2018年春

程序设计实习(I): C++程序设计

第十讲 标准模板库2

刘家瑛

liujiaying@pku.edu.cn

课前多叽歪

- 抓紧本周上机机会
  - **4.14 (本周)** 上机
  - **4.21** 运动会**停**
  - **4.28** 模考
  - **5.05** 校庆放假 **停**
  - **5.13 期中考试**
  - **5.20 ACM校内赛**

课前多吼歪

- 助教期中前会给作业分数
- **关于POJ账号[今天内修改!]**
  - 修改昵称为学号+姓名
  - 添加进相应的助教组

4.3 deque 容器

- 所有适用于 vector的操作都适用于 deque
- 比vector的优点: **头部删除/添加元素**性能也很好
- deque还包含以下操作:
  - **push_front:**将元素插入到前面
  - **pop_front:** 删除最前面的元素

顺序容器：小结

- **vector:** 强调通过**随机访问**进行快速访问
  - 插入/删除：非尾处O(n)或结尾处O(1)
- **list:** 强调元素的**快速插入和删除**，不支持**随机访问**迭代器
  - 插入/删除为O(1)
- **deque:** 类似vector容器，但强调在**两端**处的**快速插入和删除**
  - 均为O(1)

上节课知识回顾

- 容器
  - 顺序容器介绍
  - 关联容器介绍
  - 容器适配器介绍
- 迭代器
- 算法
- **顺序容器**
  - **vector/list/deque**

上节课知识点复习

- **将一个对象放入STL中的容器里时：**

  A. 实际上被放入的是该对象的一个拷贝(副本)

  B. 实际上被放入的是该对象的指针

  C. 实际上被放入的是该对象的引用

  D. 实际上被放入的就是该对象自身

7

上节课知识点复习

- 给定如下list容器，下述哪条语句是正确的

```
list<int> v; list<int>::const_iterator ii;
A. for( ii = v.begin(); ii != v.end (); ii = ii + 1 )
     cout << * ii;
B. for( ii = v.begin(); ii != v.end (); ii ++ )
     cout << * ii;
C. for( ii = v.begin(); ii < v.end (); ii ++ )
     cout << * ii;
D. for (int i = 0; i < v.size(); i ++)
     cout << v[i];
```


8

上节课知识点复习

- **map, set, list, vector, deque这五种容器中，有几种是排序的?**

A. 1

B. 2

C. 3

D. 4

上节课知识点复习

- **vector没有以下哪个成员函数：**

A. push_back

B. front

C. pop_back

D. push_front

上节课知识点复习

- **binary_search**在查找过程中，比较元素和被查找的值是否相等时，用哪个运算符进行比较？

A. =

B. ==

C. <

D. < 和 ==，一个都不能少

主要内容

- 函数对象
- 关联容器
  - multiset容器
  - set容器
  - multimap容器
  - map容器
- 容器适配器
  - stack / queue / priority_queue

5 函数对象

- 一个类**重载了( )为成员函数** → 该类为**函数对象类**
- 这个类的对象 → **函数对象**
- 看上去像函数调用，实际上也执行了函数调用

5 函数对象

- **函数对象 – 目的**
  - 为了**STL算法** 可复用, 其中的子操作应该是参数化的
  - 比如 sort的排序原则 (顺序/ 逆序)
  - **函数对象** 就是用来描述这些**子操作**的**对象**

5 函数对象

```
class CMyAverage  {
    public:
        double operator()(int a1,  int a2,  int a3) {
       //重载 () 运算符
            return (double)(a1 + a2 + a3) / 3;
        }
}; //重载 () 运算符时, 参数可以是任意多个
 CMyAverage Average;    //函数对象
 cout << Average(3, 2, 3); // Average.operator(3, 2, 3) 用起来
                           // 看上去像函数调用
                           // 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 -- 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也可以是个函数名/函数指针

Dev C++ 中的 Accumulate 源代码1:

```
//没有函数对象的版本,仅实现累加
   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:

```
//有函数对象的版本,根据函数对象定义的方式实现累加
 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;
    }
```

- 调用accumulate时,和__binary_op对应的实参可以是 函数/函数对象

```
#include <iostream>
#include <vector>
#include <numeric> //accumulate在此文件定义
#include <iterator>
using namespace std;
int sumSquares( int total,  int value)
{
    return total + value * value;
}
```


```
template<class T>
class SumSquaresClass{
    public:
        const T  operator() ( const T & total,  const T & value) {
                return total + value * value;
        }
};
```

```
//注：VS中如下代码也可以
template<class T>
class SumSquaresClass{
    public:
        const T & operator() ( const T & total,  const T & value)
            {  return total + value * value;  }
};
```


```
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);
    ostream_iterator<int> output(cout, " ");
    cout << "1) ";
    copy(v.begin(),  v.end(),  output); cout << endl;
    int result = accumulate(v.begin(),  v.end(),  0,  sumSquares);
    cout << "2) 平方和:  " << result << endl;
    SumSquaresClass<int> s;
    result = accumulate(v.begin(),  v.end(),  0,  s);  // (1)
    cout << "3) 平方和:  " << result << endl;
    result = accumulate(v.begin(),  v.end(),  0,  SumPowers<int>(3));
    cout << "4) 立方和:  " << result << endl;
    result = accumulate(v.begin(),  v.end(),  0,  SumPowers<int>(4));
    cout << "5) 4次方和:  " << result;
    return 0;
}
```

输出：

1) 1 2 3 4 5 6 7 8 9 10

2) 平方和：385

3) 平方和：385

4) 立方和：3025

5) 4次方和：25333

```
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;
}
```


```
result=
accumulate(v.begin(),  v.end(),  0,
    SumSquaresClass<int>());
//(1)效果一样
```


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**

函数对象的参数

根据函数对象参数个数, STL算法用到的主要有三类基类:

- 没有参数的函数对象, 相当于 "f()", 例如:

```
vector<int> V(10);
generate(V.begin(), V.end(), rand);
```

- 一个参数的函数对象, 相当于 "f(x)"

  STL中用unary_function定义了一元函数基类

```
template <class _Arg,  class _Result>
struct unary_function {
    typedef _Arg argument_type;
    typedef _Result result_type;
};
```


函数对象的参数

- 两个参数的函数对象, 二元函数, 相当于 “f(x, y)”, binary_function定义了二元函数基类:

```
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 函数对象类模板

例：**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;
    }
};
```


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，

  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：

  void sort (greater<T> pr);

- 可以用来进行**降序**排序

```
#include <list>
#include <iostream>
#include <iterator>
using namespace std;
int main(){
    const int SIZE = 5;
    int a[SIZE] = {5, 1, 4, 2, 3};
    list<int> lst(a, a+SIZE);
    lst.sort(greater<int>()); // greater<int>()是个对象
                              //本句进行降序排序
    ostream_iterator<int> output(cout,  ", ");
    copy( lst.begin(),  lst.end(),  output);
    cout << endl;
    return 0;
}
```

输出: 5, 4, 3, 2, 1,

讨论: 相关概念区分

- **函数指针**

  类型名 (* 指针变量名)(参数类型1, 参数类型2, …);

  特点: 指定了参数类型

- **函数模板**

  template<class T1, class T2, class T3>

  T3 fun(T1 arg1, T2 arg2, string s, int k)

- **函数对象**

  - 定义了**函数operator()**的任意**类的对象**

  - 可以包含其他成员(属性/函数)

  - **优点:** 可在对象内部修改而不用改动外部接口, 可存储先前调用结果的数据成员, 编译器可通过内联函数对象来增强性能

函数对象 vs. 函数指针(1)

- 函数对象实质上是实现了operator() "括号操作符"的类

```
class Add {
    public:
        int operator()(int a, int b) {
            return a + b;
    }
};
Add add; // 定义函数对象
cout << add(3, 2); // 5
```


函数对象 vs. 函数指针(2)

- 而函数指针

```
int AddFunc(int a,  int b) {
    return a + b;
}
typedef int (*Add) (int a,  int b);
Add add = &AddFunc;
cout << add(3, 2); // 5
```

- 除了定义方式不一样, 使用方式可是一样的

```
cout << add(3, 2);
```


函数对象 vs. 函数指针(3)

- 既然函数对象与函数指针在使用方式上没什么区别, 那为什么要用函数对象呢?
- 简单而言, **函数对象可以携带附加数据**, 而指针就不行了

```
class less  {
    public:
        less(int num):n(num) { }
        bool operator()(int value){
            return value < n;
        }
    private:
        int n;
};
```

- 使用时

```
less isLess(10);
cout << isLess(9) << " " << isLess(12); // 输出1 0
```

函数对象 vs. 函数指针(4)

- 要想让一个函数既能接受函数指针, 也能接受函数对象, 最方便的方法就是用模板

```
template<typename FUNC>
int count_n(int* array,  int size,  FUNC func)  {
    int count = 0;
    for(int i = 0; i < size; ++i)
        if(func(array[i]))
            count ++;
    return count;
}
```

```
const int SIZE = 5;
int array[SIZE] = { 50,  30,  9,  7,  20};

//使用函数对象
cout << count_n(array, SIZE,
less(10)); // 2
```

```
//用函数指针也没有问题:
bool less10(int v)  {
    return v < 10;
}
cout << count_n(array, SIZE,
less10); // 2
```

```
#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;
        }
}; //a1和a2的个位比较, 若a1的个位小则返回True, 否则返回False
bool MyCompare(int a1,  int a2){
    if( (a1%10) < (a2%10) )
        return false;
    else
        return true;
} //a1和a2的个位比较, 若a1的个位小则返回False, 否则返回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

//解题思路:

1) 确定函数首部;

2) 确定函数的主要处理逻辑

```
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;
};
```


6 关联容器

set / multiset / map / multimap

- 内部元素有序排列，新元素插入的位置取决于它的值
- 查找速度快

除了各容器都有的函数外，还支持以下成员函数：

- **find:** 查找
- **lower_bound**
- **upper_bound**
- **count:** 计算等于某个值的元素个数
- **insert:** 插入元素用

6.1 multiset

定义

```
template<class Key,  class Pred = less<Key>,
                class A = allocator<Key> >
class multiset { … };
```

- 第一个参数Key – 容器中每个元素类型
- 第二个参数 Pred 是个 **函数对象**
- Pred决定了multiset 中的元素, "一个比另一个小" 是怎么定义的

  即 Pred(x, y) 如果返回值为true, 则 x比y小
- **Pred的缺省类型是 less<Key>**

6.1 multiset

- less 模板的定义

```
template<class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y)
    { return x < y ; } const;
}; //less模板是靠 < 来比较大小的
```


multiset 的用法

```
class A{
    …
};
multiset <A> a;
```

就等效于

```
multiset<A, less<A>> a;
```

- 由于**less模板是用** < 进行比较的, 所以这都要求 A 的对象能用 < 比较, 即适当重载了 <

//出错的例子:

```
#include <set>
using namespace std;
class A { };
main() {
    multiset<A> a;
    a.insert( A() );  //error
}
```

//编译出错是因为, 插入元素时, multiset会将被插入元素和已有元素进行比较, 以决定新元素的存放位置.

//本例中缺省地就是用less<A>函数对象进行比较, 然而less<A>函数对象进行比较时, 前提是A对象能用 < 进行比较

//但本例中没有适当重载 <

multiset应用实例

- 从 **begin()** 到 **end()** 遍历一个 multiset对象，就是**从小到大遍历**各个元素
- 例子程序

multiset的成员函数

iterator find(const T & val);

在容器中查找值为val的元素，返回其迭代器；如果找不到，返回end()

iterator insert(const T & val);

将val插入到容器中并返回其迭代器

void insert(iterator first, iterator last);

将区间[first, last)插入容器

int count(const T & val);

统计有多少个元素的值和val相等

iterator lower_bound(const T & val);

查找一个**最大的位置** it，使得[begin(), it) 中所有的元素都比 val 小

iterator upper_bound(const T & val);

找一个**最小的位置** it，使得[it, end()) 中所有的元素都比 val 大

pair<iterator, iterator>

equal_range(const T & val);

同时求得**lower_bound和upper_bound**

预备知识：pair模板

讲例子之前先看 **pair 模板**(stl_pair.h里源代码):

```
template<class _T1, class _T2>
struct pair{
  _T1 first;
  _T2 second;
  pair():first(), second() { } //无参数构造函数初始化
  pair(const _T1& __a, const _T2& __b):first(__a), second(__b) { }
  template<class _U1, class _U2>
  pair(const pair<_U1, _U2>& __p):first(__p.first), second(__p.second){ }
};
```

pair模板可以用于生成 key-value对

第三个构造函数： `pair<int, int> p(pair<double, double>(5.5, 4.6));`

`//p.first = 5, p.second = 4`

pair模板

pair模板类支持如下操作：

- **pair<T1, T2> p1:** 创建一个空的pair对象

→它的两个元素分别是T1和T2类型，采用值初始化

- **pair<T1, T2> p1(v1, v2):** 创建一个pair对象

→它的两个元素分别是T1和T2类型

→其中first成员初始化为v1, second成员初始化为v2

- **make_pair(v1, v2):** 以v1和v2值创建一个新的pair对象

→其元素类型分别是v1和v2的类型

```
pair<int, string> p4 = make_pair(200, "Hello");
cout << p4.first << ", " << p4.second << endl;
//输出200, Hello
```


```
#include <set> //使用multiset需包含此文件
#include <iostream>
using namespace std;
class MyLess;
class A {
    private: int n;
    public:
        A(int n_ ) { n = n_; }
    friend bool operator< ( const A & a1,  const A & a2 )
    { return a1.n < a2.n;  }
    friend ostream &  operator<< ( ostream & o,  const A & a2 )
    { o << a2.n;    return o;   }
    friend class MyLess;
};
```


```
class MyLess {
public:
    bool operator()( const A & a1,  const A & a2) {
        return ( a1.n % 10 ) < ( a2.n % 10 );
    }
};
typedef multiset<A> MSET1;
typedef multiset<A,  MyLess> MSET2;
// MSET2 里, 元素的排序规则与 MSET1不同,
//假设, le是一个 MyLess对象, a1和a2是MSET2对象
//里的元素, 那么, le(a1, a2) == true 就说明 a1的个位比a2小
```


```
int main()  {
    const int SIZE = 5;
    A a[SIZE] = { 4, 22, 19, 8, 33 };
    ostream_iterator<A> output(cout, ", ");
    MSET1 m1;
    m1.insert(a,  a+SIZE);   //注意set添加元素的函数与vector不同
    m1.insert(22);           //vector要指定插入起始位置
    cout << "1) " << m1.count(22) << endl;
    MSET1::const_iterator p;
    cout << "2) ";
    for( p = m1.begin(); p != m1.end(); p ++ )
        cout << * p << ", " ;
    cout << endl;
    MSET2 m2;
    m2.insert(a,  a+SIZE);
```

```
1) 2
2) 4, 8, 19, 22, 22, 33,
```


```
    cout << "3) " ;
    copy(m2.begin(),  m2.end(),  output); //COPY函数
    cout << endl;
    MSET1::iterator pp =  m1.find(19);
    if( pp != m1.end() ) //找到
        cout << "found" << endl;
    cout << "4) " ;
    copy(m1.begin(),  m1.end(),  output);
    pair<MSET1::iterator,  MSET1::iterator> pr;
    cout <<  endl;
    cout << "5) ";
    cout << * m1.lower_bound(22) << ", ";
    cout << * m1.upper_bound(22)<< endl;
    pr = m1.equal_range(22);
    cout << "6) " << * pr.first << ", " << * pr.second;
}
```

```
3) 22, 33, 4, 8, 19,
found
4) 4, 8, 19, 22, 22, 33,
5) 22, 33
6) 22, 33
```

输出：

1) 2

2) 4, 8, 19, 22, 22, 33,

3) 22, 33, 4, 8, 19,

found

4) 4, 8, 19, 22, 22, 33,

5) 22, 33

6) 22, 33

6.2 set

template<class Key, class Pred = less<Key>,
class A = allocator<Key> >
class set { … }

- 插入set中已有的元素时, 插入不成功
- 与multiset的区别: 是否允许重复元素
- 与map的区别: 是否显示定义key
  - set/multiset使用**元素本身**作为key

回顾：pair 模板

```
template<class T,  class U>
struct pair {
    typedef T first_type;
    typedef U second_type;
    T first; U second;
    pair();
    pair(const T& x,  const U& y);
    template<class V,  class W>
    pair(const pair<V,  W>& pr);
};
```

map/multimap容器中都是pair模版类的对象且按first从小到大排序

- **pair模板可以用于生成 key-value对**

```
#include <set>
#include <numeric>
#include <iostream>
using namespace std;
int main()  {
    typedef set<double,  less<double> > double_set;
    const int SIZE = 5;
    double a[SIZE] = {2.1, 4.2, 9.5, 2.1, 3.7 };
    double_set doubleSet(a,  a+SIZE);
    ostream_iterator<double> output(cout, " ");
    cout << "1) ";
    copy(doubleSet.begin(),  doubleSet.end(),  output);
    cout << endl;
```

输出：

1) 2.1 3.7 4.2 9.5

```
    pair<double_set::const_iterator,  bool> p;
    p = doubleSet.insert(9.5);
    if( p.second )
        cout << "2) " << * (p.first)  << " inserted" << endl;
    else
        cout << "2) " << * (p.first)  << " not inserted" << endl;
}
```

//insert函数返回值是一个pair对象，其first是被插入元素的迭代器, second代表是否成功插入了

输出：

1) 2.1 3.7 4.2 9.5

2) 9.5 not inserted

6.3 multimap

```
template<class Key,  class T,  class Pred = less<Key>,  class A =
allocator<T> >
class multimap {
    ….
    typedef pair<const Key,  T> value_type;
    …….
};   //Key 代表关键字
```

- multimap中的元素由<关键字, 值>组成, 每个元素是一个pair对象
- multimap中允许多个元素的关键字相同
- 元素按照关键字升序排列, 缺省情况下用 less<Key> 定义关键字的“小于”关系

```
#include <iostream>
#include <map>
using namespace std;
int main()    {
    typedef multimap<int,  double,  less<int> > mmid;
    mmid mmpairs;
    cout << "1) " << mmpairs.count(15) << endl;
    mmpairs.insert(mmid::value_type(15,  99.3));
    mmpairs.insert(mmid::value_type(15,  2.7));
    cout <<  "2) " << mmpairs.count(15) << endl;
    mmpairs.insert(mmid::value_type(30,  111.11));
    mmpairs.insert(mmid::value_type(10,  22.22));
```

输出：

1) 0

2) 2

```
    mmpairs.insert(mmid::value_type(25,  33.333));
    mmpairs.insert(mmid::value_type(20,  9.3));
    for( mmid::const_iterator i = mmpairs.begin();
        i != mmpairs.end(); i ++ )
        cout << "(" << i->first << ", " << i->second
                << ")" << ", ";
}
//输出:
1) 0
2) 2
(10, 22.22),  (15, 99.3), (15, 2.7),  (20, 9.3), (25, 33.333), (30, 111.11)
```

6.4 map

```
template<class Key,  class T,  class Pred = less<Key>,
class A = allocator<T> >
class map {
    ....
    typedef pair<const Key,  T> value_type;
    .......
};
```

- map 中的元素**关键字各不相同**
- 元素按照关键字升序排列, 缺省情况下用 less 定义 "小于"

6.4 map

- 可以用**pairs[key]形式**访问map中的元素
  - pairs 为map容器名, key为关键字的值
  - 该表达式返回的是对关键值为key的元素的值的**引用**
  - 如果没有关键字为key的元素, 则会往pairs里插入一个关键字为key的元素, 并返回其值的引用
- 例如：

map<int, double> pairs;

则 pairs[50] = 5; 会修改pairs中关键字为50的元素, 使其值变成5

```
#include <iostream>
#include <map>
using namespace std;
ostream & operator <<(ostream & o,  const pair<  int, double> & p){
    o << "(" << p.first  << ", " << p.second << ")";
    return o;
}

int main()  {
    typedef map<int, double, less<int> > mmid;
    mmid pairs;
    cout << "1) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(15, 2.7));
    pairs.insert(make_pair(15, 99.3));  //make_pair生成一pair对象
    cout << "2) " << pairs.count(15) << endl;
```

输出：

1) 0

2) 1

```
    pairs.insert(mmid::value_type(20, 9.3));
    mmid::iterator i;
    cout << "3) ";
    for( i = pairs.begin(); i != pairs.end();i ++ )
        cout << * i  << ", ";
    cout << endl;       cout << "4) ";
    int n =  pairs[40]; //如果没有关键字为40的元素, 则插入一个
    for( i = pairs.begin(); i != pairs.end(); i ++ )
        cout << * i  << ", ";
    cout << endl;       cout << "5) ";
    pairs[15] = 6.28; //把关键字为15的元素值改成6.28
    for( i = pairs.begin(); i != pairs.end(); i ++ )
        cout << * i  << ", ";
}
```

输出：

3) (15, 2.7), (20, 9.3),

4) (15, 2.7), (20, 9.3), (40, 0),

5) (15, 6.28), (20, 9.3), (40, 0),

输出：

1) 0

2) 1

3) (15, 2.7), (20, 9.3),

4) (15, 2.7), (20, 9.3), (40, 0),

5) (15, 6.28), (20, 9.3), (40, 0),

7 容器适配器

- 可以用**某种顺序容器**来实现

(让已有的顺序容器以栈/队列的方式工作)

1) **stack**: 头文件 <stack>
   - **栈**，后进先出

2) **queue**: 头文件 <queue>
   - **队列**，先进先出

3) **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

7.1 容器适配器:stack

- 可用 vector, list, deque来实现
  - 缺省情况下, 用deque实现
  - 用vector和deque实现, 比用list实现性能好

```
template<class T, class Cont = deque<T> >
class stack {
    …..
};
```

- stack 是**后进先出**的数据结构, 只能**插入/删除/访问栈顶**的元素

容器适配器:stack

- stack的使用

```
stack<int> stk; //int型栈, 用deque实现
stack<string, vector<string>> str_stk; //string型栈, 用
    vector实现
stack<string, vector<string>> str_stk(svec); //string型栈,
用 vector实现, 并且用向量svec初始化
```

- stack上可以进行以下操作:
  - push: 插入元素
  - pop: 弹出元素
  - top: 返回栈顶元素的引用

7.2 容器适配器: queue

- 和stack 基本类似, 可以用 list和deque实现, 缺省情况下用deque实现

```
template<class T,  class Cont = deque<T> >
class queue {
      …
};
```

- 同样也有push, pop, top函数
- 但是push发生在队尾, pop, top发生在队头, 先进先出

7.3 容器适配器: priority_queue

- 和 queue类似, 可以用vector和deque实现, 缺省情况下用vector实现
- priority_queue 通常用**堆排序技术**实现, 保证**最大的元素总是在最前面**
  - 执行**pop**操作时, **删除的是最大的元素**
  - 执行**top**操作时, **返回的是最大元素的引用**
- 默认的元素比较器是 less<T>

```
#include <queue>
#include <iostream>
using namespace std;
main()  {
    priority_queue<double> priorities;
    priorities.push(3.2);
    priorities.push(9.8);
    priorities.push(5.4);
    while( !priorities.empty() ) {
        cout << priorities.top() << " "; //输出最大元素的引用
        priorities.pop(); //删除最大元素
    }
}
//输出结果：9.8 5.4 3.2
```


总　结

- **函数对象**
  - 函数指针, 函数模板和函数对象
  - accumulate, greater
- **multiset /set / multimap / map容器**
  - multiset/set容器中的对象类**必须重载<**
  - find, lower_bound / upper_bound, count, insert
  - less 函数模板
  - pair对象
- **容器适配器**
  - **stack**
  - **queue**
  - **priority_queue**