not_of(args)	在s中查找最后一个不在args中的字符

args必须是以下形式之一

操作	描述
c,pos	从s中位置pos开始查找字符c。pos默认为0
s2,pos	从s中位置pos开始查找字符串s2。pos默认为0
cp,pos	从s中位置pos开始查找指针cp指向的以空字符结尾的C风格字符串。pos默认为0
cp,pos,n	从s中位置pos开始查找指针cp指向的数组的前n个字符。pos和n无默认值

指定在哪里开始搜索

我们可以传递给find操作一个可选的开始位置。这个可选的参数指出从哪个位置开始进行搜索。默认情况下,此位置被置为0。一种常见的程序设计模式是用这个可选参数在字符串中循环地搜索子字符串出现的所有位置:

```
string::size type pos=0;
//每步循环查找name中下一个数
while((pos=name.find_first_of(numbers,pos))
! =string::npos){
cout <<"found number at index:"<<pos
<<"element is"<< name[pos]<<endl;
++pos;//移动到下一个字符
}
```

while的循环条件将pos重置为从pos开始遇到的第一个数字的下标。只要find_first_of返回一个合法下标,我们就打印当前结果并递增pos。

如果我们忽略了递增pos,循环就永远也不会终止。为了搞清楚原因,考虑如果不做递增运算会发生什么。在第二步循环中,我们从pos指向的字符开始搜索。这个字符是一个数字,因此find_first_of会(重复地)返回

pos!

逆向搜索

到现在为止,我们已经用过的find操作都是由左至右搜索。标准库还提供了类似的,但由右至左搜索的操作。rfind成员函数搜索最后一个匹配,即子字符串最靠右的出现位置:

```
string river("Mississippi");
auto first_pos=river.find("is");//返回1
auto last pos=river.rfind("is");//返回4
```

find返回下标1,表示第一个“is”的位置,而rfind返回下标4,表示最后一个“is”的位置。

类似的,find_last函数的功能与find_first函数相似,只是它们返回最后一个而不是第一个匹配:

- find_last_of搜索与给定string中任何一个字符匹配的最后一个字符。
- find_last_not_of搜索最后一个不出现在给定string中的字符。

每个操作都接受一个可选的第二参数,可用来指出从什么位置开始搜索。

9.5.4 compare函数

除了关系运算符外(参见3.2.2节,第79页),标准库string类型还提供了一组compare函数,这些函数与C标准库的strcmp函数(参见3.5.4节,第109页)很相似。类似strcmp,根据s是等于、大于还是小于参数指定的字符串,s.compare返回0、正数或负数。

如表9.15所示,compare有6个版本。根据我们是要比较两个string还是一个string与一个字符数组,参数各有不同。在这两种情况下,都可以比较整个或一部分字符串。

表9.15:s.compare的几种参数形式

参数	描述
s	比较s和s2
pos1,n1,s2	将s中从pos1开始的n1个字符与s2进行比较
pos1,n1,s2,pos2,n2	将s中从pos1开始的n1个字符与s2中从pos2开始的n2个字符进行比较
cp	比较s与cp指向的以空字符结尾的字符数组
pos1,n1,cp	将s中从pos1开始的n1个字符与cp指向的以空字符结尾的字符数组进行比较
pos1,n1,cp,n2	将s中从pos1开始的n1个字符与指针cp指向的地址开始的n2个字符进行比较

9.5.5数值转换

字符串中常常包含表示数值的字符。例如,我们用两个字符的string表示数值15一字符1’后跟字符5。一般情况,一个数的字符表示不同于其数值。数值15如果保存为16位的short类型,则其二进制位模式为0000000000001111,而字符串"15"存为两个Latin-1编码的char,二进制位模式为0011000100110101。第一个字节表示字符1’,其八进制值为061,第二个字节表示5,其Latin-1编码为八进制值065。

新标准引入了多个函数,可以实现数值数据与标准库string之间的转换:

```
inti=42;strings=to_string(i);//将整数i转换为字符表示形式
doubled=stod(s);//将字符串s转换为浮点数
```

此例中我们调用to_string将42转换为其对应的string表示,然后调用stod将此string转换为浮点值。

要转换为数值的string中第一个非空白符必须是数值中可能出现的字符:

```
string s2="pi=3.14";
//转换s中以数字开始的第一个子串,结果d=3.14
d=stod(s2.substr(s2.find_first_of("+-.0123456789")));
```

在这个stod调用中,我们调用了find first of(参见9.5.3节,第325页)来获得s中第一个可能是数值的一部分的字符的位置。我们将s中从此位置开始的子串传递给stod。stod函数读取此参数,处理其中的字符,直至遇到不可能是数值的一部分的字符。然后它就将找到的这个数值的字符串表示形式转换为对应的双精度浮点值。

string参数中第一个非空白符必须是符号(+或一)或数字。它可以以0x或0x开头来表示十六进制数。对那些将字符串转换为浮点值的函数,string参数也可以以小数点(.)开头,并可以包含e或E来表示指数部分。对于那些将字符串转换为整型值的函数,根据基数不同,string参数可以包含字母字符,对应大于数字9的数。

> **Note**
> 如果string 不能转换为一个数值,这些函数抛出一个invalid argument Nate异常(参见5.6节,第173页)。如果转换得到的数值无法用任何类型来表示,则抛出一个out_of_range异常。

表9.16:string和数值之间的转换

转换	操作
to_string(val)	一组重载函数,返回数值val的string表示。val可以是任何算术类型(参见2.1.1节,第30页)。对每个浮点类型和int或更大 的整型,都有相应版本的to string。 与往常一样,小整型会被提升(参见4.11.1节,第142页)

转换	操作
stoi(s,p,b)	返回s的起始子串(表示整数内容)的数值,返回值类型分别是int、long、unsigned long、1ong long、unsignedlonglong。b表示转换所用的基数,默认值为10。p是sizet指针,用来保存s中第一个非数值字符的下标,p默认为0,即,函数不保存下标
stol(s,p,b)	同上
stoul(s,p,b)	同上
stoll(s,P,b)	同上
stoull(s,p,b)	同上
stof(s,p)	返回s的起始子串(表示浮点数内容)的数值,返回值类型分别是float、double或1ong double。参数p的作用与整数转换函数中一样
stod(s,p)	同上
stold(s，p)	同上

9.6容器适配器

除了顺序容器外,标准库还定义了三个顺序容器适配器:stack、queue和priority_queue。适配器(adaptor)是标准库中的一个通用概念。容器、迭代器和函数都有适配器。本质上,一个适配器是一种机制,能使某种事物的行为看起来像另外一种事物一样。一个容器适配器接受一种已有的容器类型,使其行为看起来像一种不同的类型。例如,stack适配器接受一个顺序容器(除array或forward_list外),并使其操作起来像一个stack一样。表9.17列出了所有容器适配器都支持的操作和类型。

表9.17:所有容器适配器都支持的操作和类型

操作	描述
size_type	一种类型,足以保存当前类型的最大对象的大小
value_type	元素类型
container_type	实现适配器的底层容器类型
Aa;	创建一个名为a的空适配器
Aa(c);	创建一个名为a的适配器,带有容器c的一个拷贝
关系运算符	每个适配器都支持所有关系运算符:一、！=、<、<=、>和>= 这些运算符返回底层容器的比较结果

操作	描述
a.empty()	若a包含任何元素,返回false,否则返回true
a.size()	返回a中的元素数目
swap(a,b) a.swap(b)	交换a和b的内容,a和b必须有相同类型,包括底层容器类型也必须相同

定义一个适配器

每个适配器都定义两个构造函数:默认构造函数创建一个空对象,接受一个容器的构造函数拷贝该容器来初始化适配器。例如,假定deq是一个 `deque<int>` ,我们可以用deq来初始化一个新的stack,如下所示:

```
stack<int>stk(deq);//从deq拷贝元素到stk
```

默认情况下,stack和queue是基于deque实现的,priority_queue是在vector之上实现的。我们可以在创建一个适配器时将一个命名的顺序容器作为第二个类型参数,来重载默认容器类型。

```
//在vector上实现的空栈
stack<string,vector<string>>str stk;
//str_stk2在 vector上实现,初始化时保存svec的拷贝
stack<string,vector<string>>str stk2(svec);
```

对于一个给定的适配器,可以使用哪些容器是有限制的。所有适配器都要求容器具有添加和删除元素的能力。因此,适配器不能构造在array之上。类似的,我们也不能用forward_list 来构造适配器,因为所有适配器都要求容器具有添加、删除以及访问尾元素的能力。stack只要求push back、pop_back和back操作,因此可以使用除array和forward_list 之外的任何容器类型来构造stack。queue适配器要求back、push_back、front和push_front,因此它可以构造于1ist或deque之上,但不能基于vector构造。priority_queue除了front、push back和popback操作之外还要求随机访问能力,因此它可以构造于vector或deque之上,但不能基于list构造。

栈适配器

stack类型定义在stack头文件中。表9.18列出了stack所支持的操作。下面的程序展示了如何使用stack:

```
stack<int>intStack;//空栈
//填满栈
for(size_tix=0;ix! =10;++ix)
intStack.push(ix);//intStack保存0到9十个数
while(! intStack.empty()){//intStack中有值就继续循环
int value=intstack.top();
//使用栈顶值的代码
intStack.pop();//弹出栈顶元素,继续循环
}
```

其中,声明语句

```
stack<int>intStack;//空栈
```

定义了一个保存整型元素的栈intStack,初始时为空。for循环将10个元素添加到栈中,这些元素被初始化为从0开始连续的整数。while循环遍历整个stack,获取top值,将其从栈中弹出,直至栈空。

表9.18:表9.17未列出的栈操作

栈默认基于deque实现,也可以在list或vector之上实现。

操作	描述
s.pop()	删除栈顶元素,但不返回该元素值
s.push(item) s.emplace(args)	创建一个新元素压入栈顶,该元素通过拷贝或移动item而来, 或者由args构造
s.top()	返回栈顶元素,但不将元素弹出栈

每个容器适配器都基于底层容器类型的操作定义了自己的特殊操作。我们只可以使用适配器操作,而不能使用底层容器类型的操作。例如,

```
intstack.push(ix);//intStack保存0到9十个数
```

此语句试图在intStack的底层deque 对象上调用push back。虽然stack是基于deque实现的,但我们不能直接使用deque操作。不能在一个stack上调用push back,而必须使用stack自己的操作push。

队列适配器

queue和priority_queue适配器定义在queue头文件中。表9.19列出了它们所支持的操作。

表9.19:表9.17未列出的queue和priority_queue操作

queue 默认基于deque实现,priority_queue默认基于vector实现;
queue 也可以用list或vector 实现,priority_queue也可以用deque实现。

操作	描述
q.pop()	返回queue的首元素或priority_queue的最高优先级的元素,但不删除此元素
q.front()	返回首元素或尾元素,但不删除此元素
q.back()	只适用于queue
q.top()	返回最高优先级元素,但不删除该元素

只适用于priority_queue

操作	描述
q.push(item)	在queue 末尾或priority_queue中恰当的位置创建一个元素,其值为item, 或者由args构造
q.emplace(args)	同上

标准库queue使用一种先进先出(first-in,first-out,FIFO)的存储和访问策略。进入队列的对象被放置到队尾,而离开队列的对象则从队首删除。饭店按客人到达的顺序来为他们安排座位,就是一个先进先出队列的例子。

priority_queue允许我们为队列中的元素建立优先级。新加入的元素会排在所有优先级比它低的已有元素之前。饭店按照客人预定时间而不是到来时间的早晚来为他们安排座位,就是一个优先队列的例子。默认情况下,标准库在元素类型上使用<运算符来确定相对优先级。我们将在11.2.2节(第378页)学习如何重载这个默认设置。

术语表

- `适配器(adaptor)` 标准库类型、函数或迭代器,它们接受一个类型、函数或迭代器,使其行为像另外一个类型、函数或迭代器一样。标准库提供了三种顺序容器适配器:stack、queue和priority queue。每个适配器都在其底层顺序容器类型之上定义了一个新的接口。
- `数组(array)` 固定大小的顺序容器。为了定义一个array,除了元素类型之外还必须给定大小。array中的元素可以用其位置下标来访问。array支持快速的随机访问。
- `begin` 容器操作,返回一个指向容器首元素的迭代器,如果容器为空,则返回尾后迭代器。是否返回const迭代器依赖于容器的类型。
- `cbegin` 容器操作,返回一个指向容器首元素的const iterator,如果容器为空,则返回尾后迭代器。
- `cend` 容器操作,返回一个指向容器尾元素之后(不存在的)的const iterator。

- 容器(container) 保存一组给定类型对象的类型。每个标准库容器类型都是一个模板类型。为了定义一个容器,我们必须指定保存在容器中的元素的类型。除了array之外,标准库容器都是大小可变的。
- deque 顺序容器。deque中的元素可以通过位置下标来访问。支持快速的随机访问。deque各方面都与vector类似,唯一的差别是,deque支持在容器头尾位置的快速插入和删除,而且在两端插入或删除元素都不会导致重新分配空间。
- end 容器操作,返回一个指向容器尾元素之后(不存在的)元素的迭代器。是否返回const迭代器依赖于容器的类型。
- forward_list 顺序容器,表示一个单向链表。forwardlist中的元素只能顺序访问。从一个给定元素开始,为了访问另一个元素,我们只能遍历两者之间的所有元素。forward_list上的迭代器不支持递减运算(--)。forward_list 支持任意位置的快速插入(或删除)操作。与其他容器不同,插入和删除发生在一个给定的迭代器之后的位置。因此,除了通常的尾后迭代器之外,forward_list还有一个“首前”迭代器。在添加新元素后,原有的指向forwardlist的迭代器仍有效。在删除元素后,只有原来指向被删元素的迭代器才会失效。
- 选代器范围(iterator range) 由一对迭代器指定的元素范围。第一个迭代器表示序列中第一个元素,第二个迭代器指向最后一个元素之后的位置。如果范围为空,则两个迭代器是相等的(反之亦然,如果两个迭代器不等,则它们表示一个非空范围)。如果范围非空,则必须保证,通过反复递增第一个迭代器,可以到达第二个迭代器。通过递增迭代器,序列中每个元素都能被访问到。
- 左闭合区间(left-inclusive interval) 值范围,包含首元素,但不包含尾元素。通常表示为[i,j),表示序列从i开始(包含)直至结束(不包含)。
- list 顺序容器,表示一个双向链表。list中的元素只能顺序访问。从一个给定元素开始,为了访问另一个元素,我们只能遍历两者之间的所有元素。1ist上的迭代器既支持递增运算(++),也支持递减运算(--)。list支持任意位置的快速插入(或删除)操作。当加入新元素后,迭代器仍然有效。当删除元素后,只有原来指向被删除元素的迭代器才会失效。
- 首前迭代器(off-the-beginning iterator) 表示一个forwardlist开始位置之前(不存在的)元素的迭代器。是forward_list 的成员函数before begin的返回值。与end()迭代器类似,不能被解引用。
- 尾后迭代器(off-the-end iterator) 表示范围中尾元素之后位置的迭代器。通常被称为“末尾迭代器”(end iterator)。
- priority_queue 顺序容器适配器,生成一个队列,插入其中的元素不放在末尾,而是根据特定的优先级排列。默认情况下,优先级用元素类型上的小于运算符确定。
- queue 顺序容器适配器,生成一个类型,使我们能将新元素添加到末尾,从头部删除元素。
- 顺序容器(sequential container) 保存相同类型对象有序集合的类型。顺序容器中的元素通过位置来访问。
- stack 顺序容器适配器,生成一个类型,使我们只能在其一端添加和删除元素。
- vector 顺序容器。vector中的元素可以通过位置下标访问。支持快速的随机访问。我们只能在vector末尾实现高效的元素添加/删除。向 vector添加元素可能导致内存空间的重新分配,从而使所有指向vector的迭代器失效。在vector内部添加(或删除)元素会使所有指向插入(删除)点之后元素的迭代器失效。

primer学习

第10章泛型算法

标准库容器定义的操作集合惊人得小.标准库并未给每个容器添加大量功能,而是提供了一组算法,这些算法中的大多数都独立于任何特定的容器.这些算法是通用的(generic,或称泛型的)：它们可用于不同类型的容器和不同类型的元素.

泛型算法和关于迭代器的更多细节,构成了本章的主要内容.

顺序容器只定义了很少的操作：在多数情况下,我们可以添加和删除元素、访问首尾元素、确定容器是否为空以及获得指向首元素或尾元素之后位置的迭代器.

我们可以想象用户可能还希望做其他很多有用的操作：查找特定元素、替换或删除一个特定值、重排元素顺序等.

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.

10.1概述

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(参见9.2.1节,第296页)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.例如,假定我们有一个int的vector,希望知道vector中是否包含一个特定值.回答这个问题最方便的方法是调用标准库算法find：

```
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()
auto result=find(vec.cbegin(),vec.cend(),val);
//报告结果
cout<<"The value"<<val
<<(result==vec.cend()
? "is not present": "is present")<< endl;
```

传递给find的前两个参数是表示元素范围的迭代器,第三个参数是一个值.find将范围中每个元素与给定值进行比较.它返回指向第一个等于给定值的元素的迭代器.如果范围中无匹配元素,则find返回第二个参数

来表示搜索失败.因此,我们可以通过比较返回值和第二个参数来判断搜索是否成功.我们在输出语句中执行这个检测,其中使用了条件运算符(参见4.7节,第134页)来报告搜索是否成功.

由于find操作的是迭代器,因此我们可以用同样的find函数在任何容器中查找值.例如,可以用find在一个string的list中查找一个给定值:

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

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[]={27,210,12,47,109,83};
int val=83;
int*result=find(begin(ia),end(ia),val);
```

此例中我们使用了标准库begin和end函数(参见3.5.3节,第106页)来获得指向ia中首元素和尾元素之后位置的指针,并传递给find.

还可以在序列的子范围中查找,只需将指向子范围首元素和尾元素之后位置的迭代器(指针)传递给find.例如,下面的语句在ia[1]、ia[2]和ia[3]中查找给定元素:

```
//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);
```

算法如何工作

为了弄清这些算法如何用于不同类型的容器,让我们更近地观察一下find.find的工作是在一个未排序的元素序列中查找一个特定元素.概念上,find应执行如下步骤:

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,重复执行步骤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,它应该返回一个指出元素未找到的值.此值和步骤3返回的值必须具有相容的类型.

这些步骤都不依赖于容器所保存的元素类型.因此,只要有一个迭代器可用来访问元素,find就完全不依赖于容器类型(甚至无须理会保存元素的是不是容器).

迭代器令算法不依赖于容器,……

在上述find函数流程中,除了第2步外,其他步骤都可以用迭代器操作来实现：利用迭代器解引用运算符可以实现元素访问;如果发现匹配元素,find可以返回指向该元素的迭代器;用迭代器递增运算符可以移动到下一个元素;尾后迭代器可以用来判断find是否到达给定序列的末尾;find可以返回尾后迭代器(参见9.2.1节,第296页)来表示未找到给定元素.

……,但算法依赖于元素类型的操作

虽然迭代器的使用令算法不依赖于容器类型,但大多数算法都使用了一个(或多个)元素类型上的操作.例如,在步骤2中,find用元素类型的一运算符完成每个元素与给定值的比较.其他算法可能要求元素类型支持<运算符.不过,我们将会看到,大多数算法提供了一种方法,允许我们使用自定义的操作来代替默认的运算符.

> **关键概念：算法永远不会执行容器的操作**
> 泛型算法本身不会执行容器的操作,它们只会运行于迭代器之上,执行迭代器的操作.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,也可能在容器内移动元素,但永远不会直接添加或删除元素.

如我们将在10.4.1节(第358页)所看到的,标准库定义了一类特殊的迭代器,称为插入器(inserter).与普通送代器只能遍历所绑定的容器相比,插入器能做更多的事情.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,当一个算法操作一个这样的迭代器时,迭代器可以完成向容器添加元素的效果,但算法自身永远不会做这样的操作.

10.2初识泛型算法

标准库提供了超过100个算法.幸运的是,与容器类似,这些算法有一致的结构.比起死记硬背全部100多个算法,理解此结构可以帮助我们更容易地学习和使用这些算法.在本章中,我们将展示如何使用这些算法,并介绍刻画了这些算法的统一原则.附录A按操作方式列出了所有算法.

除了少数例外,标准库算法都对一个范围内的元素进行操作.我们将此元素范围称为“输入范围”.接受输入范围的算法总是使用前两个参数来表示此范围,两个参数分别是指向要处理的第一个元素和尾元素之后位置的迭代器.

虽然大多数算法遍历输入范围的方式相似,但它们使用范围中元素的方式不同.理解算法的最基本的方法就是了解它们是否读取元素、改变元素或是重排元素顺序.

10.2.1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.find就是这样一种算法,我们在10.1节练习(第337页)中使用的count函数也是如此.

另一个只读算法是accumulate,它定义在头文件numeric中.accumulate函数接受三个参数,前两个指出了需要求和的元素的范围,第三个参数是和的初值.假定vec是一个整数序列,则:

```
//对vec中的元素求和,和的初值是0
int sum=accumulate(vec.cbegin(),vec.cend(),0);
```

这条语句将sum设置为vec中元素的和,和的初值被设置为0.

Note accumulate的第三个参数的类型决定了函数中使用哪个加法运算符以及返回值的类型.

算法和元素类型

accumulate将第三个参数作为求和起点,这蕴含着一个编程假定:将元素类型加到和的类型上的操作必须是可行的.即,序列中元素的类型必须与第三个参数匹配,或者能够转换为第三个参数的类型.在上例中,vec中的元素可以是int,或者是double、long long或任何其他可以加到int上的类型.

下面是另一个例子,由于string定义了+运算符,所以我们可以通过调用accumulate来将vector中所有string元素连接起来:

```
string sum=accumulate(v.cbegin(),v.cend(),string(""));
```

此调用将v中每个元素连接到一个string上,该string初始时为空串.注意,我们通过第三个参数显式地创建了一个string.将空串当做一个字符串字面值传递给第三个参数是不可以的,会导致一个编译错误.

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

原因在于,如果我们传递了一个字符串字面值,用于保存和的对象的类型将是const char*.如前所述,此类型决定了使用哪个+运算符.由于const char*并没有+运算符,此调用将产生编译错误.

> **Best Prastices**
> 对于只读取而不改变元素的算法,通常最好使用cbegin()和cend()(参见 9.2.3节,第298页).但是,如果你计划使用算法返回的迭代器来改变元素的值,就需要使用begin()和end()的结果作为参数.

操作两个序列的算法

另一个只读算法是equa1,用于确定两个序列是否保存相同的值.它将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.如果所有对应元素都相等,则返回true,否则返回false.此算法接受三个迭代器:前两个(与以往一样)表示第一个序列中的元素范围,第三个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(),rosterl.cend(),roster2.cbegin());
```

由于equa1利用迭代器完成操作,因此我们可以通过调用equa1来比较两个不同类型的容器中的元素.而且,元素类型也不必一样,只要我们能用一来比较两个元素类型即可.例如,在此例中,roster1可以是 `vector<string>` ,而roster2是 `list<const char*>` .

但是,equa1基于一个非常重要的假设：它假定第二个序列至少与第一个序列一样长.此算法要处理第一个序列中的每个元素,它假定每个元素在第二个序列中都有一个与之对应的元素.

> **WARNING**
> 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4八与第一个序列一样长.

10.2.2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,必须注意确保序列原大小至少不小于我们要求算法写入的元素数目.记住,算法不会执行容器操作,因此它们自身不可能改变容器的大小.

一些算法会自己向输入范围写入元素.这些算法本质上并不危险,它们最多写入与给定序列一样多的元素.

例如,算法fi11接受一对迭代器表示一个范围,还接受一个值作为第三个参数.fil1将给定的这个值赋予输入序列中的每个元素.

```
fi1l(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
fi1l(vec.begin(),vec.begin()+vec.size()/2,10);
```

> **关键概念：迭代器参数**
> 由于fi11向给定输入序列中写入数据,因此,只要我们传递了一个有效的输入序列,写入操作就是安全的.
> 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.例如,第一个序列可能保存于一个vector中,而第二个序列可能保存于一个list、deque、内置数组或其他容器中.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.例如,对equal算法,元素类型不要求相同,但是我们必须能使用一来比较来自两个序列中的元素.

操作两个序列的算法之间的区别在于我们如何传递第二个序列.一些算法,例如equa1,接受三个迭代器：前两个表示第一个序列的范围,第三个表示第二个序列中的首元素.其他算法接受四个迭代器：前两个表示第一个序列的元素范围,后两个表示第二个序列的范围.

用一个单一迭代器表示第二个序列的算法都假定第二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—equa1会试图访问第二个序列中末尾之后(不存在)的元素.

算法不检查写操作

一些算法接受一个迭代器来指出一个单独的目的位置.这些算法将新值赋予一个序列中的元素,该序列从目的位置迭代器指向的元素开始.例如,函数fi11n接受一个单迭代器、一个计数值和一个值.它将给定值赋予迭代器指向的元素开始的指定个元素.我们可以用fi11_n将一个新值赋予vector中的元素:

```
vector<int>vec;//空vector
//使用vec,赋予它不同值
fi11_n(vec.begin(),vec.size(),0);//将所有元素重置为0
```

函数fi11_n假定写入指定个元素是安全的.即,如下形式的调用

```
fi1l_n(dest,n,val)
```

fi11_n假定dest指向一个元素,而从dest开始的序列至少包含n个元素.

一个初学者非常容易犯的错误是在一个空容器上调用fi11_n(或类似的写元素的算法):

```
vector<int>vec;//空向量
//灾难: 修改vec中的10个(不存在)元素
fi11 n(vec.begin(),10,0);
```

这个调用是一场灾难.我们指定了要写入10个元素,但vec中并没有元素——它是空的.这条语句的结果是未定义的.

> **WARNING**
> 向目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.

介绍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 iterator).插入迭代器是一种向容器中添加元素的迭代器.通常情况,当我们通过一个迭代器向容器元素赋值时,值被赋予迭代器指向的元素.而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.

我们将在10.4.1节中(第358页)详细介绍插入迭代器的内容.但是,为了展示如何用算法向容器写入数据,我们现在将使用back_inserter,它是定义在头文件iterator中的一个函数.

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.当我们通过此迭代器赋值时,赋值运算符会调用push back将一个具有给定值的元素添加到容器中:

```
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
```

我们常常使用back_inserter来创建一个迭代器,作为算法的目的位置来使用.例如:

```
vector<int>vec;//空向量
//正确: backinserter创建一个插入迭代器,可用来向vec添加元素
fi11_n(back_inserter(vec),10,0);//添加10个元素到vec
```

在每步迭代中,fi11n向给定序列的一个元素赋值.由于我们传递的参数是back_inserter返回的迭代器,因此每次赋值都会在vec上调用push back.最终,这条fi11n调用语句向vec的末尾添加了10个元素,每个元素的值都是0.

拷贝算法

拷贝(copy)算法是另一个向目的位置迭代器指向的输出序列中的元素写入数据的算法.此算法接受三个迭代器,前两个表示一个输入范围,第三个表示目的序列的起始位置.此算法将输入范围中的元素拷贝到目的序列中.传递给copy的目的序列至少要包含与输入序列一样多的元素,这一点很重要.

我们可以用copy实现内置数组的拷贝,如下面代码所示:

```
int al[]={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(al),a2);//把a1的内容拷贝给a2
```

此例中我们定义了一个名为a2的数组,并使用sizeof确保a2与数组a1包含同样多的元素(参见4.9节,第139页).接下来我们调用copy完成从a1到a2的拷贝.在调用copy后,两个数组中的元素具有相同的值.

copy返回的是其目的位置迭代器(递增后)的值.即,ret恰好指向拷贝到a2的尾元素之后的位置.

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.

例如,replace算法读入一个序列,并将其中所有等于给定值的元素都改为另一个值.此算法接受4个参数:前两个是迭代器,表示输入序列,后两个一个是要搜索的值,另一个是新值.它将所有等于第一个值的元素替换为第二个值:

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

此调用将序列中所有的0都替换为42.如果我们希望保留原序列不变,可以调用replace_copy.此算法接受额外第三个迭代器参数,指出调整后序列的保存位置:

```
//使用back inserter按需要增长目标序列
replace copy(ilst.cbegin(),ilst.cend(),back inserter(ivec),0,42);
```

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42.

10.2.3重排容器元素的算法

某些算法会重排容器中元素的顺序,一个明显的例子是sort.调用sort会重排输入序列中的元素,使之有序,它是利用元素类型的<运算符来实现排序的.

例如,假定我们想分析一系列儿童故事中所用的词汇.假定已有一个vector,保存了多个故事的文本.我们希望化简这个vector,使得每个单词只出现一次,而不管单词在任意给定文档中到底出现了多少次.

为了便于说明问题,我们将使用下面简单的故事作为输入:

the quick red fox jumps over the slow red turtle

给定此输入,我们的程序应该生成如下vector:

fox	jumps	over	quick	red	slow	the	turtle

消除重复单词

为了消除重复单词,首先将vector排序,使得重复的单词都相邻出现.一旦vector排序完毕,我们就可以使用另一个称为unique的标准库算法来重排vector,使得不重复的元素出现在vector的开始部分.由于算法不能执行容器的操作,我们将使用vector的erase成员来完成真正的删除操作:

```
void elimpups(vector<string>&words)
{
//按字典序排序words,以便查找重复单词
sort(words.begin(),words.end());
//unique重排输入范围,使得每个单词只出现一次
//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());
//使用向量操作erase删除重复单词
words.erase(end unique,words.end());
}
```

sort算法接受两个迭代器,表示要排序的元素范围.在此例中,我们排序整个vector.完成sort后,words的顺序如下所示:

fox	jumps	over	quick	red	red	slow	the	the	turtle

注意,单词red和the各出现了两次.

使用unique

words排序完毕后,我们希望将每个单词都只保存一次.unique算法重排输入序列,将相邻的重复项“消除”,并返回一个指向不重复值范围末尾的迭代器.调用unique后,vector将变为:

fox	jumps	over	lquick	red	slow	the	turtle	？？？	？？？
								end unique	
								(最后一个不重复元素之后的位置)	

words的大小并未改变,它仍有10个元素.但这些元素的顺序被改变了——相邻的重复元素被“删除”了.我们将删除打引号是因为unique并不真的删除任何元素,它只是覆盖相邻的重复元素,使得不重复元素出现在序列开始部分.unique返回的迭代器指向最后一个不重复元素之后的位置.此位置之后的元素仍然存在,但我们不知道它们的值是什么.

> **Note**
> 标准库算法对迭代器而不是容器进行操作.因此,算法不能(直接)添加或删除元素.

使用容器操作删除元素

为了真正地删除无用元素,我们必须使用容器操作,本例中使用erase(参见9.3.3节,第311页).我们删除从end unique开始直至words末尾的范围内的所有元素.这个调用之后,words包含来自输入的8个不重复的单词.

值得注意的是,即使words中没有重复单词,这样调用erase也是安全的.在此情况下,unique会返回words.end().因此,传递给erase的两个参数具有相同的值：words.end().迭代器相等意味着传递给erase的元素范围为空.删除一个空范围没有什么不良后果,因此程序即使在输入中无重复元素的情况下也是正确的.

10.3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<或一运算符完成比较.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算法默认使用元素类型的<运算符.但可能我们希望的排序顺序与<所定义的顺序不同,或是我们的序列可能保存的是未定义<运算符的元素类型(如Sales data).在这两种情况下,都需要重载sort的默认行为.

10.3.1向算法传递函数

作为一个例子,假定希望在调用elimDups(参见10.2.3节,第343页)后打印vector的内容.此外还假定希望单词按其长度排序,大小相同的再按字典序排列.为了按长度重排vector,我们将使用sort的第二个版本,此版本是重载过的,它接受第三个参数,此参数是一个谓词(predicate).

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(第378页)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.我们在6.2.2节(第189页)中定义的isshorter就是一个满足这些要求的函数,因此可以将isshorter传递给sort.这样做会将元素按大小重新排序：

```
//比较函数,用来按长度排序单词
bool isshorter(const string&sl,const string&s2)
{
return sl.size()<s2.size();
}
//按长度由短至长排序words
sort(words.begin(),words.end(),isshorter);
```

如果words包含的数据与10.2.3节(第343页)中一样,此调用会将words重排,使得所有长度为3的单词排在长度为4的单词之前,然后是长度为5的单词,依此类推.

排序算法

在我们将words按大小重排的同时,还希望具有相同长度的元素按字典序排列.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.

通常情况下,我们不关心有序序列中相等元素的相对顺序,它们毕竟是相等的.但是,在本例中,我们定义的“相等”关系表示“具有相同长度”.而具有相同长度的元素,如果看其内容,其实还是各不相同的.通过调用stable_sort,可以保持等长元素间的字典序：

```
elimDups(words);//将words按字典序重排,并消除重复单词
//按长度重新排序,长度相同的单词维持字典序
stable sort(words.begin(),words.end(),isshorter);
for(const auto&s:words)//无须拷贝字符串
cout<<s<<"";//打印每个元素,以空格分隔
cout<<end1;
```

假定在此调用前 words是按字典序排列的,则调用之后,words会按元素大小排序,而长度相同的单词会保持字典序.如果我们对原来的vector内容运行这段代码,输出为：

fox red the over slow jumps quick turtle

10.3.2 lambda表达式

根据算法接受一元谓词还是二元谓词,我们传递给算法的谓词必须严格接受一个或两个参数.但是,有时我们希望进行的操作需要更多参数,超出了算法对谓词的限制.例如,为上一节最后一个练习所编写的程序中,就必须将大小5硬编码到划分序列的谓词中.如果在编写划分序列的谓词时,可以不必为每个可能的大小都编写一个独立的谓词,显然更有实际价值.

一个相关的例子是,我们将修改10.3.1节(第345页)中的程序,求大于等于一个给定长度的单词有多少.我们还会修改输出,使程序只打印大于等于给定长度的单词.

我们将此函数命名为biggies,其框架如下所示:

```
void biggies(vector<string>&words,vector<string>: : size type sz)
{
elimpups(words);//将 words按字典序排序,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(),isshorter);
//获取一个迭代器,指向第一个满足size()>=sz的元素
//计算满足size>=sz的元素的数目
//打印长度大于等于给定值的单词,每个单词后面接一个空格
}
```

我们的新问题是在vector中寻找第一个大于等于给定长度的元素.一旦找到了这个元素,根据其位置,就可以计算出有多少元素的长度大于等于给定值.

我们可以使用标准库find_if算法来查找第一个具有特定大小的元素.类似find(参见10.1节,第336页),find_if算法接受一对迭代器,表示一个范围.但与find不同的是,findif的第三个参数是一个谓词.findif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.

编写一个函数,令其接受一个string和一个长度,并返回一个bool值表示该string的长度是否大于给定长度,是一件很容易的事情.但是,find_if接受一元谓词一我们传递给find_if的任何函数都必须严格接受一个参数,以便能用来自输入序列的一个元素调用它.没有任何办法能传递给它第二个参数来表示长度.为了解决此问题,需要使用另外一些语言特性.

介绍lambda

我们可以向一个算法传递任何类别的可调用对象(callable object).对于一个对象或一个表达式,如果可以对其使用调用运算符(参见1.5.2节,第21页),则称它为可调用的.即,如果e是一个可调用的表达式,则我们可以编写代码e(args),其中args是一个逗号分隔的一个或多个参数的列表.

到目前为止,我们使用过的仅有的两种可调用对象是函数和函数指针(参见6.7节,第221页).还有其他两种可调用对象：重载了函数调用运算符的类,我们将在14.8节(第506页)介绍,以及lambda表达式(lambda expression).

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{fiunction body}` 其中,capture list(捕获列表)是一个lambda所在函数中定义的局部变量的列表(通常为空);return toype、parameter list 和function body与任何普通函数一样,分别表示返回类型、参数列表和函数体.但是,与普通函数不同,lambda必须使用尾置返回(参见6.3.3节,第206页)来指定返回类型.

我们可以忽略参数列表和返回类型,但必须永远包含捕获列表和函数体

```
autof=[]{return 42;};
```

此例中,我们定义了一个可调用对象f,它不接受参数,返回42.

lambda的调用方式与普通函数的调用方式相同,都是使用调用运算符：

```
cout<<f()<<endl;//打印42
```

在lambda中忽略括号和参数列表等价于指定一个空参数列表.在此例中,当调用f时,参数列表是空的.如果忽略返回类型,lambda根据函数体中的代码推断出返回类型.如果函数体只是一个return语句,则返回类型从返回的表达式的类型推断而来.否则,返回类型为void.

> **Note**
> 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.

向lambda传递参数

与一个普通函数调用类似,调用一个lambda时给定的实参被用来初始化lambda的形参.通常,实参和形参的类型必须匹配.但与普通函数不同,lambda不能有默认参数(参见6.5.1节,第211页).因此,一个lambda调用的实参数目永远与形参数目相等.一旦形参初始化完毕,就可以执行函数体了.

作为一个带参数的lambda的例子,我们可以编写一个与isshorter函数完成相同功能的lambda：

```
[](const string&a,const string&b)
{return a.size()<b.size();}
```

空捕获列表表明此lambda不使用它所在函数中的任何局部变量.lambda的参数与isshorter的参数类似,是const string的引用.lambda的函数体也与isshorter类似,比较其两个参数的size(),并根据两者的相对大小返回一个布尔值.

如下所示,可以使用此lambda 来调用stable_sort:

```
//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();});
```

当stable_sort 需要比较两个元素时,它就会调用给定的这个lambda表达式.

使用捕获列表

我们现在已经准备好解决原来的问题了一—编写一个可以传递给find_if的可调用表达式.我们希望这个表达式能将输入序列中每个string的长度与biggies函数中的sz参数的值进行比较.

虽然一个lambda可以出现在一个函数中,使用其局部变量,但它只能使用那些明确指明的变量.一个lambda通过将局部变量包含在其捕获列表中来指出将会使用这些变量.捕获列表指引lambda在其内部包含访问局部变量所需的信息.

在本例中,我们的lambda会捕获sz,并只有单一的string参数.其函数体会将string的大小与捕获的sz的值进行比较:

```
[sz](const string&a)
{return a.size()>=sz;};
```

lambda以一对[]开始,我们可以在其中提供一个以逗号分隔的名字列表,这些名字都是它所在函数中定义的.

由于此lambda捕获sz,因此lambda的函数体可以使用sz.lambda不捕获words,因此不能访问此变量.如果我们给lambda提供一个空捕获列表,则代码会编译错误:

```
//错误: sz未捕获
[](const string&a)
{return a.size()>=sz;};
```

Note 一个lambda 只有在其捕获列表中捕获一个它所在函数中的局部变量,才能在函数体中使用该变量.

调用find if

使用此lambda,我们就可以查找第一个长度大于等于sz的元素:

```
//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;});
```

这里对find_if的调用返回一个迭代器,指向第一个长度不小于给定参数sz的元素.如果这样的元素不存在,则返回words.end()的一个拷贝.

我们可以使用find_if返回的迭代器来计算从它开始到words的末尾一共有多少个元素(参见3.4.2节,第99页):

```
//计算满足size>=sz的元素的数目
auto count=words.end()-wc;
cout << count<<""<<make plural(count,"word","s")
<<"of length"<<sz<<"or longer"<<endl;
```

我们的输出语句调用make_plural(参见6.3.2节,第201页)来输出“word”或“words”,具体输出哪个取决于大小是否等于1.

for_each 算法

问题的最后一部分是打印words中长度大于等于sz的元素.为了达到这一目的,我们可以使用for_each算法.此算法接受一个可调用对象,并对输入序列中每个元素调用此对象:

```
//打印长度大于等于给定值的单词,每个单词后面接一个空格
for_each(wc,words.end(),
[](const string ss){cout<<s<<"";});
cout<<end1;
```

此lambda中的捕获列表为空,但其函数体中还是使用了两个名字：s和cout,前者是它自己的参数.

捕获列表为空,是因为我们只对lambda所在函数中定义的(非static)变量使用捕获列表.一个lambda可以直接使用定义在当前函数之外的名字.在本例中,cout不是定义在biggies中的局部名字,而是定义在头文件iostream中.因此,只要在biggies出现的作用域中包含了头文件iostream,我们的lambda就可以使用cout.

> **Note**
> 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.

完整的biggies

到目前为止,我们已经解决了程序的所有细节,下面就是完整的程序:

```
void biggies(vector<string>&words,vector<string>: : size_type sz)
{
elimDups(words);//将words按字典序排序,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 sa,const string sb)
{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 << count<<""<< make plural(count,"word","s")
<<"of length"<<sz<<"or longer"<<endl;
//打印长度大于等于给定值的单词,每个单词后面接一个空格
for each(wc,words.end(),
[](const string&s){cout<<s<<"";});
cout<<endl;
}
```

10.3.3 lambda捕获和返回

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.我们将在14.8.1节(第507页)介绍这种类是如何生成的.目前,可以这样理解,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：传递的参数就是此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,定义了一个从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.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.

值捕获

类似参数传递,变量的捕获方式也可以是值或引用.表10.1(第352页)列出了几种不同的构造捕获列表的方式.到目前为止,我们的lambda采用值捕获的方式.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：

```
void fcn1()
{
sizet v1=42;//局部变量
//将v1拷贝到名为f的可调用对象
autof=[vl]{return v1;};
v1=0;
autoj=f();//j为42;f保存了我们创建它时v1的拷贝
```

由于被捕获变量的值是在lambda创建时拷贝,因此随后对其修改不会影响到lambda内对应的值.

引用捕获

我们定义lambda时可以采用引用方式捕获变量.例如:

```
void fcn2()
{
sizetv1=42;//局部变量
//对象f2包含v1的引用
auto f2=[6v1]{return v1;};
v1=0;
autoj=f2();//j为0;f2保存v1的引用,而非拷贝
```

v1之前的s指出v1应该以引用方式捕获.一个以引用方式捕获的变量与其他任何类型的引用的行为类似.当我们在lambda函数体内使用此变量时,实际上使用的是引用所绑定的对象.在本例中,当lambda返回v1时,它返回的是v1指向的对象的值.

引用捕获与返回引用(参见6.3.2节,第201页)有着相同的问题和限制.如果我们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.

引用捕获有时是必要的.例如,我们可能希望biggies函数接受一个ostream的引用,用来输出数据,并接受一个字符作为分隔符:

```
void biggies(vector<string>&words,vector<string>: : size type sz,ostream&os=cout,charc='')
{
//与之前例子一样的重排words的代码
//打印count的语句改为打印到os
for each(words.begin(),words.end(),
[sos,c](const string&s){os<<s<<c;});
}
```

我们不能拷贝ostream对象(参见8.1.1节,第279页),因此捕获os的唯一方法就是捕获其引用(或指向os的指针).

当我们向一个函数传递一个lambda时,就像本例中调用for_each那样,1ambda会立即执行.在此情况下,以引用方式捕获os没有问题,因为当for_each执行时,biggies中的变量是存在的.

我们也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.

> **WARNING**
> 当以引用方式捕获一个变量时,必须保证在lambda执行时变量是存在的.

> **建议：尽量保持lambda的变量捕获简单化**
> 一个lambda捕获从lambda被创建(即,定义lambda的代码执行时)到lambda自身执行(可能有多次执行)这段时间内保存的相关信息.确保lambda每次执行的时候这些信息都有预期的意义,是程序员的责任.

捕获一个普通变量,如int、string或其他非指针类型,通常可以采用简单的值捕获方式.在此情况下,只需关注变量在捕获时是否有我们所需的值就可以了.

如果我们捕获一个指针或迭代器,或采用引用捕获方式,就必须确保在lambda执行时,绑定到迭代器、指针或引用的对象仍然存在.而且,需要保证对象具有预期的值.在lambda从创建到它执行的这段时间内,可能有代码改变绑定的对象的值.也就是说,在指针(或引用)被捕获的时刻,绑定的对象的值是我们所期望的,但在lambda执行时,该对象的值可能已经完全不同了.

一般来说,我们应该尽量减少捕获的数据量,来避免潜在的捕获导致的问题.而且,如果可能的话,应该避免捕获指针或引用.

隐式捕获

除了显式列出我们希望使用的来自所在函数的变量之外,还可以让编译器根据lambda体中的代码来推断我们要使用哪些变量.为了指示编译器推断捕获列表,应在捕获列表中写一个&或=.s告诉编译器采用捕获引用方式,=则表示采用值捕获方式.例如,我们可以重写传递给findif的lambda:

```
//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;});
```

如果我们希望对一部分变量采用值捕获,对其他变量采用引用捕获,可以混合使用隐式捕获和显式捕获:

```
void biggies(vector<string>&words,
vector<string>: : size type sz,ostream sos=cout,charc='')
//其他处理与前例一样
//os隐式捕获,引用捕获方式;c显式捕获,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;});
```

当我们混合使用隐式捕获和显式捕获时,捕获列表中的第一个元素必须是一个&或=.此符号指定了默认捕获方式为引用或值.

当混合使用隐式捕获和显式捕获时,显式捕获的变量必须使用与隐式捕获不同的方式.即,如果隐式捕获是引用方式(使用了s),则显式捕获命名变量必须采用值方式,因比不能在其名字前使用s.类似的,如果隐式捕获采用的是值方式(使用了=),则显式捕获命名变量必须采用引用方式,即,在名字前使用&.

表10.1：lambda 捕获列表

[]	空捕获列表.lambda不能使用所在函数中的变量. 一个lambda只有捕获变量后才能使用它们
[names]	names是一个逗号分隔的名字列表,这些名字都是lambda所在函数的局部变量. 默认情况下,捕获列表中的变量都被拷贝.名字前如果使用了&, 则采用引用捕获方式
[&]	隐式捕获列表, 采用引用捕获方式.lambda体中所使用的来自所在函数的实体都采用引用方式使用
[=]	隐式捕获列表, 采用值捕获方式.lambda体将拷贝所使用的来自所在函数的实体的值
[&,identifier_list]	identifier_list是一个逗号分隔的列表,包含0个或多个来自所在函数的变量. 这些变量采用值捕获方式, 而任何隐式捕获的变量都采用引用方式捕获.identifier_list中的名字前面不能使用&
[=,identifier_list]	identifier_list中的变量都采用引用方式捕获, 而任何隐式捕获的变量都采用值方式捕获.identifier_list中的名字不能包括this, 且这些名字之前必须使用&

可变lambda

默认情况下,对于一个值被拷贝的变量,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.因此,可变lambda能省略参数列表:

```
void fcn3()
{
sizetv1=42;//局部变量
//f可以改变它所捕获的变量的值
autof=[v1]()mutable{return ++v1;};
v1=0;
autoj=f();//j为43
}
```

一个引用捕获的变量是否(如往常一样)可以修改依赖于此引用指向的是一个const类型还是一个非const类型:

```
void fcn4()
{
sizetv1=42;//局部变量
//v1是一个非const变量的引用
//可以通过f2中的引用来改变它
auto f2=[&v1]{return ++vl;};
v1=0;
auto j=f2();//j为1
}
```

指定lambda返回类型

到目前为止,我们所编写的lambda都只包含单一的return语句.因此,我们还未遇到必须指定返回类型的情况.默认情况下,如果一个lambda体包含return之外的任何语句,则编译器假定此lambda返回void.与其他返回void的函数类似,被推断返回void的lambda不能返回值.

下面给出了一个简单的例子,我们可以使用标准库transform算法和一个lambda来将一个序列中的每个负数替换为其绝对值:

```
transform(vi.begin(),vi.end(),vi.begin(),
[](int i){returni<0? -i:i;});
```

函数transform接受三个迭代器和一个可调用对象.前两个迭代器表示输入序列,第三个迭代器表示目的位置.算法对输入序列中每个元素调用可调用对象,并将结果写到目的位置.如本例所示,目的位置迭代器与表示输入序列开始位置的迭代器可以是相同的.当输入迭代器和目的迭代器相同时,transform将输入序列中每个元素替换为可调用对象操作该元素得到的结果.

在本例中,我们传递给transform一个lambda,它返回其参数的绝对值.lambda体是单一的return语句,返回一个条件表达式的结果.我们无须指定返回类型,因为可以根据条件运算符的类型推断出来.

但是,如果我们将程序改写为看起来是等价的if语句,就会产生编译错误:

```
//错误:不能推断1ambda的返回类型
transform(vi.begin(),vi.end(),vi.begin(),
[](int i){if(i<0)return-i;else return i;});
```

编译器推断这个版本的lambda返回类型为void,但它返回了一个int值.

当我们需要为一个lambda定义返回类型时,必须使用尾置返回类型(参见6.3.3节,第206页):

```
transform(vi.begin(),vi.end(),vi.begin(),
[](int i)->int
{if(i<0)return-i;else return i;});
```

在此例中,传递给transform的第四个参数是一个lambda,它的捕获列表是空的,接受单一int参数,返回一个int值.它的函数体是一个返回其参数的绝对值的if语句.

10.3.4参数绑定

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.

如果lambda的捕获列表为空,通常可以用函数来代替它.如前面章节所示,既可以用一个lambda,也可以用函数isshorter来实现将 vector中的单词按长度排序.类似的,对于打印vector内容的lambda,编写一个函数来替换它也是很容易的事情,这个函数只需接受一个string并在标准输出上打印它即可.

但是,对于捕获局部变量的lambda,用函数来替换它就不是那么容易了.例如,我们用在find_if调用中的lambda比较一个string和一个给定大小.我们可以很容易地编写一个完成同样工作的函数:

```
bool check size(const string&s,string:: size type sz)
{
return s.size()>=sz;
}
```

但是,我们不能用这个函数作为findif的一个参数.如前文所示,find if接受一个一元谓词,因此传递给findif的可调用对象必须接受单一参数.biggies传递给find_if的lambda使用捕获列表来保存sz.为了用checksize来代替此lambda,必须解决如何向sz形参传递一个参数的问题.

标准库bind函数

我们可以解决向checksize传递一个长度参数的问题,方法是使用一个新的名为bind的标准库函数,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(参见9.6节,第329页),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,arglist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newCalable时,newCallable会调用 callable,并传递给它arg list中的参数.

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newCalable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newCalable的第一个参数,2为第二个参数,依此类推.

绑定 check_size的sz参数

作为一个简单的例子,我们将使用bind生成一个调用check_size的对象,如下所示,它用一个定值作为其大小参数来调用check_size：

```
//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);
```

此bind调用只有一个占位符,表示check6只接受单一参数.占位符出现在arg_list的第一个位置,表示check6的此参数对应check_size的第一个参数.此参数是一个const strings.因此,调用check6必须传递给它一个string类型的参数,check6会将此参数传递给 check_size.

```
strings="hello";
boolb1=check6(s);//check6(s)会调用check_size(s,6)
```

使用bind,我们可以将原来基于lambda的find_if调用：

```
auto wc=find_if(words.begin(),words.end(),
[sz](const string sa)
```

替换为如下使用checksize的版本：

```
auto wc=find if(words.begin(),words.end(),bind(check_size,1,sz));
```

此bind 调用生成一个可调用对象,将check_size的第二个参数绑定到sz的值.当find_if对words中的string调用这个对象时,这些对象会调用check_size,将给定的string和sz传递给它.因此,find_if可以有效地对输入序列中每个string调用check_size,实现string的大小与sz的比较.

使用placeholders名字

名字n都定义在一个名为placeholders的命名空间中,而这个命名空间本身定义在std命名空间(参见3.1节,第74页)中.为了使用这些名字,两个命名空间都要写上.与我们的其他例子类似,对bind的调用代码假定之前已经恰当地使用了using声明.例如,1对应的using声明为：

```
using std:: placeholders:: _1;
```

此声明说明我们要使用的名字_1定义在命名空间placeholders中,而此命名空间又定义在命名空间std中.

对每个占位符名字,我们都必须提供一个单独的using声明.编写这样的声明很烦人,也很容易出错.可以使用另外一种不同形式的using语句(详细内容将在18.2.2节(第702页)中介绍),而不是分别声明每个占位符,如下所示:

```
using namespace namespace name;
```

这种形式说明希望所有来自namespace_name的名字都可以在我们的程序中直接使用.例如:

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

使得由placeholders定义的所有名字都可用.与bind函数一样,placeholders命名空间也定义在functiona1头文件中.

bind的参数

如前文所述,我们可以用bind修正参数的值.更一般的,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.例如,假定f是一个可调用对象,它有5个参数,则下面对bind的调用:

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

生成一个新的可调用对象,它有两个参数,分别用占位符_2和1表示.这个新的可调用对象将它自己的参数作为第三个和第五个参数传递给f.f的第一个、第二个和第四个参数分别被绑定到给定的值a、b和c上.

传递给g的参数按位置绑定到占位符.即,第一个参数绑定到1,第二个参数绑定到_2.因此,当我们调用g时,其第一个参数将被传递给f作为最后一个参数,第二个参数将被传递给f作为第三个参数.实际上,这个bind调用会将

`9(_1,_2)`

映射为

`f(a,b,_2,c,_1)`

即,对g的调用会调用f,用g的参数代替占位符,再加上绑定的参数a、b和c.例如,调用g(x,Y)会调用

`f(a,b,Y,c,X)`

用bind 重排参数顺序

下面是用bind重排参数顺序的一个具体例子,我们可以用bind颠倒isshroter的含义:

```
//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));
```

在第一个调用中,当sort需要比较两个元素A和B时,它会调用isshorter(A,B).在第二个对sort的调用中,传递给isshorter的参数被交换过来了.因此,当sort比较两个元素时,就好像调用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被拷贝到bind返回的可调用对象中.但是,与lambda类似,有时对有些绑定的参数我们希望以引用方式传递,或是要绑定参数的类型无法拷贝.

例如,为了替换一个引用方式捕获ostream的lambda:

```
//os是一个局部变量,引用一个输出流
//c是一个局部变量,类型为char
for_each(words.begin(),words.end(),
[&os,c](const string&s){os<<s<<c;});
```

可以很容易地编写一个函数,完成相同的工作:

```
ostream&print(ostream&os,const string&s,char c)
{
return os<<s<<c;
}
```

但是,不能直接用bind来代替对os的捕获:

```
//错误:不能拷贝os
for_each(words.begin(),words.end(),bind(print,os,_1,));
```

原因在于bind拷贝其参数,而我们不能拷贝一个ostream.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(words.begin(),words.end(),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.标准库中还有一个cref函数,生成一个保存const引用的类.与bind一样,函数ref和cref也定义在头文件functional中.

向后兼容:参数绑定:

旧版本C++提供的绑定函数参数的语言特性限制更多,也更复杂.标准库定义了两个分别名为bind1st和bind2nd的函数.类似bind,这两个函数接受一个函数作为参数,生成一个新的可调用对象,该对象调用给定函数,并将绑定的参数传递给它.但是,这些函数分别只能绑定第一个或第二个参数.由于这些函数局限太强,在新标准中已被弃用(deprecated).所谓被弃用的特性就是在新版本中不再支持的特性.新的C++程序应该使用bind.

10.4再探迭代器

除了为每个容器定义的迭代器之外,标准库在头文件iterator中还定义了额外几种迭代器.这些迭代器包括以下几种.

- 插入迭代器(insert iterator)：这些迭代器被绑定到一个容器上,可用来向容器插入元素.
- 流迭代器(stream iterator)：这些迭代器被绑定到输入或输出流上,可用来遍历所关联的IO流.
- 反向迭代器(reverse iterator)：这些迭代器向后而不是向前移动.除了forward_list之外的标准库容器都有反向迭代器.
- 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.我们将在13.6.2节(第480页)介绍移动迭代器.

10.4.1插入迭代器

插入器是一种迭代器适配器(参见9.6节,第329页),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.当我们通过一个插入迭代器进行赋值时,该迭代器调用容器操作来向给定容器的指定位置插入一个元素.表10.2列出了这种迭代器支持的操作.

表10.2：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

插入器有三种类型,差异在于元素插入的位置：

- back inserter(参见10.2.2节,第341页)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.元素将被插入到给定迭代器所表示的元素之前.

> **Note**
> 只有在容器支持push_front的情况下,我们才可以使用front inserter.类似的,只有在容器支持push_back的情况下,我们才能使用back_inserter.

理解插入器的工作过程是很重要的：当调用inserter(c,iter)时,我们得到一个迭代器,接下来使用它时,会将元素插入到iter原来所指向的元素之前的位置.即,如果it是由inserter生成的迭代器,则下面这样的赋值语句

```
*it=val;
```

其效果与下面代码一样

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

front inserter生成的迭代器的行为与inserter生成的迭代器完全不一样.当我们使用front_inserter时,元素总是插入到容器第一个元素之前.即使我们传递给inserter的位置原来指向第一个元素,只要我们在此元素之前插入一个新元素,此元素就不再是容器的首元素了:

```
list<int>lst={1,2,3,4};list<int>lst2,lst3;//空list
//拷贝完成之后,lst2包含4321
copy(lst.cbegin(),lst.cend(),front_inserter(lst2));
//拷贝完成之后,lst3包含1234
copy(lst.cbegin(),lst.cend(),inserter(lst3,lst3.begin()));
```

当调用front inserter(c)时,我们得到一个插入迭代器,接下来会调用push front.当每个元素被插入到容器c中时,它变为c的新的首元素.因此,front inserter生成的迭代器会将插入的元素序列的顺序颠倒过来,而inserter和back inserter则不会.

10.4.2 iostream迭代器

虽然iostream类型不是容器,但标准库定义了可以用于这些I0类型对象的迭代器(参见8.1节,第278页).istream_iterator(参见表10.3)读取输入流,ostream_iterator(参见表10.4节,第361页)向一个输出流写数据.这些迭代器将它们对应的流当作一个特定类型的元素序列来处理.通过使用流迭代器,我们可以用泛型算法从流对象读取数据以及向其写入数据.

istream_iterator操作

当创建一个流迭代器时,必须指定迭代器将要读写的对象类型.一个istream_iterator 使用>>来读取流.因此,istream_iterator 要读取的类型必须定义了输入运算符.当创建一个istream_iterator时,我们可以将它绑定到一个流.当然,我们还可以默认初始化迭代器,这样就创建了一个可以当作尾后值使用的迭代器.

```
istream_iterator<int>int_it(cin);//从cin读取int
istream_iterator<int>int_eof;//尾后迭代器
ifstream in("afile");
istream_iterator<string>str it(in);//从”afile“读取字符串
```

下面是一个用istream_iterator从标准输入读取数据,存入一个vector的例子:

```
istream_iterator<int>in iter(cin);//从cin 读取int
istream_iterator<int>eof;//istream尾后迭代器
while(in iter! =eof)//当有数据可供读取时
//后置递增运算读取流,返回迭代器的旧值
//解引用迭代器,获得从流读取的前一个值
vec.push_back(*in_iter++);
```

此循环从cin读取int值,保存在vec中.在每个循环步中,循环体代码检查in_iter是否等于eof.eof被定义为空的istreamiterator,从而可以当作尾后迭代器来使用.对于一个绑定到流的迭代器,一旦其关联的流遇到文件尾或遇到10错误,迭代器的值就与尾后迭代器相等.

此程序最困难的部分是传递给 push back的参数,其中用到了解引用运算符和后置递增运算符.该表达式的计算过程与我们之前写过的其他结合解引用和后置递增运算的表达式一样(参见4.5节,第131页).后置递增运算会从流中读取下一个值,向前推进,但返回的是迭代器的旧值.迭代器的旧值包含了从流中读取的前一个值,对迭代器进行解引用就能获得此值.

我们可以将程序重写为如下形式,这体现了istreamiterator更有用的地方:

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

本例中我们用一对表示元素范围的迭代器来构造vec.这两个迭代器是istreamiterator,这意味着元素范围是通过从关联的流中读取数据获得的.这个构造函数从cin中读取数据,直至遇到文件尾或者遇到一个不是int的数据为止.从流中读取的数据被用来构造vec.

表10.3：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

使用算法操作流迭代器

由于算法使用迭代器操作来处理数据,而流迭代器又至少支持某些迭代器操作,因此我们至少可以用某些算法来操作流迭代器.我们在10.5.1节(第365页)会看到如何分辨哪些算法可以用于流迭代器.下面是一个例子,我们可以用一对istream_iterator来调用accumulate:

```
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:

23 109 45 89 6 34 12 90 34 23 56 23 8 89 23

则输出为664.

istream_iterator允许使用懒惰求值

当我们将一个istream_iterator绑定到一个流时,标准库并不保证迭代器立即从流读取数据.具体实现可以推迟从流中读取数据,直到我们使用迭代器时才真正读取.标准库中的实现所保证的是,在我们第一次解引用迭代器之前,从流中读取数据的操作已经完成了.对于大多数程序来说,立即读取还是推迟读取没什么差别.但是,如果我们创建了一个istream_iterator,没有使用就销毁了,或者我们正在从两个不同的对象同步读取同一个流,那么何时读取可能就很重要了.

ostream iterator操作

我们可以对任何具有输出运算符(<<运算符)的类型定义ostreamiterator.当创建一个ostream iterator时,我们可以提供(可选的)第二参数,它是一个字符串,在输出每个元素后都会打印此字符串.此字符串必须是一个C风格字符串(即,一个字符串字面常量或者一个指向以空字符结尾的字符数组的指针).必须将ostreamiterator绑定到一个指定的流,不允许空的或表示尾后位置的ostream_iterator.

表10.4: 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va1写入到out 所绑定的ostream中.va1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

我们可以用ostream iterator来输出值的序列:

```
ostream iterator<int>out iter(cout,"");
for(autoe:vec)
*out iter++=e;//赋值语句实际上将元素写到cout
cout<<endl;
```

此程序将vec中的每个元素写到cout,每个元素后加一个空格.每次向out iter赋值时,写操作就会被提交.

值得注意的是,当我们向outiter赋值时,可以忽略解引用和递增运算.即,循环可以重写成下面的样子:

```
for(auto e:vec)
out_iter=e;//赋值语句将元素写到cout
cout<<end1;
```

运算符*和++实际上对ostream_iterator对象不做任何事情,因此忽略它们对我们的程序没有任何影响.但是,推荐第一种形式.在这种写法中,流迭代器的使用与其他迭代器的使用保持一致.如果想将此循环改为操作其他迭代器类型,修改起来非常容易.而且,对于读者来说,此循环的行为也更为清晰.

可以通过调用copy来打印vec中的元素,这比编写循环更为简单:

```
copy(vec.begin(),vec.end(),out_iter);
cout<<end1;
```

使用流迭代器处理类类型

我们可以为任何定义了输入运算符(>>)的类型创建istream_iterator对象.类似的,只要类型有输出运算符(<<),我们就可以为其定义ostream_iterator.由于Salesitem既有输入运算符也有输出运算符,因此可以使用IO迭代器重写1.6节(第21页)中的书店程序:

```
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itemiter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;//输出sum当前值
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
```

此程序使用item_iter从cin 读取sales_item交易记录,并将和写入cout,每个结果后面都跟一个换行符.定义了自己的迭代器后,我们就可以用item_iter读取第一条交易记录,用它的值来初始化sum:

```
//将第一条交易记录保存在sum中,并读取下一条记录
Sales item sum=*item iter++;
```

此处,我们对itemiter执行后置递增操作,对结果进行解引用操作.这个表达式读取下一条交易记录,并用之前保存在item_iter中的值来初始化sum.

while循环会反复执行,直至在cin上遇到文件尾为止.在while循环体中,我们检查sum与刚刚读入的记录是否对应同一本书.如果两者的ISBN不同,我们将sum赋予out iter,这将会打印sum的当前值,并接着打印一个换行符.在打印了前一本书的交易金额之和后,我们将最近读入的交易记录的副本赋予sum,并递增迭代器,这将读取下一条交易记录.循环会这样持续下去,直至遇到错误或文件尾.在退出之前,记住要打印输入中最后一本书的交易金额之和.

10.4.3反向迭代器

反向迭代器就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,递增(以及递减)操作的含义会颠倒过来.递增一个反向迭代器(++it)会移动到前一个元素;递减一个迭代器(—it)会移动到下一个元素.

除了forward list之外,其他容器都支持反向迭代器.我们可以通过调用rbegin、rend、crbegin和crend成员函数来获得反向迭代器.这些成员函数返回指向容器尾元素和首元素之前一个位置的迭代器.与普通迭代器一样,反向迭代器也有const和非const版本.

图10.1显示了一个名为vec的假设的vector上的4种迭代器:

vec.cbegin()	vec.cend()
vec.crend()	vec.crbegin()

图10.1：比较cbegin/cend和crbegin/crend

下面的循环是一个使用反向迭代器的例子,它按逆序打印vec中的元素:

```
vector<int>vec={0,1,2,3,4,5,6,7,8,9};
//从尾元素到首元素的反向迭代器
for(auto r_iter=vec.crbegin();//将riter绑定到尾元素
r_iter! =vec.crend();//crend指向首元素之前的位置
++r_iter)//实际是递减,移动到前一个元素
cout<<*r_iter<<endl;//打印9,8,7,..0
```

虽然颠倒递增和递减运算符的含义可能看起来令人混淆,但这样做使我们可以用算法透明地向前或向后处理容器.例如,可以通过向sort传递一对反向迭代器来将vector整理为递减序:

```
sort(vec.begin(),vec.end());//按“正常序”排序vec
//按逆序排序：将最小元素放在vec的末尾
sort(vec.rbegin(),vec.rend());
```

反向迭代器需要递减运算符

不必惊讶,我们只能从既支持++也支持一的迭代器来定义反向迭代器.毕竟反向迭代器的目的是在序列中反向移动.除了forward_list之外,标准容器上的其他迭代器都既支持递增运算又支持递减运算.但是,流迭代器不支持递减运算,因为不可能在一个流中反向移动.因此,不可能从一个forward list或一个流迭代器创建反向迭代器.

反向迭代器和其他迭代器间的关系

假定有一个名为1ine的string,保存着一个逗号分隔的单词列表,我们希望打印line中的第一个单词.使用find可以很容易地完成这一任务:

```
//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;
```

如果line中有逗号,那么comma将指向这个逗号;否则,它将等于line.cend().当我们打印从line.cbegin()到comma之间的内容时,将打印到逗号为止的字符,或者打印整个string(如果其中不含逗号的话).

如果希望打印最后一个单词,可以改用反向迭代器:

```
//在一个逗号分隔的列表中查找最后一个元素
auto rcomma = find(line.crbegin(),line.crend(),',');
```

由于我们将crbegin()和crend()传递给find,find将从line的最后一个字符开始向前搜索.当find完成后,如果line中有逗号,则rcomma指向最后一个逗号一即,它指向反向搜索中找到的第一个逗号.如果line中没有逗号,则rcomma指向line.crend().

当我们试图打印找到的单词时,最有意思的部分就来了.看起来下面的代码是显然的方法

```
//错误：将逆序输出单词的字符
cout << string(line.crbegin(),rcomma)<< endl;
```

但它会生成错误的输出结果.例如,如果我们的输入是

```
FIRST,MIDDLE,LAST
```

则这条语句会打印TSAL!

图10.2说明了问题所在：我们使用的是反向迭代器,会反向处理string.因此,上述输出语句从crbegin开始反向打印line中内容.而我们希望按正常顺序打印从rcomma开始到line末尾间的字符.但是,我们不能直接使用rcomma.因为它是一个反向迭代器,意味着它会反向朝着string的开始位置移动.需要做的是,将rcomma转换回一个普通迭代器,能在line中正向移动.我们通过调用reverse_iterator的base成员函数来完成这一转换,此成员函数会返回其对应的普通迭代器：

```
//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;
```

给定和之前一样的输入,这条语句会如我们的预期打印出LAST.

cbegin()comma rcomma.base()cend()
FI RST,MIDD LE,LA S T
rcomma crbegin()

图10.2：反向迭代器和普通迭代器间的关系

图10.2中的对象显示了普通迭代器与反向迭代器之间的关系.例如,rcomma和rcomma.base()指向不同的元素,line.crbegin 和line.cend()也是如此.这些不同保证了元素范围无论是正向处理还是反向处理都是相同的.

从技术上讲,普通迭代器与反向迭代器的关系反映了左闭合区间(参见9.2.1节,第296 页)的特性.关键点在于[line.crbegin(),rcomma)和[rcomma.base(),line.cend())指向line中相同的元素范围.为了实现这一点,rcomma和rcomma.base()必须生成相邻位置而不是相同位置,crbegin()和cend()也是如此.

> **Note**
> 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Note的结果：当我们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,结果迭代器与原迭代器指向的并不是相同的元素.

10.5泛型算法结构

任何算法的最基本的特性是它要求其迭代器提供哪些操作.某些算法,如find,只要求通过迭代器访问元素、递增迭代器以及比较两个迭代器是否相等这些能力.其他一些算法,如sort,还要求读、写和随机访问元素的能力.算法所要求的迭代器操作可以分为5个迭代器类别(iterator category),如表10.5所示.每个算法都会对它的每个迭代器参数指明须提供哪类迭代器.

表10.5：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

第二种算法分类的方式(如我们在本章开始所做的)是按照是否读、写或是重排序列中的元素来分类.附录A按这种分类方法列出了所有算法.

算法还共享一组参数传递规范和一组命名规范,我们在介绍迭代器类别之后将介绍这些内容.

10.5.1 5类迭代器

类似容器,迭代器也定义了一组公共操作.一些操作所有迭代器都支持,另外一些只有特定类别的迭代器才支持.例如,ostream_iterator只支持递增、解引用和赋值.vector、string和deque的迭代器除了这些操作外,还支持递减、关系和算术运算.

迭代器是按它们所提供的操作来分类的,而这种分类形成了一种层次.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.

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.例如,find算法在一个序列上进行一遍扫描,对元素进行只读操作,因此至少需要输入迭代器.replace函数需要一对迭代器,至少是前向迭代器.类似的,replace_copy的前两个迭代器参数也要求至少是前向迭代器.其第三个迭代器表示目的位置,必须至少是输出迭代器.其他的例子类似.对每个迭代器参数来说,其能力必须与规定的最小类别至少相当.向算法传递一个能力更差的迭代器会产生错误.

> **WARNING**
> 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WARNING或提示.

迭代器类别

输入迭代器(input iterator)：可以读取序列中的元素.一个输入迭代器必须支持

- 用于比较两个迭代器的相等和不相等运算符(==、！=)
- 用于推进迭代器的前置和后置递增运算(++)
- 用于读取元素的解引用运算符(*);解引用只会出现在赋值运算符的右侧
- 箭头运算符(->),等价于(*it).member,即,解引用迭代器,并提取对象的成员

输入迭代器只用于顺序访问.对于一个输入迭代器,*it++保证是有效的,但递增它可能导致所有其他指向流的迭代器失效.其结果就是,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.算法find和accumulate要求输入迭代器;而istream_iterator是一种输入迭代器.

输出迭代器(output iterator)：可以看作输入迭代器功能上的补集-一只写而不读元素.输出迭代器必须支持

- 用于推进迭代器的前置和后置递增运算(++)
- 解引用运算符(*),只出现在赋值运算符的左侧(向一个已经解引用的输出迭代器赋值,就是将值写入它所指向的元素)

我们只能向一个输出迭代器赋值一次.类似输入迭代器,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.例如,copy函数的第三个参数就是输出迭代器.ostream_iterator类型也是输出迭代器.

前向迭代器(forward iterator)：可以读写元素.这类迭代器只能在序列中沿一个方向移动.前向迭代器支持所有输入和输出迭代器的操作,而且可以多次读写同一个元素.因此,我们可以保存前向迭代器的状态,使用前向迭代器的算法可以对序列进行多遍扫描.算法replace要求前向迭代器,forward_list上的迭代器是前向迭代器.

双向迭代器(bidirectional iterator)：可以正向/反向读写序列中的元素.除了支持所有前向迭代器的操作之外,双向迭代器还支持前置和后置递减运算符(--).算法reverse要求双向迭代器,除了forward_list之外,其他标准库都提供符合双向迭代器要求的迭代器

随机访问迭代器(random-access iterator)：提供在常量时间内访问序列中任意元素的能力.此类迭代器支持双向迭代器的所有功能,此外还支持表3.7(第99页)中的操作：

- 用于比较两个迭代器相对位置的关系运算符(<、<=、>和>=)
- 迭代器和一个整数值的加减运算(+、+=、-和-=),计算结果是迭代器在序列中前进(或后退)给定整数个元素后的位置
- 用于两个迭代器上的减法运算符(-),得到两个迭代器的距离
- 下标运算符(iter[n]),与*(iter[n])等价

算法sort要求随机访问迭代器.array、deque、string和vector的迭代器都是随机访问迭代器,用于访问内置数组元素的指针也是.

10.5.2 算法形参模式

在任何其他算法分类之上,还有一组参数规范.理解这些参数规范对学习新算法很有帮助--通过理解参数的含义,你可以将注意力集中在算法所做的操作上.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);
```

其中alg是算法的名字,beg和end表示算法所操作的输入范围.几乎所有算法都接受个输入范围,是否有其他参数依赖于要执行的操作.这里列出了常见的一种--dest、beg2 和end2,都是迭代器参数.顾名思义,如果用到了这些迭代器参数,它们分别承担指定目的位置和第二个范围的角色.除了这些迭代器参数,一些算法还接受额外的、非迭代器的特定参数.

接受单个目标迭代器的算法

dest 参数是一个表示算法可以写入的目的位置的迭代器.算法假定(assume)：按其需要写入数据,不管写入多少个元素都是安全的.

> **WARNING**
> 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.

如果dest是一个直接指向容器的迭代器,那么算法将输出数据写到容器中已存在的元素内.更常见的情况是,dest被绑定到一个插入迭代器(参见10.4.1节,第358页)或是一个ostream_iterator(参见10.4.2节,第359页).插入迭代器会将新元素添加到容器中,因而保证空间是足够的.ostream_iterator会将数据写入到一个输出流,同样不管要写入多少个元素都没有问题

接受第二个输入序列的算法

接受单独的beg2或是接受beg2和end2的算法用这些迭代器表示第二个输入范围.这些算法通常使用第二个范围中的元素与第一个输入范围结合来进行一些运算.

如果一个算法接受beg2和end2,这两个迭代器表示第二个范围.这类算法接受两个完整指定的范围：[beg,end)表示的范围和[beg2 end2)表示的第二个范围.

只接受单独的beg2(不接受end2)的算法将beg2作为第二个输入范围中的首元素.此范围的结束位置未指定,这些算法假定从beg2开始的范围与beg和end所表示的范围至少一样大.

> **WARNING**
> 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

除了参数规范,算法还遵循一套命名和重载规范.这些规范处理诸如：如何提供一个操作代替默认的<或=运算符以及算法是将输出数据写入输入序列还是一个分离的目的位置等问题.

一些算法使用重载形式传递一个谓词

接受谓词参数来代替<或=运算符的算法,以及那些不接受额外参数的算法,通常都是重载的函数.函数的一个版本用元素类型的运算符来比较元素;另一个版本接受一个额外谓词参数,来代替<或=:

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

两个调用都重新整理给定序列,将相邻的重复元素删除.第一个调用使用元素类型的==运算符来检查重复元素;第二个则调用comp来确定两个元素是否相等.由于两个版本的函数在参数个数上不相等,因此具体应该调用哪个版本不会产生歧义(参见6.4节,第208页).

_if版本的算法

接受一个元素值的算法通常有另一个不同名的(不是重载的)版本,该版本接受一个谓词(参见10.3.1节,第344页)代替元素值.接受谓词参数的算法都有附加的if前缀:

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

这两个算法都在输入范围中查找特定元素第一次出现的位置.算法find查找一个指定值;算法find_if查找使得pred返回非零值的元素.

这两个算法提供了命名上差异的版本,而非重载版本,因为两个版本的算法都接受相同数目的参数.因此可能产生重载歧义,虽然很罕见,但为了避免任何可能的歧义,标准库选择提供不同名字的版本而不是重载.

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回给定的输入序列中.这些算法还提供另一个版本,将元素写到一个指定的输出目的位置.如我们所见,写到额外目的空间的算法都在名字后面附加一个_copy(参见10.2.2节,第341页):

```
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
```

些算法同时提供_copy和_if版本.这些版本接受一个目的位置迭代器和一个谓词:

```
//从v1中删除奇数元素
remove_if(v1.begin(),v1.end(),
[](int i)(return i&2;1);
//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;});
```

两个算法都调用了lambda(参见10.3.2节,第346页)来确定元素是否为奇数.在第一个调用中,我们从输入序列中将奇数元素删除.在第二个调用中,我们将非奇数(亦即偶数)元素从输入范围拷贝到v2中.

10.6 特定容器算法

与其他容器不同,链表类型list和forward_list定义了几个成员函数形式的算法,如表10.6所示.特别是,它们定义了独有的sort,merge,remove,reverse和unique.通用版本的sort要求随机访问迭代器,因此不能用于list和forward_list,因为这两个类型分别提供双向迭代器和前向迭代器.

链表类型定义的其他算法的通用版本可以用于链表,但代价太高.这些算法需要交换输入序列中的元素.一个链表可以通过改变元素间的链接而不是真的交换它们的值来快速"交换"元素.因此,这些链表版本的算法的性能比对应的通用版本好得多.

> **Best Practices**
> 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.

表10.6：list和forward_list成员函数版本的算法

这些操作都返回void	
lst.merge(lst2)	将来自lst2的元素合并入lst.lst和lst2都必须是有序的.
lst.merge(lst2,comp)	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

splice成员

链表类型还定义了splice算法,其描述见表10.7.此算法是链表数据结构所特有的,因此不需要通用版本.

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flstolst2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容器

多数链表特有的算法都与其通用版本很相似,但不完全相同.链表特有版本与通用版本间的一个至关重要的区别是链表版本会改变底层的容器.例如,remove的链表版本会删除指定的元素.unique的链表版本会删除第二个和后继的重复元素.

类似的,merge和splice会销毁其参数.例如,通用版本的merge将合并的序列写到一个给定的目的迭代器;两个输入序列是不变的.而链表版本的merge函数会销毁给定的链表-一元素从参数指定的链表中删除,被合并到调用merge的链表对象中.在merge之后,来自两个链表中的元素仍然存在,但它们都已在同一个链表中.

小结
标准库定义了大约100个类型无关的对序列进行操作的算法.序列可以是标准库容器类型中的元素、一个内置数组或者是(例如)通过读写一个流来生成的.算法通过在迭代器上进行操作来实现类型无关.多数算法接受的前两个参数是一对迭代器,表示一个元素范围.额外的迭代器参数可能包括一个表示目的位置的输出迭代器,或是表示第二个输入范围的另一个或另一对迭代器.

根据支持的操作不同,迭代器可分为五类：输入、输出、前向、双向以及随机访问迭代器.如果一个迭代器支持某个迭代器类别所要求的操作,则属于该类别.

如同迭代器根据操作分类一样,传递给算法的迭代器参数也按照所要求的操作进行分类.仅读取序列的算法只要求输入迭代器操作.写入数据到目的位置迭代器的算法只要求输出迭代器操作,依此类推.

算法从不直接改变它们所操作的序列的大小.它们会将元素从一个位置拷贝到另一个位置,但不会直接添加或删除元素.

虽然算法不能向序列添加元素,但插入迭代器可以做到.一个插入迭代器被绑定到个容器上.当我们将一个容器元素类型的值赋予一个插入迭代器时,迭代器会将该值添加到容器中.

容器forward_list和list对一些通用算法定义了自己特有的版本.与通用算法不同,这些链表特有版本会修改给定的链表.

术语表

- back_inserter 这是一个迭代器适配器,它接受一个指向容器的引用,生成一个插入迭代器,该插入迭代器用push_back向指定容器添加元素.
- 双向迭代器(bidirectional iterator) 支持前向迭代器的所有操作,还具有用--在序列中反向移动的能力.
- 二元谓词(binary predicate)接受两个参数的谓词.
- bind 标准库函数,将一个或多个参数绑定到一个可调用表达式.bind定义在头文件functional 中.
- 可调用对象(callable object) 可以出现在调用运算符左边的对象.函数指针、lambda以及重载了函数调用运算符的类的对象都是可调用对象.
- 捕获列表(capture list) lambda表达式的一部分,指出lambda表达式可以访问所在上下文中哪些变量.
- cref 标准库函数,返回一个可拷贝的对象,其中保存了一个指向不可拷贝类型的const对象的引用.
- 前向迭代器(forward iterator) 可以读写元素,但不必支持--的迭代器.
- front_inserter迭代器适配器 ,给定一个容器,生成一个用push_front向容器开始位置添加元素的插入迭代器.
- 泛型算法(generic algorithm) 类型无关的算法.
- 输入迭代器(input iterator)可以读但不能写序列中元素的迭代器.
- 插入迭代器(insert iterator) 迭代器适配器,生成一个迭代器,该迭代器使用容器操作向给定容器添加元素.
- 插入器(inserter) 迭代器适配器,接受个迭代器和一个指向容器的引用,生成个插入迭代器,该插入迭代器用insert在给定迭代器指向的元素之前的位置添加元素
- istream_iterator 读取输入流的流选代
- 迭代器类别(iterator category) 根据所支持的操作对迭代器进行的分类组织.迭代器类别形成一个层次,其中更强大的类别支持更弱类别的所有操作.算法使用迭代器类别来指出迭代器参数必须支持哪些操作.只要迭代器达到所要求的最小类别,它就可以用于算法.例如,一些算法只要求输入迭代器.这类算法可处理除只满足输出迭代器要求的迭代器之外的任何迭代器.而要求随机访问迭代器的算法只能用于支持随机访问操作的迭代器
- lambda 表达式 (lambda expression)可调用的代码单元.一个lambda类似一个未命名的内联函数.一个lambda以一个捕获列表开始,此列表允许lambda访问所在函数中的变量.类似函数,lambda有一个(可能为空的)参数列表、一个返回类型和个函数体.lambda可以忽略返回类型.如果函数体是一个单一的return语句,返回类型就从返回对象的类型推断.否则,忽略的返回类型默认定为void.
- 移动迭代器(move iterator)迭代器适配器,生成一个迭代器,该迭代器移动而不是拷贝元素.移动迭代器将在第13章中进行介绍.
  ostream_iterator写输出流的迭代器.
- 输出迭代器(output iterator)可以写元素,但不必具有读元素能力的迭代器.
- 谓词(predicate) 返回可以转换为bool类型的值的函数.泛型算法通常用来检测元素.标准库使用的谓词是一元(接受个参数)或二元(接受两个参数)的.

- 随机访问迭代器(random-access iterator) 支持双向迭代器的所有操作再加上比较迭代器值的关系运算符、下标运算符和迭代器上的算术运算,因此支持随机访问元素.
- ref 标准库函数,从一个指向不能拷贝的类型的对象的引用生成一个可拷贝的对象.
- 反向迭代器(reverse iterator) 在序列中反向移动的迭代器.这些迭代器交换了++和一的含义.
- 流迭代器(stream iterator) 可以绑定到一个流的迭代器.
- 一元谓词(unary predicate) 接受一个参数的谓词.