1 顺序容器概述

1. 顺序容器类型

vector	可变大小数组。支持快速随机访问。
deque	双端队列。支持快速随机访问。
list	双向链表。只支持双向顺序访问。额外内存开销大。
forward_list	单向链表。只支持单向顺序访问。额外内存开销大。
array	固定大小数组。支持快速随机访问。
string	与vector相似的容器，但专门用于保存字符。随机访问快

2 容器操作

1. 迭代器

- 标准容器迭代器的运算符

*iter	返回迭代器iter所指元素的引用
++iter	令iter指示容器中的下一个元素
--iter	令iter指示容器中的上一个元素 •forward_list迭代器不支持递减运算符
iter1==iter2 iter1!=iter2	判断两个迭代器是否相等，如果两个迭代器指示同一个元素或者它们是同一个容器的尾后迭代器，则相等，否则不相等

- 仅vector，string，deque，array迭代器支持的运算

iter+n, iter-n	
iter+=n, iter-=n	
iter1-iter2	两个迭代器相减的结果是它们之间的距离，返回类型是difference_type的有符号整数，即这个距离可正可负 •两个迭代器不能相加
<, <=, >, >=	

- 获取迭代器

获取迭代器	
c.begin(), c.end()	返回指向c的首元素和尾元素之后位置的迭代器iterator •如果容器为空，则它们返回同一个迭代器，都是尾后迭代器 •如果对一个const对象调用c.begin()或c.end()，会返回const_iterator
c.cbegin(), c.cend()	返回const_iterator
反向容器的额外成员	**•forward_list不支持**
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.rend()	返回指向c的尾元素和首元素之前位置的迭代器reverse_iterator
c.crbegin(), c.crend()	返回const_reverse_iterator

2. 容器类型成员

- 用法，显示使用其类名，如vector<int>::difference_type count;

iterator	此容器类型的迭代器类型
const_iterator	可以读取元素，但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的大小
difference_type	有符号整数类型，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&

3. 容器定义和初始化

C c;	默认构造函数，构造空容器 •如果C是array，则c中元素按默认方式初始化
C c1(c2); C c1=c2;	c1初始化为c2的拷贝，c1与c2必须是相同容器类型且保存相同元素类型 •如果C是array，两者还必须具有相同大小

C c(b,e);	将迭代器b和e指定的范围内的元素拷贝到c进行初始化，范围中元素类型必须与C的元素类型相容(此时并不要求容器类型是相同的，且元素类型只要可以转换即可)
	•array不支持
	`list<string> authors = { "Milton","Shakespeare","Austen" };`
	`list<string> list2(authors); //正确，类型匹配`
	`vector<string> v(authors); //错误，容器类型不匹配`
	`vector<const char*> articles = { "a","an","the" };`
	`//正确，可以将const char* 元素转换为string`
	`forward_list<string> words(articles.begin(), articles.end());`

C c{a,b,...}; C c={a,b,...};	c初始化为初始化列表中元素的拷贝 •如果C是array，元素数目必须小于等于其大小，任何遗漏元素进行值初始化 •对于除array之外的容器类型，初始化列表还隐含指定了容器的大小
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，如果没有默认构造函数，则必须采用下面一种初始化方法指定一个显示的元素初始值 •只有顺序容器(不包括array)的构造函数才能接受大小参数 •string不适用
C seq(n,t);	seq包含n个初始化为值t的元素 •只有顺序容器(不包括array)的构造函数才能接受大小参数

- 标准库array具有固定大小，定义array时必须指定容器大小，如array<int,42> a;
  内置数组类型无法进行拷贝和对象赋值操作，但array可以

```
int a[2] = { 0,1 };
int aa[2] = a; //错误
array<int, 2> b = { 0,1 };
array<int, 2> bb = b; //正确
```

4. 赋值和swap

c1=c2;	将c1中的元素替换为c2中元素的拷贝，c1和c2必须具有相同的类型

c1={a,b,...};	将c1中的元素替换为列表中元素的拷贝 •array不支持
a.swap(b); swap(a,b);	交换a和b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，除array外，swap不会对任何元素进行拷贝、删除或插入，swap只是交换了两个容器的内部数据结构。而对于array的swap则会真正交换他们的元素。 •一般建议用非成员函数版本的swap
以下assign操作不适用于关联容器和array	
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

- 赋值相关运算会导致指向左边容器内部的迭代器、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迭代器、引用和指针失效(array和string除外)。

5. 容器大小操作

c.size()	c中元素的数目 •forward_list不支持
c.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
c.empty()	若c中存储了元素，返回false，否则返回true

6. 容器的关系运算符

==, !=	所有容器都支持相等(不等)运算符，两边的运算对象必须是相同类型的容器，并且保存相同类型的元素。比较两个容器实际上是进行元素的逐个比较。 •容器的判等运算符实际上是使用元素的==运算符实现比较的，而其他关系运算符则是使用元素的<运算符实现比较的，所以如果要使用容器的判等或其他关系运算符，必须保证元素类型定义了所需运算符
<, <=, >, >=	•无序关联容器不支持

3 顺序容器操作

1. 向顺序容器添加元素

- array不支持以下所有操作
- forward_list有自己专有版本的insert和emplace
- forward_list不支持push_back和emplace_back
- vector不支持push_front和emplace_front；string不支持push_front和所有emplace操作
- 向一个vector、string或deque插入元素会使所有指向容器的迭代器、引用和指针失效

c.push_back(t);
c.emplace_back(args);
	在c的尾部创建一个值为t或由args创建的元素，返回void
	• push、insert均是拷贝元素到容器中，而emplace是构造元素，调用emplace时将参数传递给元素类型的构造函数，然后在内存空间中直接构造元素，所以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
	`c.emplace_back(); //使用默认构造函数`
	`c.emplace(iter, "9"); //使用带一个string参数的构造函数`
	`//使用以string, int, double为参数的构造函数`
	`c.emplace_front("987", 25, 15.99);`

c.push_front(t); c.emplace_front(args);	在c的头部创建一个值为t或由args创建的元素，返回void
c.insert(p,t); c.emplace(p,args);	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.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
c.insert(p,il);	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

2. 在顺序容器中访问元素

- at和下标操作只适用于string、vector、deque和array
- back不适用于forward_list
- 不能对空容器访问元素，所以在访问元素前一个良好的习惯是判断容器是否非空

c.back();	返回c中尾元素的引用
c.front();	返回c中首元素的引用

c[n]	返回c中下标为n的元素的引用
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

3. 顺序容器的删除操作

- array不支持以下所有操作
- forward_list有自己专有版本的erase
- forward_list不支持pop_back
- vector和string不支持pop_front
- 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和string中删除点之后位置的迭代器、引用和指针都会失效。

c.pop_back();	删除c中尾元素，返回void
c.pop_front();	删除c中首元素，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，则返回尾后迭代器
c.erase(b,e);	删除迭代器b和e所指定范围内的元素，返回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

4. 特殊的forward_list操作

lst.before_begin(); lst.cbefore_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.insert_after(p,t);
lst.insert_after(p,n,t);
lst.insert_after(p,b,e);
lst.insert_after(p,il);
	在迭代器p之后的位置插入元素。返回指向最后一个插入元素的迭代器，如果范围为空则返回p。p不能是尾后迭代器。

lst.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。p不能是尾后迭代器。
lst.erase_after(p); lst.erase_after(b,e);	删除p指向的位置之后的元素，或删除从b之后的元素直到(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。p不能是lst的尾元素或者尾后迭代器。

5. 顺序容器的大小操作

- array不支持resize操作
- 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。

c.resize(n);	调整c的大小为n个元素。若n<c.size()，则容器后部多出的元素会被删除；若

	n>c.size()，则会将新元素(值初始化)添加到容器后部(forward_list也是添加新元素至后部)
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

4 vector的其他操作

1. 管理容量的成员函数

c.shrink_to_fit();	请将capacity()减少为与size()相同大小 • 只适用于vector、string和deque • 调用该函数只是一个请求，标准库并不保证一定会退还内存
c.capacity();	c的容量，即不重新分配内存空间的话，c可以保存多少元素 • 只适用于vector和string
c.reserve(n);	分配至少能容纳n个元素的内存空间，它并不改变容器中元素的数量 • 只适用于vector和string • 只有当需要的内存空间超过当前容量时，调用reserve才会改变vector的容量，且至少分配与需求一样大甚至更大的内存空间；如果需求大小小于等于当前容量，reserve什么也不做

5 string的其他操作

1. 构造string的其他方法

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 •也可以不带参数n，则会拷贝cp数组中的字符直到遇到空字符
string s(s2,pos2);	s是string s2从下标pos2开始的字符的拷贝
string s(s2,pos2,len2);	s是string s2从下标pos2开始len2个字符的拷贝，不管len2值是多少，构造函数至多拷贝s2.size()-pos2个字符
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，至多拷贝到s的末尾。pos的默认值是0，n的默认值是s.size()-pos，即拷贝从pos开始的所有字符。

```
const char* cp = "Hello World!"; //以空字符结束的数组
char noNull[] = { 'H','i','!' }; //不是以空字符结束
string str1(cp); //str1=="Hello World!"
string str2(noNull, 2); //str2=="Hi"
string str3(noNull); //未定义，因为noNull不是以空字符结束
string str4(cp + 6, 5); //str4=="World"
string str5(str1, 6, 5); //str5=="World"
string str6(str1, 6); //str6=="World!"
string str7(str1, 6, 20); //str7=="World!"
string str8(str1, 16); //抛出一个out_of_range异常
//以下初始化方式同顺序容器的初始化
string s1;
string s2(str1); //等同于string s2=str1;
string s3(noNull, noNull + 3); //s3=="Hi!"
string s4{ 'a','b' }; //s4=="ab"
string s5(3, 'a');  //s5=="aaa"
```

```
string s("hello world");
string s2 = s.substr(6, 22); //s2=="world"
string s3 = s.substr(2); //s3=="llo world"
string s4 = s.substr(); //s4==s
```

2. 改变string的其他方法

除了顺序容器中的操作以外，string还定义了额外的insert和erase版本等。

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1个插入字符的迭代器。
s.erase(pos,len);	删除从位置pos开始的len个字符。如果len被省略，则删除从pos开始直到s末尾的所有字符。返回指向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：(str不能与s相同，迭代器b和e不能指向s)

str	字符串str
str,pos,len	str中从pos开始最多len个字
cp,len	从cp指向的字符数组的前(最多)len个字符
cp	cp指向的以空字符结尾的字符数组
n,c	n个字符c
b,e	迭代器b和e指定的范围内的字符
初始化列表	花括号包围的，以逗号分隔的字符列表

- append和assign可以使用以上所有形式。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	是	否	cp
是	是	是	是	n,c
否	是	否	是	b2,e2
是	是	是	是	初始化列表

以下为实例：

```
string s("abc"), str = "XYZ";
char c[] = { 'X','Y','Z','\0' };
s.push_back('d'); //"abcd"
s.insert(s.end(), 'e'); //"abcde"
s.insert(s.end(), 2, 'f'); //"abcdeff"
s.insert(s.end(), { 'g','h' }); //"abcdeffgh"
s.insert(s.end(), c, c + 3); //"abcdeffghXYZ"
s.insert(0, str); //"XYZabcdeffghXYZ"
s.insert(0, str, 1, 2); //"YZXYZabcdeffghXYZ"
s.insert(0, c, 1); //"XYZXYZabcdeffghXYZ"
s.insert(0, c); //"XYZXYZXYZabcdeffghXYZ"
s.insert(0, 2, 'm'); //"mmXYZXYZXYZabcdeffghXYZ"
s.insert(1, { 'a','b' }); //"mabmXYZXYZXYZabcdeffghXYZ"
s.pop_back(); //"mabmXYZXYZXYZabcdeffghXY"
s.erase(s.begin()); //"abmXYZXYZXYZabcdeffghXY"
```

```
s.erase(s.begin(), s.begin() + 12); //"abcdeffghXY"
s.erase(6, 4); //"abcdefY"
s.clear(); //""
s.append("abc"); //"abc"
s.assign("XXYYZZ"); //"XXYYZZ"
s.replace(1, 3, string("ZZ")); //"XZZZZ"
s.replace(1, 3, string("ABC"), 1, 2); //"XBCZ"
s.replace(1, 2, c, 1); //"XXZ"
s.replace(1, 1, c); //"XXYZZ"
s.replace(1, 3, 3, 'a'); //"XaaaZ"
s.replace(1, 3, { 'a','b','c' }); //"XabcZ"
s.replace(s.begin() + 1, s.begin() + 4, string("QAQ")); //"XQAQZ"
s.replace(s.begin() + 1, s.begin() + 4, c, 2); //"XXYZ"
s.replace(s.begin() + 1, s.begin() + 3, c); //"XXYZZ"
s.replace(s.begin(), s.end(), 3, 'a'); //"aaa"
s.replace(s.begin(), s.end(), c, c + 3); //"XYZ"
s.replace(s.begin() + 1, s.begin() + 2, { 'a','b','c' }); //"XabcZ"
```

- 除此之外，string也重载了加号操作符，s1+s2返回s1和s2连接后的结果。注意由于标准库允许将字符字面值和字符串字面值转换成string对象，所以在需要string对象的地方可以使用这两种字面值替代。但是使用上述加号操作符时，必须保证加号左右两侧的运算对象至少有一个是string

3. string的搜索操作

   string类提供6个不同的搜索函数，每个函数有4个重载版本。搜索操作返回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，标准库将其定义为一个const string::size_type类型并初始化为值-1，而由于npos是一个无符号类型，此初始值意味着npos等于任何string的最大的可能大小。

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

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没有默认值

- 注意rfind，find_last_of和find_last_not_of是从右至左搜索，默认位置是从最后开始搜索

4. compare函数

   string的compare函数与C标准库的strcmp函数类似，根据s是等于、大于、还是小于参数指定的

字符串，返回0、正数或负数。s.compare(args)的参数args有以下六种情况

s2	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cp指向的地址开始的n2个字符进行比较

5. string和数值之间的转换

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。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

stoi(s,p,b);
stol(s,p,b);
stoul(s,p,b);
stoll(s,p,b);
stoull(s,p,b);
	返回s的起始子串(表示整数内容的子串)的数值，返回值类型分别是
	int、long、unsigned long、long long、unsigned long long。b表示转换所用的基数，默认值为10。p是size_t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

stof(s,p);
stod(s,p);
stold(s,p);
	返回s的起始子串(表示浮点数内容的子串)的数值，返回值类型分别是
	float、double、long double。参数p的作用同上。

6. 输入输出操作

os<<s;	将s写到输出流os中，返回os
is>>s;	从is中读取字符串赋给s(忽略空格、换行符和制表符，从第一个真正的字符开始读起，直到遇见下一处空白为止)，字符串以空白分隔，返回is
getline(is,s);	从is中读取一行赋给s(直到遇到换行符为止，注意换行符也被读入，但并不将换行符存入s中)，返回is

6 容器适配器

1. 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、priority_queue
   这些容器适配器都支持的类型和操作有：

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
A a;	默认构造函数，创建一个名为a的空适配器

A a(c);	创建一个名为a的适配器，带有容器c的一个拷贝，容器c必须是A的底层容器类型。例如
	`deque<int> deq{ 1,2,3 };`
	`stack<int> stk(deq); //从deq拷贝元素到stk`

关系运算符	每个适配器都支持所有关系运算符==、!=、<、<=、>、>=，这些运算符返回底层容器的比较结果
a.empty();	
a.size();	
swap(a,b); a.swap(b);	交换a和b的内容，a和b必须具有相同类型，包括底层容器类型也必须相同

- 默认情况下，stack和queue是基于deque实现的，priority_queue是基于vector实现的。但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，如

```
vector<string> svec{ "aa", "bb" };
stack<int, list<int>> stkl; //list上实现的空栈
stack<string, vector<string>> stkv(svec); //基于vector实现，初始化时保存svec的拷贝
```

- array和forward_list不能用来构造适配器(一般也不将string视作容器)。剩余的顺序容器都可用来构造stack；queue只能用list或deque构造；priority_queue只能用vector或deque构造。
- 适配器也支持用同类型同元素类型的适配器进行复制构造，以及用等号进行赋值操作。

2. 栈适配器
   栈的其他操作如下：

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

3. 队列适配器
   queue和priority_queue的其他操作如下：

q.pop();	删除queue的首元素或priority_queue的最高优先级元素，但不返回该元素值

q.front(); q.back();	只适用于queue，返回首元素/尾元素，但不删除此元素
q.top();	只适用于priority_queue，返回最高优先级元素，但不删除该元素
q.push(item); q.emplace(args);	在queue末尾或priority_queue中的恰当位置创建一个元素，其值为item，或者由args构造

1 泛型算法概述

1. 只读算法

- find，第3个参数是一个值，返回指向第一个等于该值的元素的迭代器，如果范围中无匹配值则返回第2个参数表示搜索失败。

```
auto result = find(vec.cbegin(), vec.cend(), val);
int ia[] = { 20,34 };
//使用标准库的begin和end函数获得指向ia的首元素指针和尾后指针
int* result = find(begin(ia), end(ia), val);
auto result = find(ia + 1, ia + 2, val);
```

find_if，第3个参数是一个一元谓词，返回第一个使谓词为true的元素的迭代器，如果不存在这样的元素则返回第2个参数。

```
vector<int>::iterator it = find_if(vec.begin(), vec.end(), func);
```

- count，第3个参数是一个值，返回该值出现的次数。

```
int num = count(vec.cbegin(), vec.cend(), val);
```

count_if，第3个参数是一个谓词，返回一个计数值表示谓词有多少次为true

```
count_if(vec.begin(), vec.end(), func);
```

- accumulate，定义在头文件<numeric>中，用于对范围中的元素求和，第3个参数是和的初值，决定了函数使用哪个加法运算符以及返回值类型

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

- equal，用于确定两个序列是否保存相同的值，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，前两个参数表示第一个序列中的元素范围，第3个参数是第二个序列的首元素迭代器。调用equal进行比较，容器可以不同类型，元素类型也可以不同，只要能用==来比较两个元素类型即可。但是它假定第二个序列至少与第一个序列一样长，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与第一个序列一样长。

```
bool t=equal(vec1.cbegin(), vec1.cend(), vec2.cbegin());
```

- for_each，第3个参数是一个可调用对象，对输入序列中每个元素调用此对象

```
for_each(vec.begin(), vec.end(), [](int& s) {cout << s; });
```

2. 写容器元素的算法

- fill，第3个参数是一个值，fill将这个值赋值给范围中的每个元素

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

- fill_n，参数为一个单迭代器dest、一个计数值n和一个值val，将val赋值给迭代器dest指向的元素开始的n个元素，注意从dest开始的序列至少需要包含n个元素(除非使用插入迭代器)

```
fill_n(vec.begin(), vec.size(), 10);
```

使用插入迭代器作为算法的目的位置，可实现对空容器添加元素的操作

```
fill_n(back_inserter(vec), 10, 0);
```

- copy，第3个参数是目的序列的起始位置，将输入范围中的元素拷贝到目的序列中，返回目的位置迭代器(递增后)的值，注意目的序列至少要包含于输入序列一样多的元素(除非使用插入迭代器)

```
copy(vec.begin(), vec.end(), back_inserter(vec2));
```

- replace，第3个参数是要搜索的值x，第4个参数是新值y，将所有等于x的元素替换为y

```
replace(vec.begin(), vec.end(), 0, 42);
```

很多算法都有"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_copy，该算法第3个参数是调整后序列的保存位置，第4和第5个参数分别是上述的x和y，该算法保留原序列不变，替换后的序列保存在第3个参数所指的位置

```
replace_copy(vec.begin(), vec.end(), back_inserter(ivec), 0, 42);
```

- transform，第3个参数是目的位置迭代器，第4个参数是一个可调用对象，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(目的位置迭代器与输入序列开始位置的迭代器可以是相同的)

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

3. 重排容器元素的算法

- sort，排序

```
sort(vec.begin(), vec.end());
```

重载版本的第3个参数接受一个二元谓词(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，分为一元谓词(1个参数)和二元谓词(2个参数))

```
bool isShorter(const string & s1, const string & s2) {
    return s1.size() < s2.size();
}
//按长度由短至长排序words
sort(words.begin(), words.end(), isShorter);
```

stable_sort是具有稳定性的排序算法，维持相等元素的原有顺序

```
stable_sort(words.begin(), words.end(), isShorter);
```

- unique，重排输入序列，将相邻的重复项"消除"(并不是真正删除)，使得不重复元素出现在序列开始部分，返回一个指向不重复元素之后位置的迭代器(此位置之后的元素仍然存在，但我们不知道它们的值)，一般unique对有序序列去重

```
unique(vec.begin(), vec.end());
```

- partition，第3个参数是一个谓词，使得谓词为true的值排在容器的前半部分，false的值排在后半部分，返回一个迭代器，指向最后一个使谓词为true的元素之后的位置

```
bool test(int num) {
    return num > 5;
```

```
    }
    partition(vec.begin(), vec.end(), test);
```

2 lambda表达式

1. 基本概念

- 我们可以向一个算法传递任何类别的可调用对象(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，则称它为可调用的)，可调用对象包括函数、函数指针、重载了函数调用运算符的类、lambda表达式。
- lambda表达式具有如下形式

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

  - 其中捕获列表是一个lambda所在函数中定义的局部变量列表。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：

```
    auto f = [] { return 42; };
    cout << f() << endl; //打印42
```

  - 如果忽略返回类型，则lambda根据函数体中的代码推断返回类型：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void(如果想包含多条语句，返回类型又不是void，则必须指定lambda返回类型)。
  - lambda不能有默认参数，一个lambda调用的实参数目永远与形参数目相等

2. 使用捕获列表

```
void biggies(vector<string>& words, int sz) {
    auto wc = find_if(words.begin(), words.end(),
        [sz](const string& a)
        { return a.size() > sz; }); //查找第一个长度大于sz的单词
}
```

- lambda的捕获列表中我们可以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局部变量，只有捕获列表中的变量才可以被lambda的函数体使用。
- 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，如cout

3. lambda捕获

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，然后定义了一个该类型的对象。

- 值捕获：采用值捕获的前提是变量可以拷贝，被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝。

```
    int num = 42; //局部变量
    auto f = [num] { return num; }; //值捕获
    num = 0;
    int j = f(); //j为42，因为f保存了我们创建它时num的拷贝
```

- 引用捕获：当以引用方式捕获一个变量时，必须保证在lambda执行时变量时存在的

```
    int num = 42; //局部变量
    auto f = [&num] { return num; }; //引用捕获
    num = 0;
    int j = f(); //j为0，因为f保存了num的引用，而非拷贝
```

- 隐式捕获：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=。&表示引用捕获，=表示值捕获
  - 我们可以混合使用隐式捕获和显式捕获，捕获列表中的第一个元素必须是一个&或=，表示指定了默认捕获方式为引用或值
  - 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

```
int num = 42, sum = 0; //局部变量
auto f = [&] { return num; }; //隐式引用捕获
auto g = [=] { return num; }; //隐式值捕获
auto h = [&, sum] { return num + sum; };
auto i = [=, &sum] { return num + sum; };
```

- 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

```
int num = 42;
auto f = [num]() mutable { return ++num; };
num = 0;
auto j = f(); //j为43，num为0
```

  - 而对于一个引用捕获的变量，只要其指向的不是const类型，就可以修改其值

```
int num = 42;
auto f = [&num]{ return ++num; };
num = 0;
auto j = f(); //j为1，num为1
```

4. lambda返回类型

   如果函数体不是单一的return语句，返回类型又不希望是void，则必须指定lambda返回类型，且只能用尾置返回类型

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

3 参数绑定

1. bind函数

标准库函数bind定义在头文件functional中，可将其看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来适应原对象的参数列表。调用bind的一般形式为：

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

其中，newCallable本身是一个可调用对象，arg_list是一个逗号分隔的参数列表，对应给定的callable的参数，即当我们调用newCallable时，newCallable会调用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newCallable的参数，它们占据了传递给newCallable的参数的位置，_1表示第一个参数，_2表示第二个参数，以此类推。

举例：我们首先用lambda表达式来查找大于n的数

```
    find_if(vec.begin(), vec.end(), [n](int a) { return a > n; });
```

也可以用等价的参数绑定方式来实现

```
bool checksize(int a, int n) {
    return a > n;
}
    find_if(vec.begin(), vec.end(), bind(checksize, _1, n));
```

此bind调用只有一个占位符，表示它只接受单一参数。占位符出现在arg_list的第一位置，表示该参数传递给checksize的第一个参数。所以，此bind调用生成一个可调用对象，将checksize的第二个参数绑定到n的值。

注：名字_n都定义在一个名为placeholders的命名空间中，所以我们要加上声明

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

给每一个名字都提供单的using声明很麻烦，所以也可以使用如下声明更为简洁

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

2. bind的参数

- 以下bind调用

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

会将g(x,y)映射为f(a,b,y,c,x)。

- 可以利用上述技巧来重排参数顺序。比如auto g = bind(f, _2, _1)，则调用g(x, y)即f(y, x)
- 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。如果我们希望对有些绑定的参数以引用方式传递，或是要绑定参数的类型无法拷贝，则我们需要使用标准库的ref函数，比如将上述参数a以引用方式传递，只需将a改成ref(a)即可。函数ref返回一个对象，包含给定的引用。标准库中还有一个cref函数，生成一个保存const引用的类。

4 迭代器

1. 插入迭代器

这些迭代器被绑定到一个容器上，可用来向容器插入元素

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)、c.insert(p,t)，其中p为传递给inserter的迭代器位置
*it、++it、it++	这些操作虽然存在，但不会对it做任何事情，每个操作都返回it

插入迭代器有三种类型：

①. back_inserter创建一个使用push_back的迭代器

②. front_inserter创建一个使用push_front的迭代器

③. inserter创建一个使用insert的迭代器，此函数接受第2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指的元素之前

例：调用inserter(c,iter)时，我们得到一个插入迭代器，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。现假设it是由inserter生成的迭代器，则赋值语句it=val;等价于

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

2. 流迭代器(见P359)

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。

3. 反向迭代器

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。

- v.cbegin()指向容器的首元素，v.cend()指向容器的尾元素之后的一个位置；而v.crbegin()指向容器的尾元素，v.crend()指向容器的首元素之前的一个位置
- 反向迭代器的自增(以及自减)操作的含义会颠倒过来
- 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.rbegin(), vec.rend()); //最小元素放在vec末尾
```

- 只能从既支持++也支持--的迭代器来定义反向迭代器。所以不能从forward_list或一个流迭代器创建反向迭代器
- 通过调用reverse_iterator的base成员函数可以将反向迭代器转换为对应的普通迭代器。假设it是指向第3个元素的反向迭代器，则it.base()将转换为普通迭代器，它指向第4个元素(注意将反向迭代器转换为普通迭代器，普通迭代器所指位置始终是反向迭代器所指位置的后一个位置)
- 我们也可以将普通迭代器转换为反向迭代器

```
//将指向v[2]的迭代器v.begin()+2转换为反向迭代器，此时re指向v[1]
```

```
    vector<int>::reverse_iterator re(v.begin() + 2);
```

4. 移动迭代器(见P480)

   这些专用的迭代器不是拷贝其中的元素，而是移动它们。

5 泛型算法结构

算法除了可以按是否读、写或者重排序列中的元素来分类，还可以按其所要求的的迭代器操作类分类，一共有5个迭代器类别。

1. 五类迭代器

①. 输入迭代器：只读，不写，单遍扫描，只能递增。一个输入迭代器必须支持以下操作：==、！=、*(解引用运算符，用于读取，只出现在赋值运算符的右侧)、->(箭头运算符，用于提取对象成员)、++。例如算法find和accumulate要求输入迭代器，而istream_iterator是一种输入迭代器

②. 输出迭代器：只写，不读，单遍扫描，只能递增。一个输出迭代器必须支持以下操作：++、*(解引用运算符，用于写入，只出现在赋值运算符的左侧)。用作目的位置的迭代器通常都是输出迭代器，例如copy的第3个参数就是输出迭代器，ostream_iterator也是输出迭代器。

③. 前向迭代器：可读写，多遍扫描，只能递增。支持所有输入和输出迭代器操作。例如replace要求前向迭代器，forward_list上的迭代器是前向迭代器。

④. 双向迭代器：可读写，多遍扫描，可递增递减。除了支持前向迭代器的操作之外，还支持--运算符。例如reverse要求双向迭代器，除了forward_list之外其他标准库容器都提供符合双向迭代器要求的迭代器。

⑤. 随机访问迭代器：可读写，多遍扫描，支持全部迭代器运算。除了支持双向迭代器的操作之外，还支持以下操作：<、<=、>、>=、+n、+=、-n、-=、-(两个迭代器相减，得到距离)、下标运算符(it[n]，与*(it[n])等价)。例如算法sort就要求随机访问迭代器，
array、deque、string、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。

2. 算法形参模式

大多数算法具有如下4种形式之一：

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

- beg和end表示算法所操作的输入范围
- 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或一个ostream_iterator，插入迭代器会将新元素直接添加到容器中，而ostream_iterator会将数据写入到一个输出流。
- 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输入范围中的元素与第一个输入范围中的元素结合来进行一些运算。只接受单独beg2的算法假定从beg2开始的范围与beg和end所表示的范围至少一样大。

3. 算法命名规范

- 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
- _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_if版本(注意并不是重载)，该版本接受一个谓词来代替元素值
- 区分拷贝元素的版本(_copy)和不拷贝的版本，有些算法还同时提供_copy和_if版本，如remove_copy_if

6 特定容器算法

链表类型list和forward_list定义了几个成员函数形式的算法，特别地，他们定义了独有的sort、merge、remove、reverse和unique。因为通用版本的sort要求随机访问迭代器，而这两种数据结构只分别提供双向迭代器和前向迭代器。这些链表版本的算法的性能比对应的通用版本好得多。

- 以下是list和forward_list成员函数版本的算法，这些操作都返回void

lst.merge(lst2) lst.merge(lst2,comp)	将来自lst2的元素合并入lst。lst和lst2都必须是有序的。元素将从lst2中删除。在合并之后，lst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，此算法是链表类型数据结构所特有的。链表调用的接口是lst.splice(args)，前向链表调用的接口是flst.splice_after(args)，以下给出参数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必须是表示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的元素

- 链表算法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。

1 关联容器概述

- 关联容器的迭代器都是双向的
- 对于有序关联容器，关键字类型必须定义元素比较的方法，默认使用＜运算符，也可自定义一个严格弱序(可看作"小于等于")：
  - 两个关键字不能同时"小于等于"对方
  - 若k1"小于等于"k2，且k2"小于等于"k3，则必须k1"小于等于"k3
  - 若存在两个关键字使得任何一个都不"小于等于"另一个，则称它们等价。若k1等价于k2，且k2等价于k3，则必须k1等价于k3

  为了使用自定义操作，比较操作类型必须是一种函数指针类型：

```
bool cmp(const int& a, const int& b){
    return a > b;
}
    multiset<int, decltype (cmp)*> A(cmp);
```

  注意用cmp来初始化A对象，这表示当我们向A添加元素时，通过调用cmp来为这些元素排序
- pair类型定义在头文件utility中，有以下操作

pair<T1,T2> p;	p是一个pair，两个类型分别为T1和T2的成员都进行值初始化
pair<T1,T2> p(v1,v2); pair<T1,T2> p={v1,v2}; pair<T1,T2> p{v1,v2};	p是一个成员类型为T1和T2的pair，first和second成员分别用v1和v2进行初始化
make_pair(v1,v2);	返回一个用v1和v2初始化的pair，其类型由v1和v2的类型推断出来
p.first; p.second;	
p1 relop p2	关系运算符(＜，＞，＜=，＞=)按字典序定义，例如当p1.first＜p2.first或!(p2.first＜p1.first)&&p1.second＜p2.second成立时，p1＜p2为true. 关系运算符利用元素的＜运算符来实现
p1==p2 p1!=p2	当first和second成员分别相等时，两个pair相等. 相等性判断利用元素的==运算符实现

2 关联容器操作

关联容器还定义了以下类型：

key_type	此容器类型的关键字类型
mapped_type	每个关键字关联的类型，只适用于map类型的容器
value_type	对于set类型的容器，与key_type相同； 对于map类型的容器，为pair<const key_type，mapped_type>

1. 关联容器迭代器

- map和set类型都支持begin、end等操作，我们可以用这些函数获取迭代器，然后遍历容器
- 当解引用一个关联容器迭代器时，我们会得到一个类型为容器的value_type的值的引用
- 虽然set类型同时定义了iterator和const_iterator类型，但两种类型都只允许只读访问set中的元素，与不能改变一个map元素的关键字一样，一个set中的关键字也是const的，可以用一个set迭代器来读取元素的值，但不能修改
- 我们通常不对关联容器使用泛型算法，关键字是const这一特性意味着关联容器只可用于只读算法，但很多这类算法都需要搜索序列，所以使用关联容器自己定义的专用find成员会比调用泛型find快得多。在实际编程中，对一个关联容器使用算法，一般要么是将它当作一个源序列，要么当作一个目的位置。

2. 添加元素

c.insert(v) c.emplace(args)	v是value_type类型的对象，args用来构造一个元素。 •对于不包含重复关键字的容器，当元素的关键字不在c中时才插入或构造元素，函数返回一个pair，其first成员是一个指向具有指定关键字的元素的迭代器，second成员是一个指示插入是否成功的bool值 •对于包含重复关键字的容器，总会插入或构造给定元素，函数返回一个指向新元素的迭代器
c.insert(b,e) c.insert(il)	b和e是迭代器，表示一个c::value_type类型值的范围；il是这种值的花括号列表。函数返回void。对于不包含重复关键字的容器，只插入关键字不在c中的元素；对于包含重复关键字的容器，则会插入范围中的每个元素。
c.insert(p,v) c.emplace(p,args)	类似insert(v)和emplace(args)，但将迭代器p作为一个提示，指出从哪里开始搜索新元素应该存储的位置。函数返回一个迭代器，指向具有给定关键字的元素。

3. 删除元素

c.erase(k)	从c中删除每个关键字为k的元素，返回一个size_type值，指出删除的元素的数量，若返回值为0则表示想删除的元素并不在容器中

c.erase(p)	从c中删除迭代器p指定的元素。p必须指向c中一个真实元素。返回一个指向p之后元素的迭代器，若p指向c中的尾元素则返回c.end()
c.erase(b,e)	删除迭代器对b和e所表示的范围中的元素，返回e

4. map和unordered_map的下标操作

c[k]	返回关键字为k的元素，如果k不在c中，则添加一个关键字为k的元素，并对其进行值初始化
c.at(k)	访问关键字为k的元素，带参数检查，若k不在c中，则抛出一个out_of_range异常

- 比如map<string,int> m; m["A"] = 1; 将先在m中搜索关键字为A的元素，未找到，然后将一个新的关键字-值对插入m中，并进行值初始化为0，最后提取出新插入的元素并赋值为1
- 由于下标运算符可能插入一个新元素，所以我们只可以对非const的map或unordered_map使用下标操作
- 对一个map进行下标操作时，会得到一个mapped_type对象(左值)；而解引用一个map迭代器则会得到一个value_type对象

5. 访问元素

c.find(k)	返回一个迭代器，指向第一个关键字为k的元素，若k不在容器中，则返回尾后迭代器
c.count(k)	返回关键字等于k的元素的数量。对于不允许重复关键字的容器，返回值永远是0或1
c.lower_bound(k)	返回一个迭代器，指向第一个关键字不小于k的元素。不适用于无序容器。
c.upper_bound(k)	返回一个迭代器，指向第一个关键字大于k的元素。不适用于无序容器。
c.equal_range(k)	返回由两个迭代器构成的pair，表示关键字等于k的元素的范围。若k不存在，则pair的两个成员均等于指向可以插入的位置的迭代器。

- 下标运算符可能会插入元素，所以要搜索关键字时应使用find
- 在multimap或multiset中查找并访问给定关键字的元素有三种常用的方法：

①. 使用find和count

```
multimap<string, int> m = { {"A",1},{"A",2},{"B",3} };
auto num = m.count("A");
auto iter = m.find("A");
while (num) {
    cout << iter->second << endl;
    ++iter; --num;
}
```

②. 使用lower_bound和upper_bound

```
auto beg = m.lower_bound("A");
```

```
    auto end = m.upper_bound("A");
    while (beg != end) {
        cout << beg->second << endl;
        ++beg;
    }
```

③. 使用equal_range

```
    auto pos = m.equal_range("A");
    while (pos.first != pos.second) {
        cout << pos.first->second << endl;
        ++pos.first;
    }
```

3 无序容器

- 无序关联容器使用一个Hash函数和关键字类型的==运算符来组织元素，用一个hash<key_type>类型的对象来生成每个元素的Hash值
- 除了Hash管理操作之外，无序容器还提供了与有序容器相同的操作(find,insert等)，这意味着我们曾用于map和set的操作也能用于unordered_map和unordered_set，通常一个无序容器可以和有序容器互相替换，只是元素的输出顺序不同
- 无序容器在存储上组织为一组桶，每个桶保存零个或多个元素，无序容器使用一个Hash函数将元素映射到桶。如果容器允许重复关键字，所有具有相同关键字的元素也都会在同一个桶中。

桶接口	
c.bucket_count()	正在使用的桶的数目
c.max_bucket_count()	容器能容纳的最多的桶的数量
c.bucket_size(n)	第n个桶中有多少个元素
c.bucket(k)	关键字为k的元素在哪个桶中
桶迭代	
local_iterator	可以用来访问桶中元素的迭代器类型
const_local_iterator	桶迭代器的const版本
c.begin(n), c.end(n)	桶n的首元素迭代器和尾后迭代器
c.cbegin(n), c.cend(n)	与前两个函数类似，但返回const_local_iterator
哈希策略	
c.load_factor()	每个桶的平均元素数量，返回float值
c.max_load_factor()	c试图维护的平均桶大小，返回float值。c会在需要时添加新的桶，以使得load_factor小于等于max_load_factor
c.rehash(n)	重组存储，使得bucket_count大于等于n且bucket_count大于size/max_load_factor
c.reserve(n)	重组存储，使得c可以保存n个元素且不必rehash

- 标准库为内置类型(包括指针)提供了hash模板，而对于关键字类型为自定义类类型的无序容器，我们必须提供我们自己的hash模板版本

```
struct book {
    string isbn;
    int num;
};
size_t hasher(const book& sd) {
    return hash<string>()(sd.isbn);
}
bool eqOp(const book& lhs, const book& rhs) {
    return lhs.isbn == rhs.isbn;
```

```
}
using SD_multiset = unordered_multiset<book,
    decltype (hasher)*, decltype (eqOp)*>;
SD_multiset bk(42, hasher, eqOp);
```

以上我们的hasher函数使用一个标准库hash类型对象来计算isbn成员的Hash值，该hash类型建立在string类型之上。类似地，eqOp函数通过比较isbn来比较两个book。接下来我们为unordered_multiset定义了一个类型别名以作简化，而在定义bk对象时参数分别是桶大小、Hash函数指针和相等性判断运算符指针。如果我们的类已经定义了==运算符，则我们可以只重载Hash函数：

```
unordered_set<book, decltype (hasher)*> bk(42, hasher);
```