@阿西拜-南昌

顺序容器

C++ Primer第五版　　第9章

概述

容器库

操作

vector增馋

string

容器适配器

为程序员提供控制元素存储和访问顺序的能力

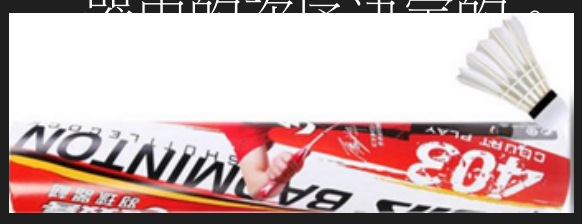
@阿西拜-南昌

顺序容器概述

容器是一种容纳特定类型对象的集合。

每种容器都是性能和功能的权衡。

- C++的容器分为顺序容器、关联容器。
- 顺序容器的元素排列由元素添加到容器中的次序决定的。



顺序容器类型	
头文件	类型
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可能很慢
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 不可随机访问	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	单向链表。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。不能添加或删除元素
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/删除速度快

如果不确定该使用哪种容器，那么可以在程序中只使用vector和list公共的操作：使用迭代器，不使用下标操作，避免随机访问。这样，在必要时选择vector或list都很方便

array 大小固定　通常 vector 是最好的选择
list forward_list 添加删除快　但是 内存开销大
如果不知道用哪个容器　，可以用 vector 和list list 复制读取后拷贝到vector
使用迭代器　，不使用下标操作　避免随机访问
随机访问 vector　or　deque
中间插入 or 删除　用 list　or forward_list
头尾插入 删除 deque

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。
- 另外一些操作仅针对顺序容器、关联容器或无序容器。
- 还有一些操作只适用于一小部分容器。

容器均定义为模板类：

- list<Sales_data>
- deque<double>

容器也可以装容器：

- vector< vector<string> > lines;

```
//假定noDefault是一个没有默认构造函数的类型
vector<noDefualt> 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
```

```
class A {
public:
        A(string b) { a = b;  }
private:
       string a;
};


int main()
{
       string AA("hello");
       vector<A> objA(100,AA);
       return 0;
}
```

迭代器：iterator

@阿西拜-南昌

- 所有容器都提供成操作
  - 可以访问元素（使用解引用实现）
  - 递增运算符（从当前元素移动到下一个元素）

迭代器范围由一对迭代器表示:（左闭合区间） [begin, end)

通常被称为begin，end,或者是first，last（有些误导）

```
while (begin != end) {        //如果相等，则范围为空
      *begin = val;           //正确：范围非空
      ++begin;                 //移动迭代器到下一个元素
}
```

每个容器都定义了很多类型，为了使用这些类型，必须显式使用其类型

```
list<string>::iterator iter;
vector<int>::difference_type cout;
```

begin和end有多个版本：带r的返回反向迭代器；以c开头的返回const迭代器：

```
list<string> a = { "Milton", "Shakespear", "Austen"};
auto it1 = 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
it1 = it3; //错误
it3 = it1; //正确
```

c开头是C11的支持auto 当不需要访问的时候使用

实际上有两个名为begin的成员。一个是const成员，返回const_iterator类型，另一个是非常量成员，返回iterator类型。rbegin、end和rend的情况类似。

```
iterator begin()
{
      ......
}
```

```
const_iterator begin() const
{
      ......
}
```

```
const list<string> b = { "Milton", "Shakespear", "Austen" };
list<string>::iterator cit1 = b.begin(); //错误,this指针指向的是const对象
list<string>::const_iterator cit2 = b.begin(); //正确
```

容器定义和初始化

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 c 为空
C c1(c2) C c1=c2	c1 初始化为 c2 的拷贝。c1 和 c2 必须是相同类型（即，它们必须是相同的容器类型，且保存的是相同的元素类型；对于 array 类型，两者还必须具有相同大小）
C c{a,b,c...} C c={a,b,c...}	c 初始化为初始化列表中元素的拷贝。列表中元素的类型必须与 C 的元素类型相容。对于 array 类型，列表中元素数目必须等于或小于 array 的大小，任何遗漏的元素都进行值初始化
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 C 的元素类型相容（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	
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 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的拷贝

对于array之外的容器类型，初始化列表还隐含地制定了容器的大小

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必须相同

```
//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：将const char*元素转换为string
forward_list<string> words(articles.begin(), articles.end());

//两个迭代器表示一个范围，拷贝元素，直到（但不包括）it指向的元素
deque<string> authList(authors.begin(),it);
```

与顺序容器大小相关的构造函数

```
//顺序容器（array除外）提供了一个构造函数
//接受一个容器大小和一个（可选的）元素初始值
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stirngs；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
```

只有顺序容器才接收大小参数，关联容器(array set map) 不支持

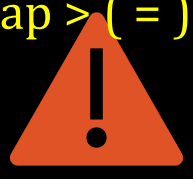
标准库array具有固定大小

```
//标准库array的大小是类型的一部分
array<int, 5> ia1; //5个默认初始化的int
array<int, 5> ia2 = {0,1,2,3,4};
array<int, 5> ia3 = {42}; //ia3[0]为42，剩余元素为0
//内置数组类型不能进行拷贝，或对象赋值操作，但array并无此限制
int digs[5] = {0,1,2,3,4};
int  cpy[5]  =  digs;  //错误：内置数组不支持拷贝或赋值
array<int, 5> digits = {0,1,2,3,4};
array<int, 5> copy = digits; //正确：只要数据类型匹配即合法
```

容器赋值运算

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
c={a,b,c...}	将 c1 中元素替换为初始化列表中元素的拷贝（array 不适用）
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	
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

swap > (=)



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（容器类型为array和string的情况除外）

```
//赋值运算后，两者的大小都与后边容器的原大小相同
c1 =  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
```

只支持顺序容器

- array不支持assign，也不允许花括号包围的值列表进行赋值（要求类型相同）。
- assign允许从一个不同但相容的类型赋值，或是从容器的一个子序列赋值

```
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());

//等价于slist1.clear()
//后跟slist1.insert(slist1.begin(),10,"Hiya!");
list<string> slist1(1); //一个元素，为空string
slist1.assign(10,"hiya!"); //10个元素，每个都是“Hiya！”
```

旧元素被替换，传递给assign的迭代器不能指向调用assign的容器

swap操作交换两个相同类型容器的内容

除array外，swap不对任何元素进行拷贝、删除或插入操作。元素不会被移动，意味着，除string外，迭代器、引用和指针在swap操作之后都不会失效。

```
vector<string> svec1(10);
vector<string> svec2(24);
swap(svec1,svec2);//调用完成后svec1包含24个string元素
```

容器大小操作

关系运算符左右两边必须类型相同

```
vector<int> v1 = {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};
v1<v2; //true
v1<v3; //false
v1 == v4; //true;
v1 == v2;//false

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

顺序容器操作

顺序容器所特有的操作

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；	
forward_list 不支持 push_back 和 emplace_back。	
vector 和 string 不支持 push_front 和 emplace_front。	
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
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

但是可以用insert( ) + x.begin( )

向一个vector、string或deque插入元素会使所有指向容器的迭代器、指针和引用失效。

```
string word;
while (cin>>word)
    container.push_back(word);
```

容器元素是拷贝

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

list、forward_list和deque容器支持将元素插入到容器头部

```
list<int> ilist;
//将元素添加到ilist开头，执行完毕后，ilist保存序列3、2、1、0
for(size_t ix = 0; ix != 4; ++ix)
    ilist.pus_front(ix);
```

比较耗时

insert成员提供了更一般的功能

```
//由于迭代器可能指向尾后，而且从开始位置插入元素很常用
//因此insert函数将元素插入到迭代器所指的位置之前
slist.insert(iter,"Hello!"); //将“Hello！”添加到iter之前的位置

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,但可以插入到begin()之前
```

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

插入范围内元素

```
//将10个元素插入到svec的末尾，并将所有元素都初始化为 “Anna”
svec.insert(svec.end(),10,"Anna");

//接受一对迭代器，或一个初始化列表
vector<string> v = {"squasi","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：迭代器表示要拷贝的范围，不能指向与目的位置相同的内容
slist.insert(slist.begin(),slist.begin(),slist.end());
```

使用insert返回值

```
//C++11，insert返回新加入元素的迭代器，如果不插入任何元素，返回第一个参数
list<string> lst;
auto iter = lst.begin();
while(cin>>word)
        iter = lst.insert(iter,word); //等价于调用push_front
```

使用emplace操作：构造元素而不是拷贝元素

```
//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("101-1-1",24,15.99);
//错误：没有接受三个参数的push_back版本
c.push_back("101-1-1",24,15.99);
//正确：创建一个零时的Sales_data对象传递给push_back
c.push_back(Sales_data("101-1-1",24,15.99));

//iter指向c中的一个元素，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,"101-1-1"); //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("101-1-1",24,15.99);
```

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，所以 传递的参数必须与元素类型的构造函数相匹配

访问元素

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。

front 都有， back 就forward_list 没有

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

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。

```
//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val2是c中第一个元素值的拷贝
    auto val = *c.begin(), val2 = c.front();
    //val3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
}
```

直接使用：front back
间接使用：begin end 需要解引用

访问成员函数返回的是引用

```
if(!c.empty()){
    c.front()=42;
    auto &v = c.back();
    v = 1024;
    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
}
```

使用auto变量保存这些函数的返回值，如果希望使用此变量能够改变元素的值，必须记得定义为引用类型

删除元素

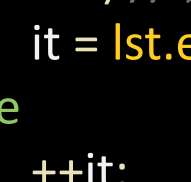
顺序容器的删除操作

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，

forward_list 不支持 pop_back；vector 和 string 不支持 pop_front。

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
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



删除deque中除首尾之外的任何元素都会使得迭代器、引用和指针都失效。指向vector或stirng中删除点之后位置的迭代器、引用和指针都会失效

pop_front和pop_back成员函数

```
while(!ilist.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
}
```

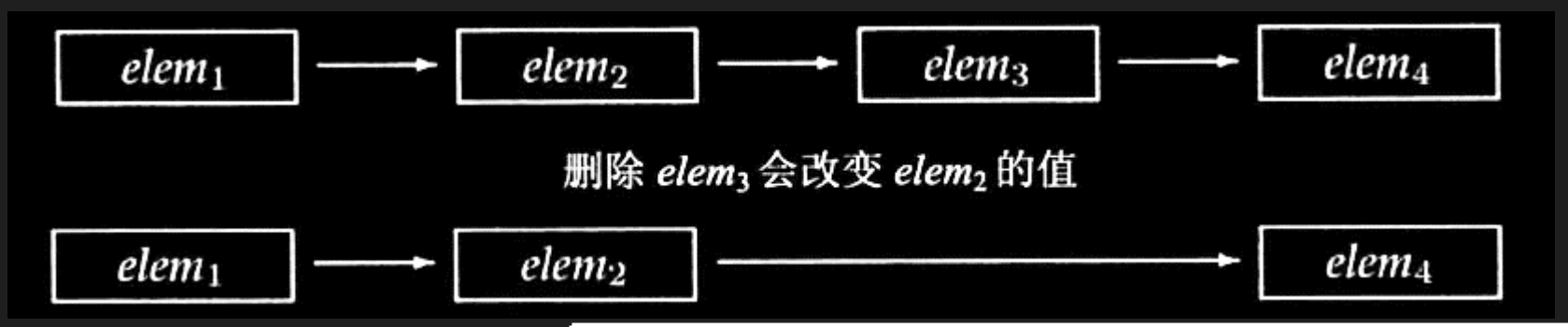
从容器内部删除元素

```
//下面的循环删除一个list中的所有奇数元素
list<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;

//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

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

elem1指向我们要输出的第一个元素
elem2指向我们要输出的最后一个元素之后的位置

特殊的forward_list操作

@阿西拜-南昌



单向列表没有简单的方法来获取一个元素的前驱

在forward_list中插入或删除元素的操作

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)	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) lst.insert_after(p,b,e) lst.insert_after(p,il)	
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)	

```
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()) {//如果有元素要处理
      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直线的元素
      }
}
```

改变容器大小

@阿西拜-南昌

顺序容器大小操作

resize 不适用于 array

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

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。

```
list<int> ilist(10,42); //10个int：每个值都是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个职位-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添加新元素的时候必须提供初初始值 or 该元素类型的构造函数

insert或erase都返回迭代器，更新迭代器很容易。

使用失效的迭代器、指针or引用是严重的运行时错误

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

程序输出 11 33 55 77 99

管理迭代器： 使用迭代器 必须保持有效
每次改变容器的操作之后都必须重新定位迭代器

不要缓存end符合的迭代器

```
//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);
    ++begin;//向前移动begin跳过我们刚刚加入的元素
}
```

```
while(begin !=v.end()) //更安全的方法：每个循环重新计算end
```

如果在一个循环 插入 or 删除 元素（deque string vector ） 不要缓冲end返回的迭代器
必须在每次 操作后 重新调用end（ ） 而不能再循环开始的时候保存它返回的迭代器

vector对象是如何增长的

@阿西拜-南昌

如果没有空间容纳新元素，就必须分配新的空间来保存已有元素和新元素。
如果每添加一个元素，vector就执行一次内存的分配和释放，性能会慢到不可接受。

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。	
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

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

标准库采用减少容器空间重新分配次数的策略

```
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;
//项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()<<endl;
```

C:\Windows\system32\cmd.exe

```
ivec:size: 0 capacity: 0
ivec:size: 24 capacity: 28
```

C:\Windows\system32\cmd.exe

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

```
ivec.reserve(50);//将capacity只是设置为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;
```

```
//添加元素用光多余容量
while(ivec.size() != ivec.capacity())
    ivec.push_back(0);
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;
```

C:\Windows\system32\cmd.exe

```
ivec:size: 51 capacity: 75
```

```
ivec.shrink_to_fit();//要求归还内存，只是一个请求，标准库并不保证退还内存
```

每个vector实现都可以选择自己都内存分配策略 ， 要求归还多余的内存给系统
在迫不得已的情况才会分配新的内存空间

额外的string操作

容器大小管理操作	
n、len2 和 pos2 都是无符号值	
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
string s(s2,pos2)	s 是 string s2 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 ←out_of_range异常
string s(s2,pos2,len2)	s 是 string s2 从下标 pos2 开始 len2 个字符的拷贝。若 pos2>s2.size()，构造函数的行为未定义。不管 len2 的值是多少，构造函数至多拷贝 s2.size()-pos2 个字符

```
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 s1(cp);                  //拷贝cp中的字符直到遇到空字符；s1=="Hello World!!!"
string s2(noNull,2);            //从noNull拷贝两个字符；s2 == "Hi"
string s3(noNull); //未定义：noNull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，直到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
```

子字符串操作	
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

```
string s("hello world");
string s2 = s.substr(0,5); //s2=hello
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
```

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

//还提供了接受C风格字符数组的insert和assign版本
const char *cp = "Stately, plump Buck";
s.assign(cp,7); //s == "Stately"
s.insert(s.size(), cp+7); //s == "Stately,plump Buck"

//我们也可以指定来自其它string或字符串的字符插入到当前string中
string s = "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());
```

append和replace函数

```
//append操作是在string末尾进行插入操作的一种简写形式
string s("C++ Primer"), s2 = s;
s.insert(s.size(), " 4th Ed."); // s=="C++ Primer 4th Ed."
s2.append(" 4th Ed."); //等价方法，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
//s.replace(11,3,"Fifth");也可以，长度无需一样
```

string搜索操作
string提供了6个不同的搜索函数

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。	
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

每个函数都有4个重载版本

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 无默认值

```
string name("AnnaBelle");
auto pos1 = name.find("Anna"); //pos1 == 0

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);

string dept("03714p3");
//返回5——字符 ‘p’的下标
auto pos = dept.find_first_not_of(numbers);
```

每个搜索操作都返回一个string::size_type值

```
string numbers("0123456789"), name("r2d2");

string::size_type pos = 0;  //pos:开始查找的位置
//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
}
```

```
C:\Windows\system32\cmd.exe
found number at index: 1 element is 2
found number at index: 3 element is 2
```

```
string river("Mississippi");
auto first_pos = river.find("is");       //返回1
auto last_pos = river.rfind(“is”);       //逆向查找，返回4
```

s.compare的几种参数形式	
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 个字符与指针 cp 指向的地址开始的 n2 个字符进行比较

比较：字典序
一个一个比
比完才比长短

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（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）
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）的数值，返回值类型分别是 float、double 或 long double。参数 p 的作用与整数转换函数中一样

```
int i = 42;
string s = to_string(i);//将整数i转换为字符表示形式
double d = stod(s); //将字符串s转换为浮点数

string s2 = "pi = 3.14!!!";
//stod直到遇到不是数值的字符停止
d = stod(s2.substr(s2.find_first_of(“+-.0123456789")));//d=3.14
//d = stod("3.14!!!");
```

容器适配器

@阿西拜-南昌

三个顺序容器适配器：stack、queue和priority_queue

一个适配器是一种机制，能使得某事物的行为看起来像另一种事物一样。

- 例如：stack适配器接受一个顺序容器（array和forward_list除外），并使其操作起来像一个stack一样。

所有的适配器都要求容器具有添加、删除以及方便访问尾元素的能力

所有容器适配器都支持的操作和类型	
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;	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
关系运算符	每个适配器都支持所有关系运算符：==、!=、<、<=、>和>= 这些运算符返回底层容器的比较结果
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

栈特有的操作	
栈默认基于 deque 实现，也可以在 list 或 vector 之上实现。	
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

队列和优先队列特有的操作	
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。	
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
q.push(item) q.emplace(*args*)	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

```
stack<int> intStack;  // empty stack

// fill up the stack
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix);   // intStack holds 0 . . . 9 inclusive

// while there are still values in intStack
while (!intStack.empty()) {
    int value = intStack.top();
    // code that uses value
    cout << value << endl;
    intStack.pop(); // pop the top element,
}
```

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。我们可以创建适配器时，通过第二个参数来指定容器类型。

```
stack<int,vector<int>> intStack;
```