第13讲 STL(Standard Template Library)

目录 CONTENTS

13.1 STL简介

- STL(Standard Template Library)，即标准模板库，是一个具有工业强度的，高效的C++程序库。
- 它包含于C++标准程序库（C++ Standard Library）中，是ANSI/ISO C++标准中最新的也是极具革命性的一部分。
- STL中的代码主要采用类模板和函数模板的方式，极大地提高了编程效率。
- 为广大C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。

亚历山大·斯特潘诺夫

STL的特点

- （1）数据结构和算法的分离，尽管这是个简单的概念，但这种分离确实使得STL变得非常通用。

- 例如，由于STL的sort()函数是完全通用的，你可以用它来操作几乎任何数据集合，包括链表，容器和数组。

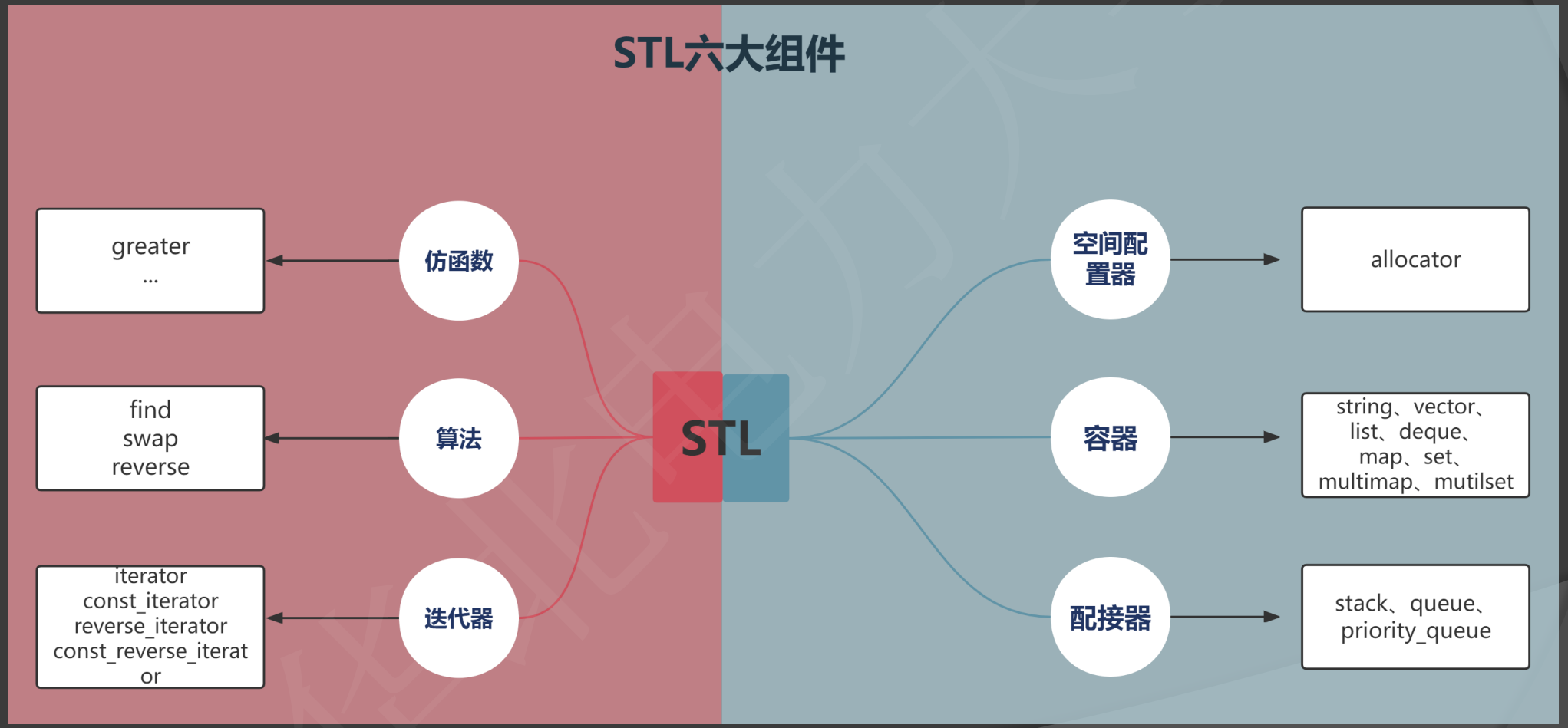
STL的特点

- （2）不是面向对象的，为了具有足够通用性，STL主要依赖于模板。
- 在STL中找不到明显的类继承关系。这看似是一种倒退，但这正好是使得STL的组件具有广泛通用性的底层特征。
- 另外，由于STL是基于模板，内联函数的使用使得生成的代码短小高效。

STL的特点

- （3）具有高性能，如map可以高效地从十万条记录里面查找出指定的记录，因为map是采用红黑树的变体实现的。

STL中六大组件


STL六大组件
greater
...
仿函数
find
swap
reverse
算法
iterator
const_iterator
reverse_iterator
const_reverse_iterat
or
迭代器
STL
空间配置器
allocator
容器
string、vector、
list、deque、
map、set、
multimap、mutilset
配接器
stack、queue、
priority_queue

STL中六大组件

1. 容器（Container），是一种数据结构，如list，vector，和deque，以类模板的方法提供。为了访问容器中的数据，可以使用由容器类输出的迭代器。

2. 迭代器（Iterator），提供了访问容器中对象的方法。

STL中六大组件

3. 算法（Algorithm），是用来操作容器中的数据的模板函数。
4. 仿函数/函数对象（Functor），就是一个行为类似于函数的对象，重载了运算符函数operator()。
5. 适配器（Adaptor，配接器），用来修饰容器、函数对象。
6. 空间配置器（Allocator），负责空间的配置与管理。

对容器、算法和迭代器的理解

什么是容器、算法和迭代器?

```
int array[100];
int *p = array;
```

数组就是容器，而 int * 类型的指针变量就可以作为迭代器

sort 算法可以作用于该容器上，对其进行排序。

```
sort(array,array+70);    //将前70个元素排序
```

STL的组织

在C++标准中，STL被组织为下面的13个不带.h后缀的头文件：

<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。

13.2 容器类

- 容器类是容纳一组对象或对象集的类。
- STL中容器分为两类：
  - 顺序容器（Sequence Container）
  - 关联容器（Associative Container）
- 共包含最基本的7个标准类模板。

顺序容器与关联容器

- 顺序容器管理若干对象的有限线性集合，所有对象都是同一类型。STL中三种基本顺序容器是：向量（vector）、线性表（list）、双向队列（deque）。

- 关联容器提供了基于Key的数据的快速检索能力。元素被排好序，检索数据时可以二分搜索。STL有四种关联容器：集合（set）、多元集合（multiset）、映射（map）、多元映射（multimap）。

容器的共通能力

- 所有容器中存放的都是值而非引用，即容器进行安插操作时内部实施的是拷贝操作。因此容器的每个元素必须能够被拷贝。如果希望存放的不是副本，容器元素只能是指针。
- 所有元素都形成一个次序（order），可以按相同的次序一次或多次遍历每个元素
- 各项操作并非绝对安全，调用者必须确保传给操作函数的参数符合需求，否则会导致未定义的行为。

容器的共通能力

STL容器中的元素要满足条件：

- 必须能够通过拷贝构造函数进行复制
- 必须可以通过赋值运算符完成赋值操作
- 必须可以通过析构函数完称销毁动作
- 序列式容器元素的默认构造函数必须可用
- 某些动作必须定义operator ==，例如搜寻操作
- 关联式容器必须定义出排序准则，默认情况是重载operator <

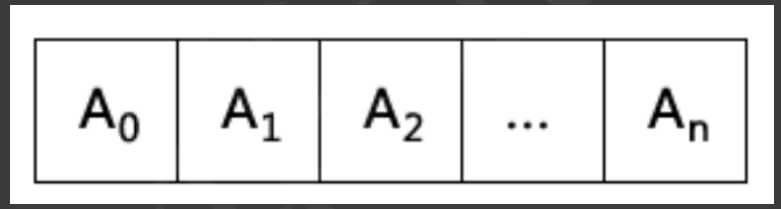
13.2.1 顺序容器

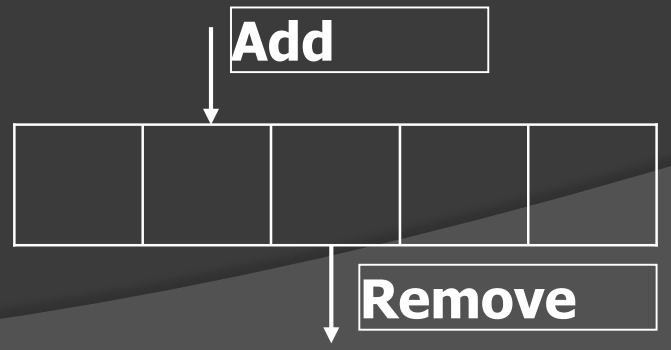
1.向量vector

- 向量vector是用于容纳不定长线性序列的容器，提供对序列的快速随机访问（也称直接访问）。 vector的数据安排以及操作方式，与C++中的数组非常相似。
- 区别在于：C++中的数组是固定大小的，分配给C++数组的空间数量在数组的生存期内是不会改变的。
- 向量vector是动态结构，它的大小不固定，可以在程序运行时增加或减少。vector中的元素在内存是连续存储的，可以直接访问。

1.向量vector

- 允许顺序访问和随机访问
  - 根据下标随机访问某个元素时间为常数
  - 在尾端增删元素具有更高性能(大部分情况下是常数时间)
  - 在中间插入慢
- 所有STL算法都能对vector操作。
- 需要导入头文件#include <vector>。





（1）向量类型对象的定义与初始化

默认构造函数，它创建一个没有任何元素的空向量vec1，其vec1. size()=0

```
vector<int>  vec1;                                  //默认初始化，vec1为空
vector<int>  vec2(vec1);                            //使用vec1初始化vec2
vector<int> vec3(vec1.begin(),vec1.end()); //使用vec1初始化vec3
```

拷贝构造函数，用另一个向量vec1来初始化此向量vec2。

从另一个支持const_iterator的容器vec1中选取一部分[begin,end)区间的元素 ）来构造一个新的vector，即vec3。

（1）向量类型对象的定义与初始化

创建一个大小位10的向量vec4，没有指定元素的初始化值，那么标准库将自行提供一个元素初始值进行。
如果为int型数据，那么标准库将用 0 值创建元素初始化值；
如果是含有构造函数的类类型（如 string）的元素，标准库将用该类型的默认构造函数创建元素初始化值；

```
vector<int> vec4(10);              //10个值为0的元素
vector<int> vec5(10,4);            //10个值为4的元素
vector<string> vec6(10,"hello");   //10个值为hello的元素
```

创建一个大小为10的向量vec5，该向量中所有的10个元素都初始化为4。

将vec6声明为一个元素类型为string的向量容器对象，每个元素都是hello。

例 vector的定义并初始化方法示例

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int num[10] = {5,5,5,5,5,5,5,5,5,5};
    int i;
    vector <int> v1(10,5);  // 初始化有10个元素、其值都是5的向量v1
    vector <int> v2(10);    // 初始化size为10、元素值为默认值0的向量v2
    vector <int> v3(v1);
    vector <int> v4(v1.begin(), v1.end());
    vector <int> v5(&num[0],&num[10]);
    // 指针可以作为迭代器来使用
```

函数end()返回容器中最后一个元素下一个位置,即指向容器中末端元素的下一个，指向一个不存在元素。

```
for (i=0;i<10;i++)  // 将向量v2中的10个元素值均修改为5
        v2[i]=5;
  // 五个vcetor对象是相等的，可以用operator==来判断
  if (v1==v2)
        cout<<"v1==v2"<<endl;
  else
        cout<<"v1!=v2"<<endl;
  if (v1==v3)
        cout<<"v1==v3"<<endl;
  else
        cout<<"v1!=v3"<<endl;
  if (v1==v4)
        cout<<"v1==v4"<<endl;
  else
        cout<<"v1!=v4"<<endl;
  if (v1==v5)
        cout<<"v1==v5"<<endl;
  else
        cout<<"v1!=v5"<<endl;
  return 0;}
```

operator==依次判断向量中对应元素的内容

```
vector <int> v6;
v6.assign(v1.begin(), v1.end());
v6.assign(10,8);
```

assign用于将一个容器的元素复制到另一个容器中。

（2）Vector元素的访问

- 向量类似于数组，量类向模板中对下标运算符“[ ]”进行了重载，因此可以按照访问数组元素的方式来访问向量中的元素。
- vec是一个向量对象，则vec[index]就代表由index指定的位置上的元素。
- 另外，向量类中还提供了3个成员函数来访问向量中的元素。

三个访问元素的成员函数

- at(index)，返回容器中指定位置index的元素的引用。
- front()，返回第一个元素的引用（不检查容器是否为空）。如果容器非空，则front()和 vec[0]相同。
- back()，返回最后一个元素引用（不检查容器是否为空）。如果容器非空，则back ()和 vec[vec.size()-1]相同。

（3）Vector元素的遍历：迭代器类型iterator

- 尽管不同容器的内部结构各异，但它们本质上都是用来存储大量数据的，都是一串能存储多个数据的存储单元。因此，诸如数据的排序、查找、求和等需要对数据进行遍历的操作方法应该是类似的。
- 既然类似，完全可以利用泛型技术，将它们设计成适用所有容器的通用算法，从而将容器和算法分离开。
- 迭代器和 C++ 的指针非常类似，它可以是需要的任意类型，通过迭代器可以指向容器中的某个元素，如果需要，还可以对该元素进行读/写操作。

（3）Vector元素的遍历：迭代器类型iterator

- 常用的迭代器按功能强弱分为输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器。
  - 前向迭代器（forward iterator）：假设 p 是一个前向迭代器，则 p 支持 ++p，p++，*p 操作，还可以被复制或赋值，可以用 == 和 != 运算符进行比较。此外，两个前向迭代器可以互相赋值。
  - 双向迭代器（bidirectional iterator）具有前向迭代器的全部功能，除此之外，则还可以进行 --p 或者 p-- 操作（即一次向后移动一个位置）。
  - 随机访问迭代器（random access iterator）随机访问迭代器具有双向迭代器的全部功能。除此之外，假设 p 是一个随机访问迭代器，i 是一个整型变量或常量，则 p 还支持p+=i的操作。

（3）Vector元素的遍历：迭代器类型iterator

容器	对应的迭代器类型
`array`	随机访问迭代器
`vector`	随机访问迭代器
`deque`	随机访问迭代器
`list`	双向迭代器
`set / multiset`	双向迭代器
`map / multimap`	双向迭代器
`forward_list`	前向迭代器
`unordered_map / unordered_multimap`	前向迭代器
`unordered_set / unordered_multiset`	前向迭代器
`stack`	不支持迭代器
`queue`	不支持迭代器

（3）Vector元素的遍历：迭代器类型iterator

- vector类包含了一个公有的iterator，通过iterator，可以声明向量容器中的迭代器。

例如，语句：

```
vector<int>::iterator intIter;
```

将intIter声明为元素为int类型的向量容器的迭代器。

因为iterator是一个定义在vector类中的typedef，所以必须使用容器名（vector）、容器元素类型和作用域符来使用iterator。

迭代器的使用

声明了迭代器后，执行语句

```
intIter = vec.begin();
```

将使迭代器intIter指向容器vec中第一个元素。

表达式

```
++intIter
```

将迭代器intIter加 1，使其指向容器vec中的下一个元素。

迭代器的使用

表达式 *intIter 将返回当前迭代器位置上的元素（即vec[1]或vec.at(1) ）。

表达式 *(intIter+2) 将返回当前迭代器位置之后两个位置上的元素（即vec[3]或 vec.at(3)）。

vector容器中有成员函数begin()和end()。

- 函数begin()返回容器中第一个元素位置的迭代器。
- 函数end()返回容器中最后一个元素下一个位置的迭代器。即指向容器中末端元素的下一个，指向一个不存在元素。

迭代器的使用

遍历容器vec中所有元素并输出的代码为：

```
for (intIter = vec.begin(); intIter != vec.end();  intIter++)
        cout<<* intIter <<"     ";
```

每次进行循环条件判断均需去重复计算end，而vector的end()函数不是常数时间的，因此可以先计算end并缓存下来，这样能提高效率。

例如，可将上面程序段改写为：

```
vector<int>:: iterator   intIter,  itEnd;
itEnd= vec.end();
for (intIter = vec.begin(); intIter !=itEnd ;intIter++)
        cout<<* intVIter <<"     ";
```

反向迭代器

函数rbegin() 返回指向当前vector末尾的反向迭代器，
函数rend() 返回指向当前vector起始位置的反向迭代器。
所谓反向迭代器就是反向移动的迭代器。例如，语句：

```
vector<int>::reverse_iterator  intRIter;
```

将intRIter声明为元素为int类型的向量容器的反向迭代器。

逆序遍历容器intVec中所有元素并输出：

```
for (intRIter = vec.rbegin(); intRIter != vec.rend();  intRIter++)
        cout<< * intRIter  << "     ";
```

（4）插入与删除元素

- push_back()可以将一个元素添加到容器的末尾;
- pop_back()可以删除容器最后一个元素;
- clear()可以删除容器中的所有元素;
- 插入函数insert();
- 删除函数erase()。

insert() 函数

- 函数原型1：

  iterator insert(iterator loc, const TYPE &val);

- ✓ 功能：在指定位置loc前插入值为val的元素，返回指向这个元素的迭代器。

- 函数原型2：

  void insert(iterator loc, size_type num, const TYPE &val);

- ✓ 功能：在指定位置loc前插入num个值为val的元素。

- 函数原型3：

  void insert(iterator loc, input_iterator start, input_iterator end);

- ✓ 功能：在指定位置loc前插入区间[start, end)的所有元素 。

```
vector<int> v1(5,1);          //初始size()是5，每个元素都是1
vector<int> v2(6, 5);         //初始size()是6，每个元素都是5
v1.push_back(9);              //在最后位置插入一个元素9
v1.insert(v1.begin()+2, 2);  //在第2个元素后面插入一个2
v1.insert(v1.begin()+2 , 4, 3); //在第2个元素后面插入4个3
v1.insert(v1.end()-1, v2.begin(), v2.begin() + 3);
//在倒数第2个元素后面插入v2中头3个元素

vector<int>::iterator ite = v1.begin();
vector<int>::iterator end = v1.end();
for(;ite!= end; ite++)
        cout<<*ite<<endl;
```

显示结果：

```
1
1
3
3
3
3
2
1
1
1
5
5
5
9
```

erase()函数的两种用法

- 函数原型1：

  iterator erase(iterator loc);

- ✓ 功能：删除指定位置loc的元素，返回值是指向删除元素的下一位置的迭代器。

- 函数原型2：

  iterator erase(iterator start, iterator end);

- ✓ 功能：删除区间[start, end)的所有元素，返回值是指向删除的最后一个元素的下一位置的迭代器。

```
vector<int> v1(5,1);          //初始size()是5，每个元素都是1
vector<int> v2(6, 5);         //初始size()是6，每个元素都是5
v1.push_back(9);              //在最后位置插入一个元素9
v1.insert(v1.begin()+2, 2);  //在第2个元素后面插入一个2
v1.insert(v1.begin()+2 , 4, 3); //在第2个元素后面插入4个3
v1.insert(v1.end()-1, v2.begin(), v2.begin() + 3);
//在倒数第2个元素后面插入v2中头3个元素

v1.erase(v1.end()-1);         //删除最后一个元素
v1.erase(v1.begin(), v1.begin()+4); //删除最前面4个元素

vector<int>::iterator ite = v1.begin();
vector<int>::iterator end = v1.end();
for(;ite!= end; ite++)
        cout<<*ite<<endl;
```

显示结果：

```
1
3
1
3
3
2
3
1
3
1
3
1
2
5
1
5
1
5
1
5
5
5
9
```

（5）Vector信息

vector有4个成员函数可以返回向量的信息：

```
size_type size() ;  // 返回vector所容纳元素的数目
size_type capacity();
    // 返回vector在重新进行内存分配以前所能容纳的元素数量
size_type max_size();
    // 返回当前vector所能容纳元素数量的最大值
bool empty();
    // 如果当前vector没有容纳任何元素，则empty()
    // 函数返回true，否则返回false
```

例 访问向量信息的程序示

```
#include <iostream> #include <vector>
using namespace std;
int main()
{
    int N = 100;
    vector<char> vc;
    for(int i= 0; i < N; i++)
    {
        vc.push_back('f');
        cout <<vc.size()<<","<< vc.capacity() << endl;
    }
    return 0;
}
```

当进行insert或push_back等增加元素的操作时，如果此时动态数组的内存不够用，就要动态的重新分配当前大小的1.5倍的新内存区，再把原数组的内容复制过去。

capacity和size的区别

- size是当前vector容器真实占用的大小，也就是容器当前拥有多少个容器。
- capacity是指在发生realloc前能允许的最大元素数，即预分配的内存空间。
- 这两个属性分别对应两个方法：resize()和reserve()。使用resize()，容器内的对象内存空间是真正存在的。如果resize(m)，m<size()，保留前m个元素，否则补齐。
- 使用reserve()仅仅只是修改了capacity的值，容器内的对象并没有真实的内存空间。

内存管理与效率

- 关于STL容器，最令人称赞的特性之一就是只要不超过它们的最大容量，它们就可以自动增长到足以容纳你放进去的数据。
- 只要有元素需要插入而且容器的容量不足时就会发生重新分配（包括它们维护的原始内存分配和回收，对象的拷贝和析构和迭代器、指针和引用的失效）。
- 所以，避免重新分配的关键是使用reserve尽快把容器的容量设置为足够大，最好在容器被构造之后立刻进行。

例子

- 例如，假定你想建立一个容纳1-100值的vector<char>。没有使用reserve，可以像这样来做：

```
vector<char> vc;
for (int i = 0; i < 100; i++)
    vc.push_back('f');
```

- 在大多数STL实现中，这段代码在循环过程中将会导致多多次重新分配。

使用reserve()

把代码改为使用reserve()，我们得到这个：

```
vector<char>  vc;
vc.reserve(100);
for (int i = 0; i < 100; i++)
    vc.push_back('f');
```

这在循环中不会发生重新分配。

需要注意的情况

在执行插入和删除操作后，应注意迭代器的变化情况。

例如，设有vector <int> v1; v1容器中有8个元素1、2、3、3、5、3、7、8，现在要求删除该容器中所有值为3的元素，写出如下的代码：

```
for(vector<int>::iterator iter=v1.begin(); iter!=v1.end();  iter++ )
    if( *iter == 3)
        iter = v1.erase(iter);
```

由于erase()函数的返回值是指向删除元素的下一位置的迭代器，因此这段代码是错误的：

1）无法删除两个连续的“3”。例如，如果v1中的元素为1、2、3、3、5、3、7、8，执行后，v1中元素为1、2、3、5、7、8，有一个3没有删除。

2）当3位于vector最后位置的时候，程序在执行时会出错（在v1.end()上执行++ 操作）。

解决方法

在删除操作后弃用原来的迭代器，使用返回值；没有删除操作时，仍使用原来的迭代器即可。

正确的代码可写为：

```
for(vector<int>::iterator iter=v1.begin(); iter!=v1.end(); )
    if( *iter == 3)
        iter = v1.erase(iter);
    else
        iter ++ ;
```

（6）其他操作

- 不是成员函数，如sort、find、for_each等。
- sort方法是algorithm头文件里的一个标准函数，能进行高效的排序，默认是按元素从小到大排序
- 将sort方法用到vector和set中能实现多种符合自己需求的排序
- sort函数有三个参数：
  （1）第1个是要排序的起始地址。
  （2）第2个是要排序的结束地址
  （3）第3个参数是排序的方法，可以是从大到小也可是从小到大，还可以不写第三个参数，此时默认的排序方法是从小到大排序。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{    vector<int> number;
     number.insert(number.begin(),99);
     number.insert(number.begin(),98);
     number.insert(number.end(),97);
     number.insert(number.end(),92);
     number.insert(number.end(),90);
     cout<<"排序前："<<endl;
     vector<int>::iterator it;
     for (it=number.begin();it!=number.end();it++)
          cout<<*it<<endl;
     sort(number.begin(),number.end());
     cout<<"排序后："<<endl;
     for (it=number.begin();it!=number.end();it++)
          cout<<*it<<endl;
     return 0;
}
```

（6）其他操作

- find方法是在输入迭代器所定义的范围内查找单个对象的算法。
- find函数有三个参数：
  （1）第1个是要查找的起始地址。
  （2）第2个是要查找的结束地址
  （3）第3个参数是要查找的值value
- 函数返回的是迭代器，第一个值等于value的元素位置；若未找到，返回end。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> number;
    number.insert(number.begin(),99);
    number.insert(number.begin(),98);
    number.insert(number.end(),97);
    number.insert(number.end(),92);
    number.insert(number.end(),90);
    vector<int>::iterator it;
    it=find(number.begin(), number.end(), 98);
    for (;it!=number.end();it++)
        cout<<*it<<endl;
    it=find(number.begin(), number.end(), 100);
    if(it == number.end())
        cout<<“Not find”<<endl;
    return 0;
}
```

（6）其他操作

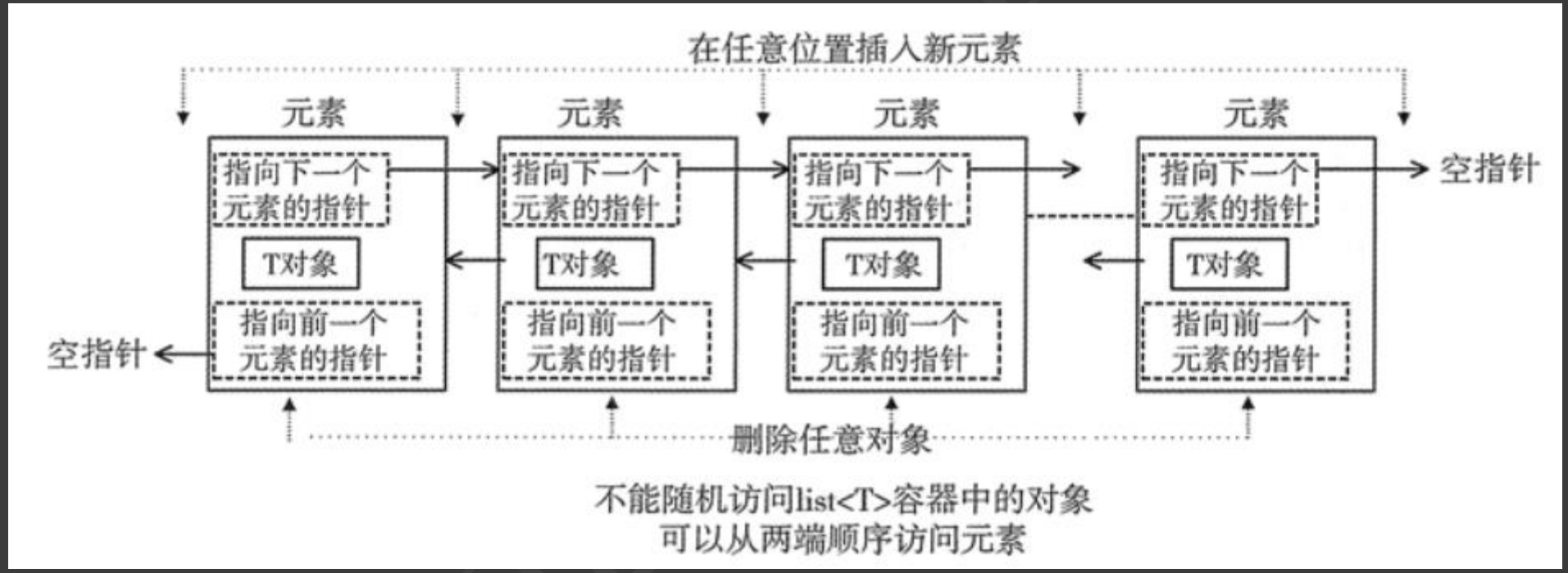
- for_each方法是依次遍历容器中的所有元素，并对每个元素执行相应的方法。
- for_each函数有三个参数：
  （1）第1个是要遍历的起始地址。
  （2）第2个是要遍历的结束地址
  （3）第3个参数是函数对象或者函数指针

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void fun(int n)
{
    cout<<"value is:"<<n<<endl;
}
int main()
{
    vector<int> number;
    number.insert(number.begin(),99);
    number.insert(number.begin(),98);
    number.insert(number.end(),97);
    number.insert(number.end(),92);
    number.insert(number.end(),90);
    for_each(number.begin(), number.end(), fun);
    return 0;
}
```

2.链表list

- 链表容器其内部数据结构实质是一个双向链表，可以在任何一端操作，与向量vector不同的是，链表容器必须进行顺序访问，不能实现随机访问，即不支持“[]”运算符，只能用对应迭代器操作元素。
- 同样，除了前面的内容介绍的顺序容器的共性外，链表还有自身的特殊操作。
- 必须包含的头文件#include <list>

2.链表list


在任意位置插入新元素
元素
元素
元素
元素
指向下一个元素的指针
T对象
指向前一个元素的指针
空指针
空指针
删除任意对象
不能随机访问list<T>容器中的对象
可以从两端顺序访问元素

2.链表list

- 优点：可以在序列已知的任何位置快速插入或删除元素（时间复杂度为O(1)），并且在 list 容器中移动元素，也比其它容器的效率高。
- 如果需要对序列进行大量添加或删除元素的操作，而直接访问元素的需求却很少，建议使用 list 。

（1）list对象的定义与初始化

（a）生成一个空的list容器：

```
list <std::string> names;
```

（b）初始化一个带有指定元素的列表：

```
list<std::string> names (20); // A list of 20 empty strings
```

（c）生成包含给定数量的相同元素列表：

```
list<double> values(50, 3.1415926);
```

50 个 double 型值的列表，并且每一个值都等于 3.1415926。

（d）list 具有拷贝构造函数，因此可以直接生成一个现有list的副本：

```
std::list<double> save_values(values);
```

（2）list常用函数

成员函数	功能
begin()	返回指向容器中第一个元素的双向迭代器。
end()	返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。
rbegin()	返回指向最后一个元素的反向双向迭代器。
rend()	返回指向第一个元素所在位置前一个位置的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
size()	返回当前容器实际包含的元素个数。
max_size()	返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$，所以我们很少会用到这个函数。

（2）list常用函数

成员函数	功能
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。
emplace_front()	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
push_front()	在容器头部插入一个元素。
pop_front()	删除容器头部的一个元素。
emplace_back()	在容器尾部直接生成一个元素。该函数和 push_back() 的功能相同，但效率更高。
push_back()	在容器尾部插入一个元素。
pop_back()	删除容器尾部的一个元素。
emplace()	在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。
insert()	在容器中的指定位置插入元素。
erase()	删除容器中一个或某区域内的元素。

（2）list常用函数

成员函数	功能
swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
resize()	调整容器的大小。
clear()	删除容器存储的所有元素。
splice()	将一个 list 容器中的元素插入到另一个容器的指定位置。
remove(val)	删除容器中所有等于 val 的元素。
remove_if()	删除容器中满足条件的元素。
unique()	删除容器中相邻的重复元素，只保留一个。
merge()	合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。
sort()	通过更改容器中元素的位置，将它们进行排序。
reverse()	反转容器中元素的顺序。

```
int main()
{
        list<int> values;
        values.push_back(0);
        values.push_back(40);
        values.push_back(20);
        values.push_back(20);
        values.push_back(40);
        values.push_back(60);
        values.push_back(80);
        values.sort();
        values.unique();
        list<int>::iterator lit = values.begin();
        list<int>::iterator end = values.end();
        for(;lit!=end;lit++)
            cout<<*lit<<endl;
        system("pause");
        return 0;
}
```

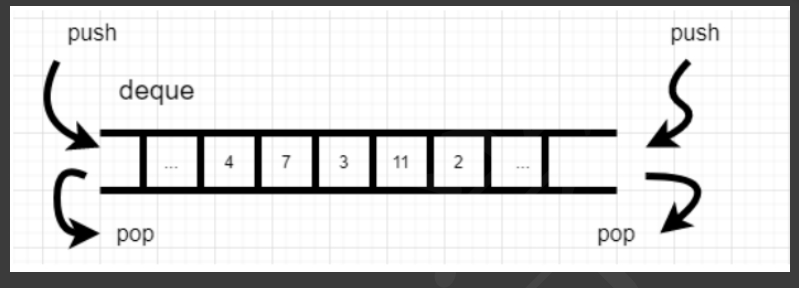
unique()函数之前必须排序，否则不能完成。

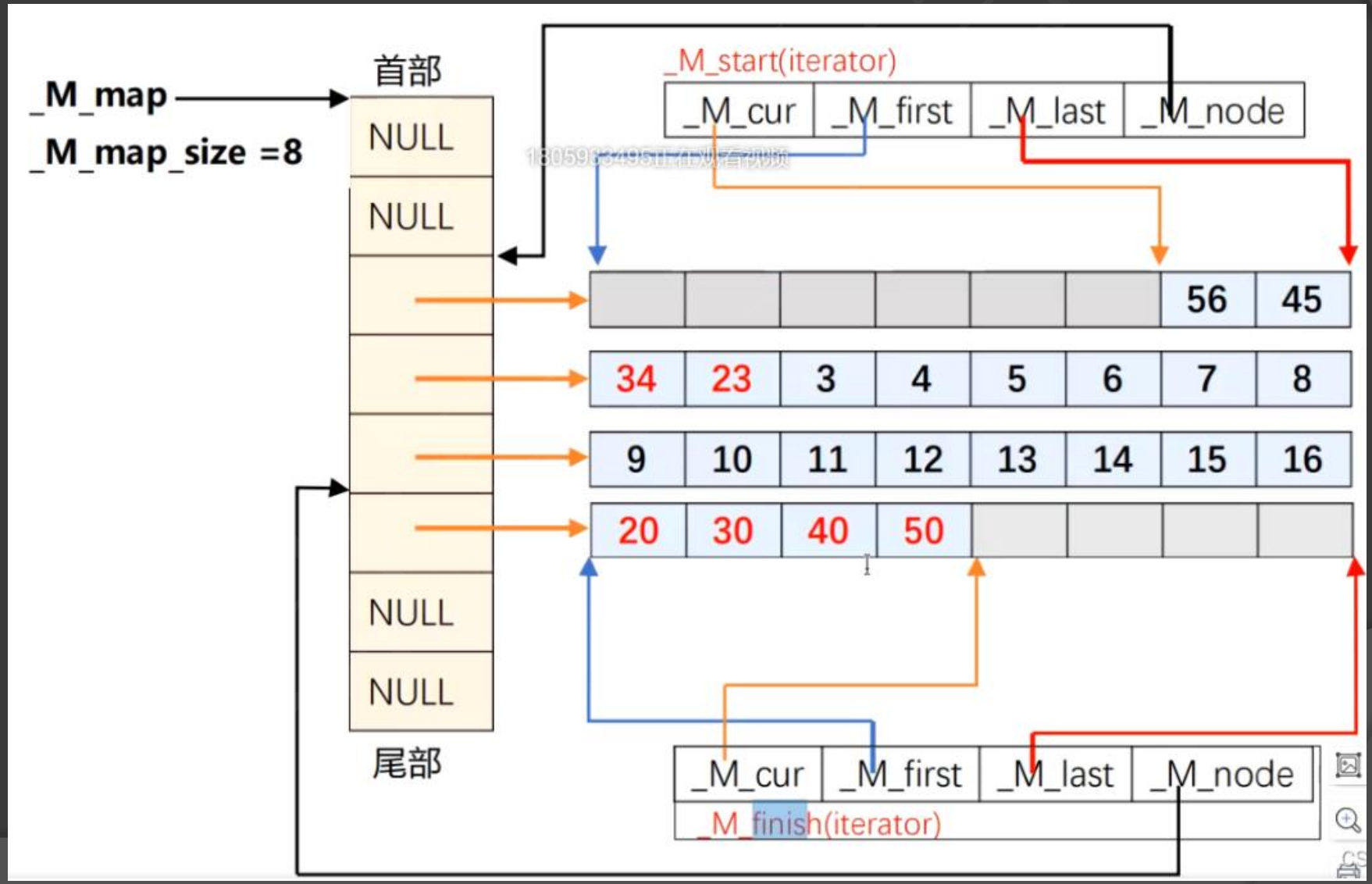
lit<end错误

3.双向队列deque

- 与容器vector相似，容器deque也是一种动态数组的形式，是一种访问形式比较自由的双端队列。
- 可以从队列的两端入队及出队（添加和删除）；
- 也可以使用运算符“[ ]”通过给定下标形式来访问队列中的元素，即可以顺序访问，也提供了随机访问的能力。
- 由于本身内部结构的特点，在队列两端添加和删除元素时，速度最快，效率较高，而在中间插入数据时比较费时，因为必须移动其他元素来实现容器的扩张。

3.双向队列deque


push
push
deque
...
4
7
3
11
2
...
pop
pop


首部
_M_map
_M_map_size =8
NULL
NULL
NULL
NULL
尾部
_M_start(iterator)
_M_cur
_M_first
_M_last
_M_node
56
45
34
23
3
4
5
6
7
8
9
10
11
12
13
14
15
16
20
30
40
50
_M_cur
_M_first
_M_last
_M_node
_M_finish(iterator)

13.2.2 关联容器

关联容器

- 关联容器提供了基于关键词（Key）快速检索数据的能力。
- 它通过关键词（Key）作为标识把单一的数据记录组织到特定的结构（如tree）中。
- STL提供的关联容器有：集合（set）、多元集合（multiset）、映射（map）、多元映射（multimap）。

关联容器

- set和map支持唯一关键词（unique key），就是每个Key，最多只保存一个元素（数据记录）。
- multiset和multimap则支持相同关键词（equal key），这样可有很多个元素用同一个Key进行存储。
- set（multiset）和map（multimap）之间的区别在于set（multiset）中的存储数据内含了Key表达式；而map（multimap）则将Key表达式和对应的数据分开存放。

1 集合set

- set，顾名思义，就是数学上的集合——每个元素最多只出现一次，并且set中的元素已经从小到大排好序。
- set 是以RB-tree（一种平衡二叉搜索树）作为底层实现机制。
- 若集合中元素类型为int、double、string等会自动（默认是升序）排序（使用平衡二叉树来实现），若集合中元素使用自定义类型而未重载关系运算符“<”就不能自动排序了。
- set容器不支持随机访问。

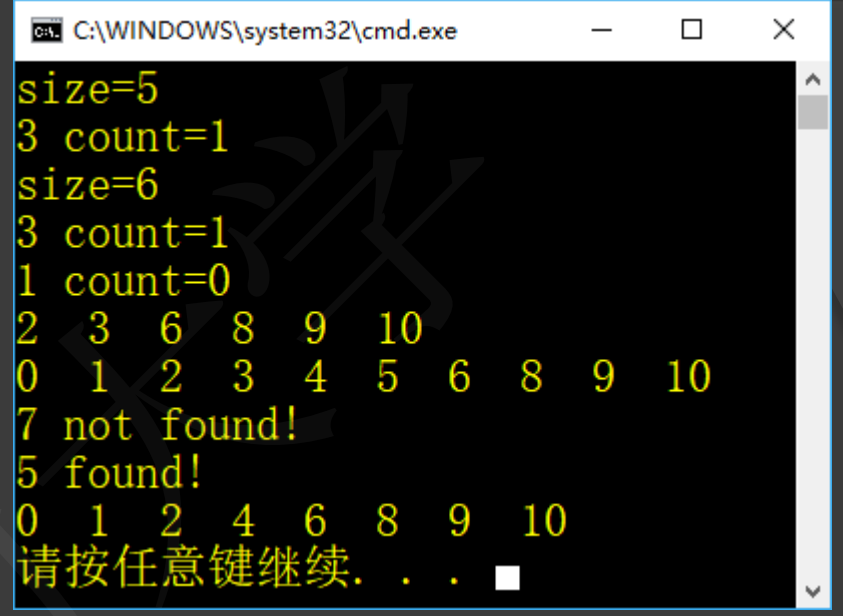
常用操作

begin()　返回set容器的第一个元素的地址
end()　返回set容器的最后一个元素地址
clear()　删除set容器中的所有的元素
empty()　判断set容器是否为空
max_size()　返回set容器可能包含的元素最大个数
size()　返回当前set容器中的元素个数
erase(it)　删除迭代器指针it处元素
insert(a)　插入某个元素
count()　查找set中某个元素出现的次数
find()　查找set中某个元素出现的位置。如果找到，就返回这个元素的迭代器，如果这个元素不存在，则返回 s.end()，即最后一个元素的下一个位置。

例 集合容器set的使用示例。

```
#include <set>
#include <iostream>
using namespace std;
int main()
{
  int ia[5] = {8,6,2,9,3};
  int ib[6]={3,4,8,5,0,1};
  set<int> iset(ia, ia+5);
  set<int>::iterator ite1;
  cout << "size=" << iset.size() << endl;                  // size=5
  cout << "3 count=" << iset.count(3) << endl;             // 3 count=1
  iset.insert(3);                                          // 插入元素 3
  cout << "size=" << iset.size() << endl;                  // size=5
  cout << "3 count=" << iset.count(3) << endl;             // 3 count=1
  cout << "1 count=" << iset.count(1) << endl;             // 1 count=0
  for(ite1=iset.begin(); ite1 != iset.end(); ++ite1)
     cout << *ite1<<"  ";                                   // 2  3  6  8  9
  cout << endl;
```

```
iset.insert(ib,ib+6);
for(ite1=iset.begin(); ite1 != iset.end(); ++ite1)
    cout << *ite1<<"  ";
 cout << endl;
 ite1 = iset.find(7);
 if (ite1 == iset.end())
    cout << "7 not found!" << endl;
 else
     cout<<" 7 found!"<<endl;
 ite1 = iset.find(5);
 if (ite1 !=  iset.end())
    cout << "5 found!" << endl;                    // 5 found!
 else
     cout<<" 5 not found!"<<endl;
 iset.erase(ite1);                                 // 删除元素 5
 iset.erase(3);                                    // 删除元素 3
 for(ite1=iset.begin(); ite1 != iset.end(); ++ite1)
    cout << *ite1<<"  ";                           // 0  1  2  4  6  8  9
 cout << endl;
 return 0;
}
```



注意

- 不要通过set 的迭代器改变set的相应元素值。
- 因为set 元素值就是其键值，关系到set 元素的排列规则。如果用迭代器随意改变set 元素值，会严重破坏set的组织。
- 在STL标准中，set<T>::iterator 被定义为底层RB-tree 的const_iterator，拒绝写入操作。

2 map

- map 是具有唯一键值对的容器，通常使用红黑树实现。
- 一个map是一个键值对序列，即（key ,value）对。它提供基于key的快速检索能力，在一个map中key值是唯一的，map提供双向迭代器。
- map要求能对key进行<操作，且保持按key值递增有序。
- map容器不支持随机访问。

map的使用示例。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
        map<int,string> person;
        person[100] = "tom";
        person.insert(pair<int, string>(101, "sally"));
        cout<<person[100]<<endl;

        map<int,string>::iterator mit = person.begin();
        for(;mit!=person.end();mit++)
                cout<<mit->first<<","<<mit->second<<endl;
}
```

vector与set(map)的对比

- map和set的插入删除效率比用其他序列容器高。因为对于关联容器来说，不需要做内存拷贝和内存移动。
- 每次insert之后，以前保存的iterator不会失效。iterator相当于指向节点的指针，内存没有变，指向内存的指针不会失效。
- 当数据元素增多时，set的搜索速度仍然很高效。在set中查找是使用二分查找。

13.3 STL算法

- 算法是STL的一个重要组成部分，包含了大约70个通用算法，这些算法主要通过迭代器对容器类进行操作，并可以在用户自定义的函数对象下进行工作。

- STL算法部分主要由头文件<algorithm>、<numeric>组成。要使用STL中的算法函数必须包含头文件<algorithm>，对于数值算法须包含<numeric>。

STL中的算法分成四组

STL中的算法分成四组：

- 非变异算法（Non-mutating Algorithms）
- 变异算法（Mutating Algorithms）
- 排序及相关操作（Sorting and Related Operations）
- 常用的数值算法（Generalized Numeric Algorithms）。

非变异算法指不直接修改其所操作容器内容的算法。包括for_each、find等函数模板。

STL中的算法分成四组

STL中的算法分成四组：

- 非变异算法（Non-mutating Algorithms）
- 变异算法（Mutating Algorithms）
- 排序及相关操作（Sorting and Related Operations）
- 常用的数值算法（Generalized Numeric Algorithms）。

变异算法指可以修改它们所操作的容器内容的算法。包括copy、transform、replace、remove、reverse等函数模板。

STL中的算法分成四组

STL中的算法分成四组：

- 非变异算法（Non-mutating Algorithms）
- 变异算法（Mutating Algorithms）
- 排序及相关操作（Sorting and Related Operations）
- 常用的数值算法（Generalized Numeric Algorithms）。

排序算法主要包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作。包括sort、binary_search、merge等函数模板。

STL中的算法分成四组

STL中的算法分成四组：

- 非变异算法（Non-mutating Algorithms）
- 变异算法（Mutating Algorithms）
- 排序及相关操作（Sorting and Related Operations）
- 常用的数值算法（Generalized Numeric Algorithms）。

常用的数值算法主要对容器内容进行数值计算。包括iota、accumulate、inner_product、partial_sum、adjacent_difference、power等函数模板。

(1) for_each

for_each 算法的函数模板定义如下：

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each (InputIterator first, InputIterator last,
UnaryFunction f)
{
    while (first != last) f(*first++);
    return f;
}
```

算法使用时有三个参数：两个输入迭代器，一个函数对象。其功能是对由迭代器指定的范围 [first, last) 中的每个元素中调用函数f。和for_each类似，所有STL算法的前两个参数都是一对迭代器 [first，last)，用来指出容器内一个范围内的元素。

(2) count和count_if

- STL的通用算法count()和count_if()用来给容器中的对象记数。

- count算法的函数模板的声明如下：

```
template <class InputIterator, class EqualityComparable>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const EqualityComparable&
value);
```

- 其功能是返回容器中从first处开始到last处结束范围内与value匹配的元素的个数。

(3) count_if算法

- count_if算法的函数模板的声明如下：

```
template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

- 其功能是返回容器中从first处开始到last处结束范围内使一元谓词pred返回true的元素个数。

(4) copy和remove

- STL算法copy将一个容器内的数据复制到另一个容器内，它的原型是：

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
InputIterator last, OutputIterator result);
```

- 其功能是把指定范围[first,last)内的元素复制到区间[result,result+(last-first)-1]中。函数返回值为result + (last - first)。

设有：

```
int a[5]={1,2,3,4,5};
vector<int> v1(a,a+5),v2(a,a+3),v3(5);
```

copy(v1.begin(), v1.end(), v3.begin());

// 将容器v1中的5个元素复制到v3中，执行后，

// v3中5个元素的值为1、2、3、4、5。

copy(v2.begin(), v2.end(), v1.begin()+1);

// 将容器v2中的3个元素复制到v1中第1个元素的后面，执行后，v1中5个元素的值为1、1、2、3、5。从这个结果可以看出，copy操作是复制，不是插入。

[问题]

```
copy(v1.begin(), v1.end(), v2.begin());
// 在运行时就会出错
// 因为容器v2只有3个元素，iv1中的5个元素复制过来，v2的空间不够，因此出错。
```

[解决办法]

一是预先给目的容器分配比要复制的元素大的空间；

例如：

```
v2.resize(v1.size());
copy(v1.begin(), v1.end(), v2.begin());
```

二是使用插入迭代器（是一种迭代器适配器）。

(5) Remove算法

Remove系列的算法有4个：remove、remove_if、remove_copy和remove_copy_if，它们的原型和功能分别是：

template <class ForwardIterator, class T>

ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);

其功能是从指定范围[first,last）中删除值等于value的元素，函数返回值为指向剩余元素结尾的迭代器。

(5) Remove算法

template <class ForwardIterator, class Predicate>

ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate pred);

其功能是从指定范围[first,last）中删除使谓词pred返回true的元素，函数返回值为指向剩余元素结尾的迭代器。

template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, const T& value);

其功能是从指定范围[first,last）中复制值不等于value的元素（即将值等于value的元素删除掉），并将结果放入result所指向的序列中。函数返回指向结果序列结尾的迭代器。

template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);

其功能是从指定范围[first,last）中复制使谓词pred返回false的元素（即将使谓词pred返回true的元素删除掉），并将结果放入result所指向的序列中。函数返回指向结果序列结尾的迭代器。

13.4 仿函数（函数对象）

- 利用C++标准模板库的算法可以为程序员减轻许多负担，但这些算法大都需要函数或仿函数作为参数。
- 使用count_if算法对及格和不及格的学生人数进行统计，函数为：
  - bool passed_test(int n)　// 判断一个成绩是否 >=60，通过了考试
  - bool failed_test(int n)　// 判断一个成绩是否 <60，不及格
- 然后，采用算法“count_if(vecScore.begin(), vecScore.end(), passed_test) ;”统计成绩大于等于60的学生人数。

这种采用普通函数给出count_if算法的统计条件的实现方法缺乏灵活性。
能不能进一步抽象，调用
　count_if(vecScore.begin(), vecScore.end(),pred(n))
就能统计出学生成绩大于或等于n的个数呢？
函数对象就提供了这样一种机制。

13.4.1 什么是函数对象

- 函数对象就是一些使用起来像调用函数一样的对象。
- 从实现的角度来看，函数对象是一种重载了operator()的class 或 class template。
- 定义和使用仿函数：先声明一个普通的类并重载“()”操作符：

```
class Negate
{
public:
    int operator() (int n) { return -n;}
};
```

在Negate类中重载的“()”操作符只有一个参数n，返回值类型与参数类型相同，均为int。函数返回与参数n符号相反的整数。

仿函数的使用

再写一小段主函数来测试仿函数：

```
int val;
Negate neg;
val=neg(10);
cout<<val<<endl;
```

程序输出结果为-10。

通过这个简单的例子可以看出，函数对象就是一个重载了“()”运算符的对象，它可以像一个函数一样使用。

在语句“val = neg(n);”中neg是对象，而不是函数。
编译器将语句“val = neg(n);”转化为
“val = neg.operator()(n);”。

编写仿函数类

为了统计出成绩大于或等于n的学生人数，可以编写仿函数类Pred如下：

```
class Pred
{
  public:
   Pred(int val):m_val(val){}
   bool operator()(int val)   {
          return val>=m_val;
   }
 private:
  int m_val;
};
```

于是，要统计成绩大于等于90的学生的人数个数，就可以写成

```
count_if(vecScore.begin(), vecScore.end(), Pred(90))
```

Pred(90)构造一个仿函数，count_if把这个对象依次应用到容器中的每一个元素，只要它返回一个真值，计数器就加1。

定义类模板

- 再进一步，可以利用模板技术，将重载的操作符“()”定义为类成员模板，以便函数对象适用于任何数据类型，并可应用到各种类型的容器上。

定义类模板

```
template<typename T>
class Pred
{
public:
   Pred(T val):m_val(val){}
   bool operator()(T val){
          return val>=m_val;
   }
private:
  T m_val;
};
```

这样，要统计成绩大于等于90的学生的人数个数，就写成“count_if(vecScore.begin(), vecScore.end(), Pred<int>(90)) ;”。

如果容器中元素是double类型的，只需把尖括号里的int换成double即可。

13.4.2 STL中的仿函数

- STL中预定义了三类函数对象供用户直接使用，它们是
  - 算术仿函数
  - 关系仿函数
  - 逻辑仿函数
- STL中定义的仿函数可以直接使用，必须包含头文件<functional>。

（1）算术仿函数

- STL 中定义的“算术仿函数”支持加法、减法、乘法、除法、取模（余数）和相反数运算。
  - 加法：plus<T>
  - 减法：minus<T>
  - 乘法：multiplies<T>
  - 除法：divides<T>
  - 取模：modulus<T>
  - 相反数：negate<T>

例如

```
plus<int> plusobj;
minus<int> minusobj;
// 以下运用上述对象，履行函数功能
cout << plusobj(3,8) << endl;        // 11
cout << minusobj(3,8) << endl;       // -5

// 以下直接以仿函数的临时对象履行函数功能
cout << plus<int>()(3,8) << endl;    // 11
cout << minus<int>()(3,8) << endl;   // -5
```

仿函数的主要用途

仿函数对象的主要用途是为了搭配STL算法。

例如，设有

vector<int> v1；

如果要计算v1中的所有元素的连乘积，可以采用STL算法accumulate并应用仿函数multiplies

语句为：

accumulate(v1.begin(),v1.end(), 1, multiplies<int>());

（2）关系仿函数

- STL中定义的“关系仿函数”支持等于、不等于、大于、大于等于、小于、小于等于六种运算。每一个都是二元仿函数。
  - 等于（Equality）：　equal_to<T>
  - 不等于（Inequality）：not_equal_to<T>
  - 大于（Greater than）：greater<T>
  - 大于或等于（Greater than or equal）：greater_equal<T>
  - 小于（Less than）：　less<T>
  - 小于或等于（Less than or equal）：less_equal<T>

关系仿函数的使用

关系仿函数通常用于和排序类算法搭配使用。

例如：

```
vector <int> v1;
//..填充向量
sort(v1.begin(), v1.end(), greater<int>() ); // 降序
sort(v1.begin(), v1.end(), less<int>() );    // 升序
```

（3）逻辑仿函数

- STL中定义的“逻辑仿函数”支持逻辑运算中的 And、Or、Not 三种运算，其中 And 和Or 为二元函数对象，Not 为一元函数对象。
  - 逻辑运算 And： logical_and<T>
  - 逻辑运算 Or： logical_or<T>
  - 逻辑运算 Not： logical_not<T>

仿函数的优点

1、仿函数可以有自己的状态信息，一个仿函数在多次调用时可以共享这个状态，而函数调用没有。

```
class Generator
{
        int m_nNumber;
public:
        Generator(int n)
        {
                m_nNumber = n;
        }
        void operator()(int& r)
        {
                r = m_nNumber++;
        }
};
```

```
int main()
{
        vector<int> v1;
        v1.insert(v1.end(), 1);
        v1.insert(v1.end(), 4);
        v1.insert(v1.end(), 2);
        v1.insert(v1.end(), 3);
        v1.insert(v1.end(), 2);
for_each(v1.begin(), v1.end(),Generator(4));
}
```

依次为v1中的元素赋值为4，5，6，7，8。

仿函数的优点

2、仿函数有自己特有的类型，而普通函数没有。在使用STL的容器时可以将函数对象的类型传递给容器作为参数来实例化相应的模板，从而来定制自己的算法，如排序算法。

```
class charSort
{
public:
        bool operator()(const char&
c1, const char& c2) const
        {
                return c1 > c2;
        }
};
```

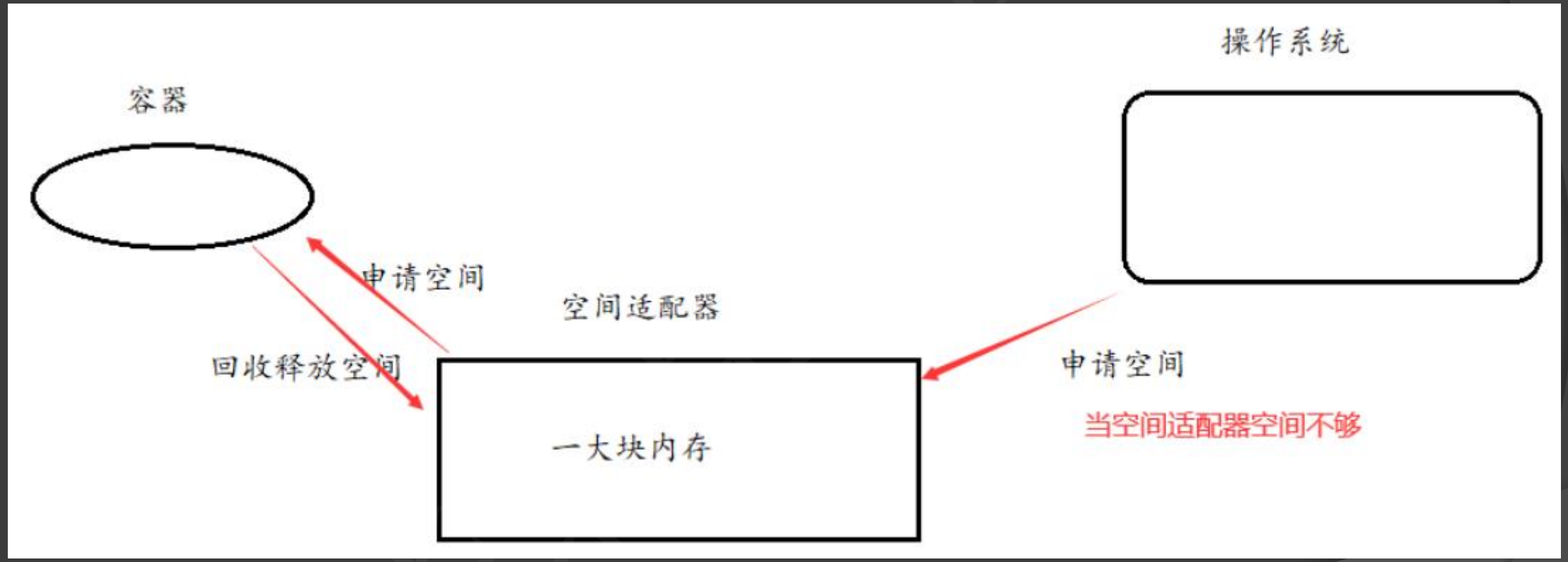
```
int main()
{
        set<char, charSort> sc;
        sc.insert('B');
        sc.insert('A');
        sc.insert('C');
        set<char,charSort>::iterator sit;
        for(sit=sc.begin();sit!=sc.end();sit++)
                cout<<*sit<<endl;
}
```

sc中的排序不再是默认的由小到大

13.5 空间配置器

- STL容器在不断保存数据时，当保存的的数据个数超过容器容量时，需要进行扩容。
- 但是，当不断保存数据时，就可能需要不断的进行扩容。此时，扩容需要不断的向操作系统申请空间，释放空间。操作系统是很繁忙的，这样会大大影响操作系统的效率。
- 空间配置器：是操作系统开辟的一大段内存空间。STL需要扩容申请内存时，就从空间配置器中申请，不需要再经过操作系统。并且，它还能回收释放的空间，供下一次使用。

13.5 空间配置器



13.6 适配器

- 适配器本身是一个新的自定义类型 class/struct，其中会包含一个或多个辅助的所适配的类型的成员，并对内含的成员的接口进行改造，再以新的适配器类型向外部提供接口。
- 适配器可以对容器、迭代器和函数对象进行适配，进而产生了多种适配器。
- STL 提供了 3 种容器适配器，分别为 stack 栈适配器、queue 队列适配器以及 priority_queue 优先权队列适配器。其中，各适配器所使用的默认基础容器以及可供用户选择的基础容器

13.6 适配器

容器适配器	基础容器筛选条件	默认使用的基础容器
stack	基础容器需包含以下成员函数: • empty() • size() • back() • push_back() • pop_back() 满足条件的基础容器有 vector、deque、list。	deque
queue	基础容器需包含以下成员函数: • empty() • size() • front() • back() • push_back() • pop_front() 满足条件的基础容器有 deque、list。	deque
priority_queue	基础容器需包含以下成员函数: • empty() • size() • front() • push_back() • pop_back() 满足条件的基础容器有vector、deque。	vector