程序设计与算法（三）

C++面向对象程序设计

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

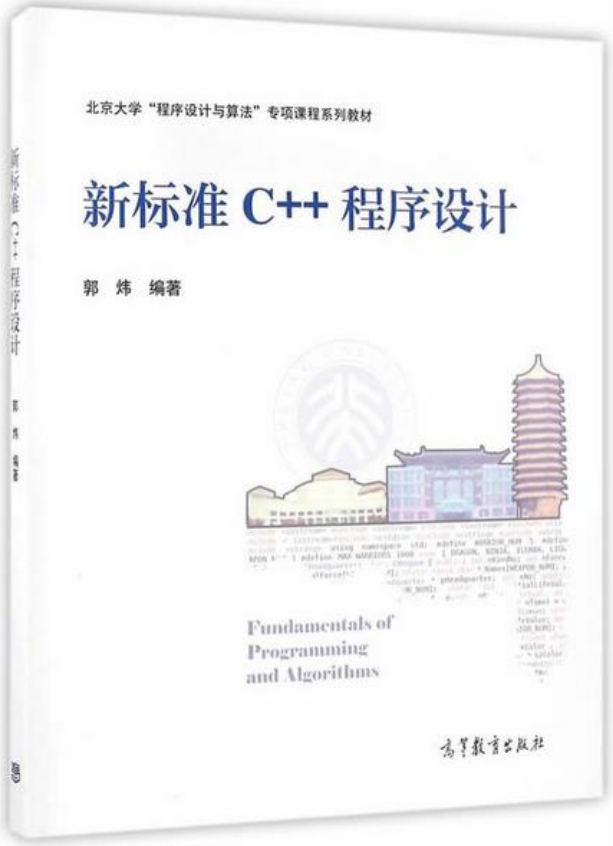
讲义照片均为郭炜拍摄

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著



标准模板库STL

(一)

STL基本概念

冰岛黛提瀑布

泛型程序设计

C++ 语言的核心优势之一就是便于软件的重用

C++中有两个方面体现重用：

1.面向对象的思想：继承和多态，标准类库

2.泛型程序设计(generic programming) 的思想： 模板机制，以及标准模板库 STL

泛型程序设计

简单地说就是使用模板的程序设计法。

将一些常用的数据结构（比如链表，数组，二叉树）和算法（比如排序，查找）写成模板，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。

标准模板库 (Standard Template Library) 就是一些常用数据结构和算法的模板的集合。

有了STL，不必再写大多的标准数据结构和算法，并且可获得非常高的性能。

STL中的基本的概念

容器：可容纳各种数据类型的通用数据结构,是类模板

迭代器：可用于依次存取容器中元素，类似于指针

算法：用来操作容器中的元素的函数模板

- sort()来对一个vector中的数据进行排序
- find()来搜索一个list中的对象

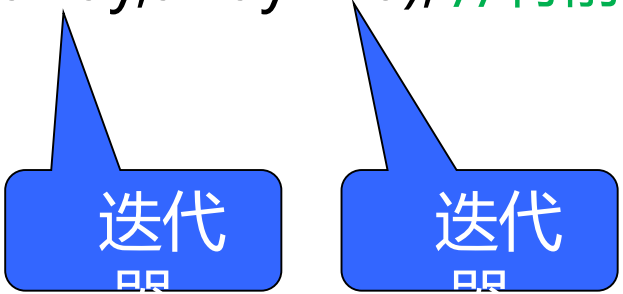
算法本身与他们操作的数据的类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

STL中的基本的概念

int array[100];

该数组就是容器，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



容器概述

张家界市天门山

容器概述

可以用于存放各种类型的数据（基本类型的变量，对象等）的数据结构，都是类模版,分为三种：

1)顺序容器

vector, deque,list

2)关联容器

set, multiset, map, multimap

3)容器适配器

stack, queue, priority_queue

容器概述

对象被插入容器中时，被插入的是对象的一个复制品。许多算法，比如排序，查找，要求对容器中的元素进行比较，有的容器本身就是排序的，所以，放入容器的对象所属的类，往往还应该重载 == 和 < 运算符。

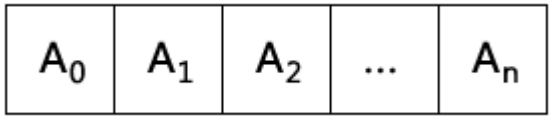
顺序容器简介

容器并非排序的，元素的插入位置同元素的值无关。

有vector,deque,list 三种

- vector 头文件 <vector>

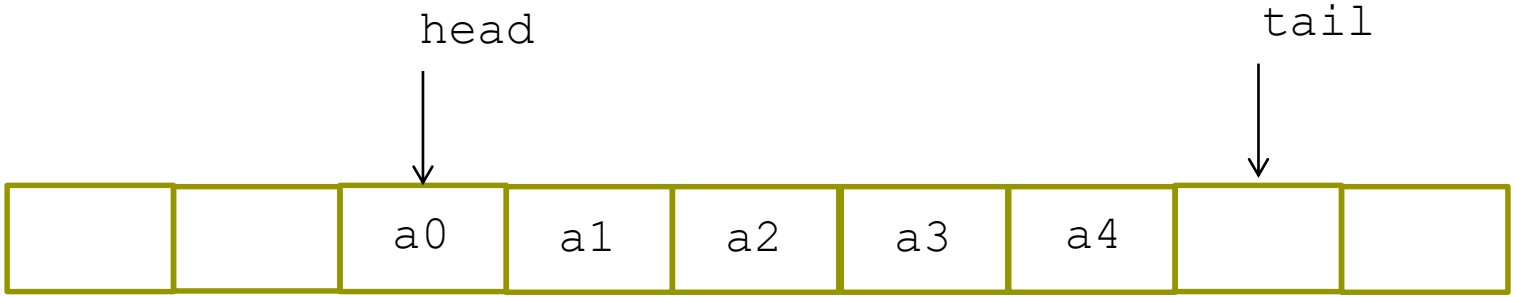
动态数组。元素在内存连续存放。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能(大部分情况下是常数时间）。



顺序容器简介

- deque 头文件 <deque>

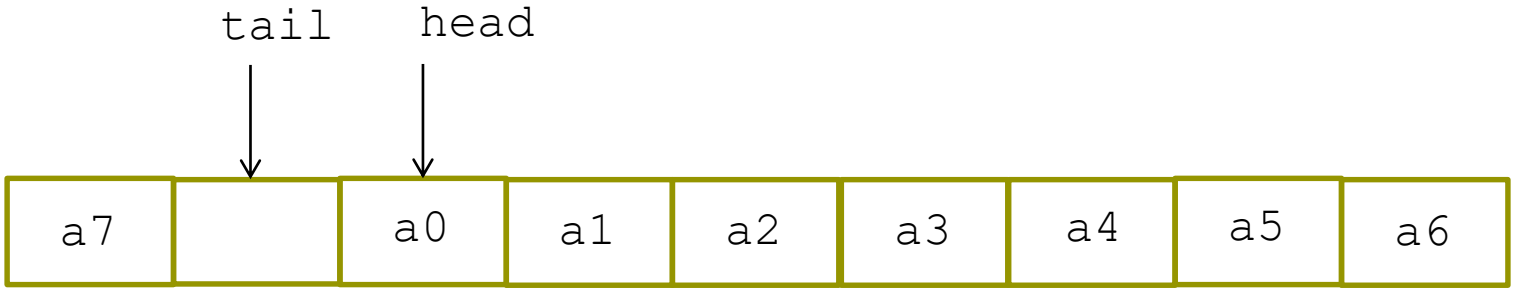
双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能(大部分情况下是常数时间）。



顺序容器简介

- deque 头文件 <deque>

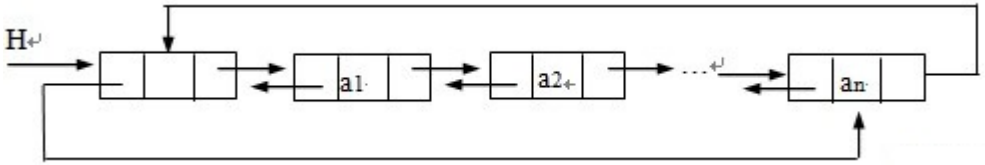
双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能(大部分情况下是常数时间）。



顺序容器简介

- list　头文件 <list>

双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。不支持随机存取。



关联容器简介

- 元素是排序的
- 插入任何元素，都按相应的排序规则来确定其位置
- 在查找时具有非常好的性能
- 通常以平衡二叉树方式实现，插入和检索的时间都是 O(log(N))

- set/multiset　头文件 <set>

set 即集合。set中不允许相同元素，multiset中允许存在相同的元素。

- map/multimap　头文件 <map>

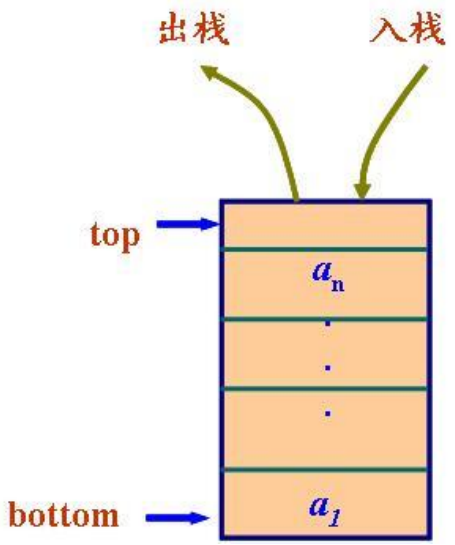
map与set的不同在于map中存放的元素有且仅有两个成员变量，一个名为first,另一个名为second, map根据first值对元素进行从小到大排序，并可快速地根据first来检索元素。

map同multimap的不同在于是否允许相同first值的元素。

容器适配器简介

- stack :头文件 <stack>

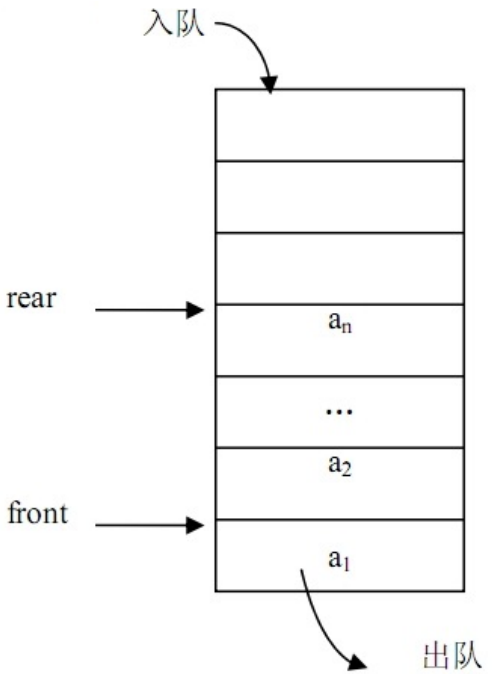
  栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项（栈顶的项）。后进先出。



容器适配器简介

- queue 头文件 <queue>

  队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。先进先出。



容器适配器简介

- priority_queue 头文件 <queue>

  优先级队列。最高优先级元素总是第一个出列

顺序容器和关联容器中都有的成员函数

begin　返回指向容器中第一个元素的迭代器

end　返回指向容器中最后一个元素后面的位置的迭代器

rbegin　返回指向容器中最后一个元素的迭代器

rend　返回指向容器中第一个元素前面的位置的迭代器

erase　从容器中删除一个或几个元素

clear　从容器中删除所有元素

顺序容器的常用成员函数

front :返回容器中第一个元素的引用

back : 返回容器中最后一个元素的引用

push_back : 在容器末尾增加新元素

pop_back : 删除容器末尾的元素

erase :删除迭代器指向的元素(可能会使该迭代器失效），或删除一个区间，返回被删除元素后面的那个元素的迭代器

迭代器

张家界

迭代器

- 用于指向顺序容器和关联容器中的元素
- 迭代器用法和指针类似
- 有const 和非 const两种
- 通过迭代器可以读取它指向的元素
- 通过非const迭代器还能修改其指向的元素

迭代器

定义一个容器类的迭代器的方法可以是：

容器类名::iterator　变量名;

或：

容器类名::const_iterator 变量名;

访问一个迭代器指向的元素：

* 迭代器变量名

迭代器

迭代器上可以执行 ++ 操作, 以使其指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，此时再使用它，就会出错，类似于使用NULL或未初始化的指针一样。

迭代器示例

```
#include <vector>
#include <iostream>
using namespace std;

int main()  {
    vector<int> v; //一个存放int元素的数组，一开始里面没有元素
    v.push_back(1);  v.push_back(2);   v.push_back(3);
    v.push_back(4);
    vector<int>::const_iterator i;   //常量迭代器
    for( i = v.begin();i != v.end();++i )
        cout << * i << ",";
    cout << endl;
```

输出结果：

```
1,2,3,4,
4,3,2,1,
100,100,100,100,
```

```
    vector<int>::reverse_iterator r;   //反向迭代器
    for( r = v.rbegin();r != v.rend();r++ )
        cout << * r << ",";
    cout << endl;
    vector<int>::iterator j;    //非常量迭代器
    for( j = v.begin();j != v.end();j ++ )
        * j =  100;
    for( i = v.begin();i != v.end();i++ )
        cout << * i << ",";
}
```

输出结果:

```
1,2,3,4,
4,3,2,1,
100,100,100,100,
```

双向迭代器

若p和p1都是双向迭代器，则可对p、p1可进行以下操作：

++p, p++	使p指向容器中下一个元素
--p, p--	使p指向容器中上一个元素
* p	取p指向的元素
p = p1	赋值
p == p1 , p!= p1	判断是否相等、不等

随机访问迭代器

若p和p1都是随机访问迭代器，则可对p、p1可进行以下操作：

- 双向迭代器的所有操作
- p += i 将p向后移动i个元素
- p -= i 将p向向前移动i个元素
- p + i 值为: 指向 p 后面的第i个元素的迭代器
- p - i 值为: 指向 p 前面的第i个元素的迭代器
- p[i] 值为: p后面的第i个元素的引用
- p < p1, p <= p1, p > p1, p>= p1
- p – p1 : p1和p之间的元素个数

容器	容器上的迭代器类别
vector	随机访问
deque	随机访问
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

容器	容器上的迭代器类别
vector	随机访问
deque	随机访问
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

有的算法，例如sort，binary_search需要通过随机访问迭代器来访问容器中的元素，那么list以及关联容器就不支持该算法！

vector的迭代器是随机迭代器，

遍历 vector 可以有以下几种做法(deque亦然)：

```
vector<int> v(100);
int i;
for(i = 0;i < v.size() ; i ++)
        cout << v[i]; //根据下标随机访问
vector<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();++ii)
        cout << * ii;
for( ii = v.begin(); ii < v.end ();++ii )
        cout << * ii;
```

```
//间隔一个输出：
ii = v.begin();
while( ii < v.end()) {
        cout << * ii;
        ii = ii + 2;
}
```

list 的迭代器是双向迭代器，

正确的遍历list的方法：

```
list<int> v;
list<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();++ii )
        cout << * ii;
```

错误的做法：

```
for( ii = v.begin(); ii < v.end ();++ii )
        cout << * ii;
//双向迭代器不支持 <,list没有 [] 成员函数
for(int i = 0;i < v.size() ; i ++)
        cout << v[i];
```

算法简介

法国普罗旺斯路边

算法简介

- 算法就是一个个函数模板，大多数在<algorithm> 中定义
- STL中提供能在各种容器中通用的算法，比如查找，排序等
- 算法通过迭代器来操纵容器中的元素。许多算法可以对容器中的一个局部区间进行操作，因此需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器。比如，排序和查找
- 有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器
- 算法可以处理容器，也可以处理普通数组

算法示例：find()

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

- first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点[first,last)。区间的起点是位于查找范围之中的，而终点不是。find在[first,last)查找等于val的元素
- 用 == 运算符判断相等
- 函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。如果找不到，则该迭代器等于last

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()  {    //find算法示例
        int array[10] = {10,20,30,40};
        vector<int> v;
        v.push_back(1);      v.push_back(2);
        v.push_back(3);      v.push_back(4);
        vector<int>::iterator p;
        p = find(v.begin(),v.end(),3);
        if( p != v.end())
                cout << * p << endl; //输出3
```

输出:

```
3
not found
3
20
```

```
    p = find(v.begin(),v.end(),9);
    if( p == v.end())
            cout << "not found " << endl;
    p = find(v.begin()+1,v.end()-2,1);
    //整个容器：[1,2,3,4]， 查找区间：[2,3)
    if( p != v.end())
            cout << * p << endl;
    int * pp = find( array,array+4,20);//数组名是迭代器
    cout << * pp << endl;
}
```

输出:

```
3
not found
3
20
```

STL中的“大”、“小”和“相等”

冰岛米湖Hverfjall 火山口

STL中“大” “小” 的概念

- 关联容器内部的元素是从小到大排序的
- 有些算法要求其操作的区间是从小到大排序的，称为“有序区间算法”
  例：binary_search
- 有些算法会对区间进行从小到大排序，称为“排序算法”
  例： sort
- 还有一些其他算法会用到“大”，“小”的概念

使用STL时，在缺省的情况下，以下三个说法等价：

1) x比y小
2) 表达式“x<y”为真
3) y比x大

STL中“相等”的概念

- 有时，“x和y相等”等价于“x==y为真”

  例：在未排序的区间上进行的算法，如顺序查找find

  ……

- 有时“x和y相等”等价于“x小于y和y小于x同时为假”

  例：

  有序区间算法，如binary_search

  关联容器自身的成员函数find

  ……

STL中“相等” 概念演示

```
#include <iostream>
#include <algorithm>
using namespace std;
class A  {
    int v;
    public:
        A(int n):v(n) { }
        bool operator < ( const A & a2) const       {
                //必须为常量成员函数
                cout << v << "<" << a2.v << "?" << endl;
                return false;
        }
        bool operator ==(const A & a2) const       {
                 cout << v << "==" << a2.v << "?" << endl;
                 return v == a2.v;
        }
};
```

STL中“相等” 概念演示

```
int main()
{
        A a [] =
{ A(1),A(2),A(3),A(4),A(5) };
        cout << binary_search(a,a+4,A(9));
//折半查找
        return 0;
}
```

STL中“相等” 概念演示

```
int main() {
    A a [] = { A(1),A(2),A(3),A(4),A(5) };
    cout << binary_search(a,a+4,A(9));
    return 0;
}
```

输出结果:

```
3<9?
2<9?
1<9?
9<1?
1
```

vector和deque

冰岛雷克雅未克市中心

vector 示例程序

```
#include <iostream>
#include <vector>
using namespace std;
template<class T>
void PrintVector( T s, T e)
{
      for(; s != e; ++s)
            cout << * s << " ";
      cout << endl;
}
```

```
int main()  {
        int a[5] = { 1,2,3,4,5 };
        vector<int> v(a,a+5);  //将数组a的内容放入v
        cout << "1) " << v.end()  - v.begin() << endl;
        //两个随机迭代器可以相减，输出 1) 5
        cout << "2) "; PrintVector(v.begin(),v.end());
        //2) 1 2 3 4 5
        v.insert(v.begin() + 2, 13); //在begin()+2位置插入 13
        cout << "3) "; PrintVector(v.begin(),v.end());
        //3) 1 2 13 3 4 5
        v.erase(v.begin() + 2); //删除位于 begin() + 2的元素
        cout << "4) "; PrintVector(v.begin(),v.end());
        //4) 1 2 3 4 5
        vector<int> v2(4,100);  //v2 有4个元素，都是100
        v2.insert(v2.begin(),v.begin()+ 1,v.begin()+3);
        //将v的一段插入v2开头
        cout << "5) v2: "; PrintVector(v2.begin(),v2.end());
        //5) v2: 2 3 100 100 100 100
```

```
    v.erase(v.begin() + 1, v.begin() + 3);
    //删除 v 上的一个区间,即 2,3
    cout << "6) "; PrintVector(v.begin(),v.end());
    //6) 1 4 5
    return 0;
}
```

用vector实现二维数组

```
#include <iostream>
#include <vector>
using namespace std;
int main()  {
        vector<vector<int> > v(3);
        //v有3个元素，每个元素都是vector<int> 容器
        for(int i = 0;i < v.size(); ++i)
                for(int j = 0; j < 4; ++j)
                        v[i].push_back(j);
        for(int i = 0;i < v.size(); ++i) {
                for(int j = 0; j < v[i].size(); ++j)
                        cout << v[i][j] << " ";
                cout << endl;
        }
        return 0;
}
```

程序输出结果：

```
0 1 2 3
0 1 2 3
0 1 2 3
```

deque

- 所有适用于 vector的操作都适用于 deque。

deque还有 push front（将元素插入到前面） 和 pop_front（删除最前面的元素）操作，复杂度是O(1)

双向链表

list

挪威松恩峡湾

list 容器

- 在任何位置插入删除都是常数时间，不支持随机存取。
- 除了具有所有顺序容器都有的成员函数以外，还支持8个成员函数：

push_front: 在前面插入

pop_front: 删除前面的元素

sort: 排序 （list 不支持 STL 的算法 sort）

remove: 删除和指定值相等的所有元素

unique: 删除所有和前一个元素相同的元素（要做到元素不重复，则unique之前还需要 sort）

merge: 合并两个链表，并清空被合并的那个

reverse: 颠倒链表

splice: 在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素

```
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
 class A {
    private:
        int n;
    public:
        A( int n_ ) { n = n_; }
        friend bool operator<( const A & a1, const A & a2);
        friend bool operator==( const A & a1, const A & a2);
        friend ostream & operator <<(  ostream & o, const A & a);
};
```

```
bool operator<( const A & a1, const A & a2) {
    return a1.n < a2.n;
}
bool operator==( const A & a1, const A & a2) {
    return a1.n == a2.n;
}
ostream & operator <<(  ostream & o, const A & a) {
    o << a.n;
    return o;
}
```

```
template <class T>
void PrintList(const list<T> & lst) {
//不推荐的写法，还是用两个迭代器作为参数更好
    int tmp = lst.size();
    if( tmp > 0  ) {
        typename list<T>::const_iterator i;
        i = lst.begin();
        for( i = lst.begin();i != lst.end(); i ++)
                cout << * i << ",";
    }
}// typename用来说明 list<T>::const_iterator是个类型
//在vs中不写也可以
```

```
int main()    {
    list<A>  lst1,lst2;
    lst1.push_back(1);lst1.push_back(3);
    lst1.push_back(2);lst1.push_back(4);
    lst1.push_back(2);
    lst2.push_back(10);lst2.push_front(20);
    lst2.push_back(30);lst2.push_back(30);
    lst2.push_back(30);lst2.push_front(40);
    lst2.push_back(40);
    cout << "1) "; PrintList( lst1); cout << endl;
    // 1) 1,3,2,4,2,
    cout << "2) "; PrintList( lst2); cout << endl;
    // 2) 40,20,10,30,30,30,40,
```

```
lst2.sort();
cout << "3) "; PrintList( lst2); cout << endl;
//3) 10,20,30,30,30,40,40,
lst2.pop_front();
cout << "4) "; PrintList( lst2); cout << endl;
//4) 20,30,30,30,40,40,
lst1.remove(2); //删除所有和A(2)相等的元素
cout << "5) "; PrintList( lst1); cout << endl;
//5) 1,3,4,
lst2.unique();  //删除所有和前一个元素相等的元素
cout << "6) "; PrintList( lst2); cout << endl;
//6) 20,30,40,
```

```
lst1.merge (lst2);  //合并 lst2到lst1并清空lst2
cout << "7) "; PrintList( lst1); cout << endl;
//7) 1,3,4,20,30,40,
cout << "8) "; PrintList( lst2); cout << endl;
//8)
lst1.reverse();
cout << "9) "; PrintList( lst1); cout << endl;
//9) 40,30,20,4,3,1,
```

```
    lst2.push_back (100);lst2.push_back (200);
    lst2.push_back (300);lst2.push_back (400);
    list<A>::iterator p1,p2,p3;
    p1 = find(lst1.begin(),lst1.end(),3);
    p2 = find(lst2.begin(),lst2.end(),200);
    p3 = find(lst2.begin(),lst2.end(),400);
    lst1.splice(p1,lst2,p2, p3);
    //将[p2,p3)插入p1之前，并从lst2中删除[p2,p3)
    cout << "10) "; PrintList( lst1); cout << endl;
    //10) 40,30,20,4,200,300,3,1,
    cout << "11) "; PrintList( lst2); cout << endl;
    //11) 100,400,
    return 0;
}
```

函数对象

阳朔漓江

函数对象

- 是个对象，但是用起来看上去象函数调用，实际上也执行了函数调用。

```
class CMyAverage  {
public:
    double operator()( int a1, int a2, int a3 ) {
    //重载 () 运算符
        return (double)(a1 + a2+a3) / 3;
    }
};
 CMyAverage average;  //函数对象
 cout << average(3,2,3); // average.operator()(3,2,3) 用起来看
   上去象函数调用 输出 2.66667
```

函数对象的应用：

STL里有以下模板：

```
template<class InIt, class T, class Pred>
T accumulate(InIt first, InIt last, T val, Pred pr);
```

- pr 就是个函数对象。

  对[first,last)中的每个迭代器 I，

  执行 val = pr(val,* I) ，返回最终的val。
- Pr也可以是个函数。

Dev C++ 中的 Accumulate 源代码1:

```
 template<typename _InputIterator, typename _Tp>
    _Tp  accumulate(_InputIterator __first, _InputIterator
__last,
                    _Tp __init)
    {

      for ( ; __first != __last; ++__first)
       __init = __init + *__first;
      return __init;
    }

// typename 等效于class
```

Dev C++ 中的 Accumulate 源代码2:

```
template<typename _InputIterator, typename _Tp,
            typename _BinaryOperation>
   _Tp    accumulate(_InputIterator __first, _InputIterator __last,
        _Tp __init,     _BinaryOperation __binary_op)
   {
       for ( ; __first != __last; ++__first)
              __init = __binary_op(__init, *__first);
            return __init;
   }
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;
int sumSquares( int total, int value)
{   return total + value * value;    }

template <class T>
void PrintInterval(T first, T last)
{ //输出区间[first,last)中的元素
        for( ; first != last; ++ first)
                cout << * first << " ";
        cout << endl;
}
```

```
template<class T>
class SumPowers
{
        private:
                int power;
        public:
                SumPowers(int p):power(p) { }
                const T  operator() ( const T & total,
                                      const T & value)
                { //计算 value的power次方，加到total上
                        T v = value;
                        for( int i = 0;i < power - 1; ++ i)
                                v = v * value;
                        return total + v;
                }
};
```

```
int main()
{
        const int SIZE = 10;
        int a1[] = { 1,2,3,4,5,6,7,8,9,10 };
        vector<int> v(a1,a1+SIZE);
        cout << "1) "; PrintInterval(v.begin(),v.end());
        int result = accumulate(v.begin(),v.end(),0,SumSquares);
        cout << "2) 平方和: " << result << endl;
        result =
          accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
        cout << "3) 立方和: " << result << endl;
        result =
          accumulate(v.begin(),v.end(),0,SumPowers<int>(4));
        cout << "4) 4次方和: " << result;
        return 0;
}
```

```
int result =
accumulate(v.begin(),v.end(),0,SumSquares);
```

实例化出:

```
int accumulate(vector<int>::iterator
first,vector<int>::iterator last,
      int init,int ( * op)( int,int))
{
      for ( ; first != last; ++first)
            init = op(init, *first);
    return init;
}
```

```
accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
```

实例化出：

```
int accumulate(vector<int>::iterator first,
               vector<int>::iterator last,
       int init, SumPowers<int> op)
{
       for ( ; first != last; ++first)
              init = op(init, *first);
    return init;
}
```

输出：

1) 1 2 3 4 5 6 7 8 9 10
2) 平方和：385
3) 立方和：3025
4) 4次方和：25333

STL 的<functional> 里还有以下函数对象类模板：

equal_to

greater

less　　…….

这些模板可以用来生成函数对象。

greater 函数对象类模板

```
template<class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x > y;
    }
};
```

binary_function定义:

```
template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;     };
```

greater 的应用

- list 有两个sort函数，前面例子中看到的是不带参数的sort函数，它将list中的元素按 < 规定的比较方法 升序排列。

- list还有另一个sort函数：

```
template <class T2>
  void sort(T2 op);
```

可以用 op来比较大小，即 op(x,y) 为true则认为x应该排在前面。

```
#include <list>
#include <iostream>
#include <iterator>
using namespace std;
class MyLess {
public:
          bool  operator()( const int & c1, const int & c2 )
          {
                    return (c1 % 10) < (c2 % 10);
          }
};
```

```
int main()
{   const int SIZE = 5;
    int a[SIZE] = {5,21,14,2,3};
    list<int> lst(a,a+SIZE);
    lst.sort(MyLess());
    ostream_iterator<int> output(cout,",");
    copy( lst.begin(),lst.end(),output); cout << endl;
    lst.sort(greater<int>()); //greater<int>()是个对象
                              //本句进行降序排序
    copy( lst.begin(),lst.end(),output); cout << endl;

    return 0;
}
```

输出:

```
21,2,3,14,5,
21,14,5,3,2,
```

ostream_iterator

copy类似于

```
template <class T1,class T2>
void Copy(T1 s,T1 e, T2 x)
{
        for(; s != e; ++s,++x)
                *x = *s;
}
```

ostream_iterator 如何编写?

```
#include <iostream>
#include <iterator>
using namespace std;
class MyLess
{
  public:
     bool operator() (int a1,int a2)
     {
       if( ( a1 % 10 ) < (a2%10) )
           return true;
       else
           return false;
      }
};
bool MyCompare(int a1,int a2)
{
   if( ( a1 % 10 ) < (a2%10) )
       return false;
   else
        return true;
}
```

```
int main()
{
    int a[] = {35,7,13,19,12};
    cout << MyMax(a,5,MyLess()) << endl;
    cout << MyMax(a,5,MyCompare) << endl;
    return 0;
}
```

输出：

```
19
12
```

要求写出MyMax

```
template <class T, class Pred>
T MyMax( T * p, int n, Pred myless)
{
  T tmpmax = p[0];
  for( int i = 1;i < n;i ++ )
       if( myless(tmpmax,p[i]))
            tmpmax = p[i];
  return tmpmax;
};
```

引入函数对象后，STL中的“大”，“小”关系

关联容器和STL中许多算法，都是可以自定义比较器的。在自定义了比较器op的情况下，以下三种说法是等价的：

1) x小于y
2) op(x,y)返回值为true
3) y大于x

比较规则的注意事项

```
struct 结构名
{
        bool operator()( const T & a1,const T & a2) {
                //若a1应该在a2前面，则返回true。
                //否则返回false。
        }
};
```

排序规则返回 true，意味着 a1 必须在 a2 前面
返回 false，意味着 a1 并非必须在 a2 前面

排序规则的写法，不能造成比较 a1,a2 返回 true 比较 a2,a1 也返回 true

否则sort会 runtime error

比较 a1,a2 返回 false 比较 a2,a1 也返回 false，则没有问题