C++模板与STL库介绍

提纲

- 1. 概论
- 2. 模板机制的介绍
- 3. STL中的基本概念
- 4. 容器概述
- 5. 迭代器
- 6. 算法简介

概论

- 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

- 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。主要由 Alex Stepanov 开发，于1998年被添加进C++标准
- 有了STL，不必再从头写大多的标准数据结构和算法，并且可获得非常高的性能。

模板

1.假如设计一个求两参数最大值的函数，在实践中我们可能需要定义四个函数：

```
int max ( int a , int b ) { return ( a > b ) ? a , b ; }
long max ( long a , long b ) { return ( a > b ) ? a , b ;}
double max ( double a , double  b ) { return ( a >b)? a , b ; }
char max ( char a , char b ) { return ( a > b ) ? a , b ;}
```

2.这些函数几乎相同，唯一的区别就是形参类型不同

3.需要事先知道有哪些类型会使用这些函数，对于未知类型这些函数不起作用

模板的概念

1. 所谓模板是一种使用无类型参数来产生一系列函数或类的机制。
2. 若一个程序的功能是对某种特定的数据类型进行处理，则可以将所处理的数据类型说明为参数，以便在其他数据类型的情况下使用，这就是模板的由来。
3. 模板是以一种完全通用的方法来设计函数或类而不必预先说明将被使用的每个对象的类型。
4. 通过模板可以产生类或函数的集合，使它们操作不同的数据类型，从而避免需要为每一种数据类型产生一个单独的类或函数。

模板分类

- 函数模板(function template)
  - 是独立于类型的函数
  - 可产生函数的特定版本
- 类模板(class template)
  - 跟类相关的模板，如vector
  - 可产生类对特定类型的版本，如vector<int>

求最大值模板函数实现

1.求两个数最大值，使用模板

```
template < class T >
T max(T a , T b){
    return ( a > b ) ? a , b;
}
```

2.template < 模板形参表>
<返回值类型> <函数名>（模板函数形参表）

```
{
  //函数定义体
}
```

模板工作方式

- 函数模板只是说明，不能直接执行，需要实例化为模板函数后才能执行

- 在说明了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数。该重载函数的定义体与函数模板的函数定义体相同，它称之为模板函数

编写一个对具有n个元素的数组a[ ]求最小值的程序，要求将求最小值的函数设计成函数模板。

```
#include <iostream>
template <class T>
T min(T a[],int n){
    int i;
    T minv=a[0];
    for( i = 1;i < n ; i++){
        if(minv>a[i])
            minv=a[i];
    }
    return minv;
}
```

```
void main(){ ina
a[]={1,3,0,2,7,6,4,5,2};
   double b[]={1.2,-3.4,6.8,9,8};
   cout<<"a数组的最小值为： "
        <<min(a,9)<< endl;
   cout<<"b数组的最小值为： "
        <<min(b,4)<<endl;  }
```

此程序的运行结果为：

a数组的最小值为：0

b数组的最小值为：-3.4

模板优缺点

- 函数模板方法克服了C语言解决上述问题时用大量不同函数名表示相似功能的坏习惯
- 克服了宏定义不能进行参数类型检查的弊端
- 克服了C++函数重载用相同函数名字重写几个函数的繁琐
- 缺点，调试比较困难
  - 一般先写一个特殊版本的函数
  - 运行正确后，改成模板函数

STL中的几个基本概念

- 容器：可容纳各种数据类型的数据结构。
- 迭代器：可依次存取容器中元素的东西
- 算法：用来操作容器中的元素的函数模板。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象。
  - 函数本身与他们操作的数据的结构和类型无关，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。

- 比如，数组int array[100]就是个容器，而 int * 类型的指针变量就可以作为迭代器，可以为这个容器编写一个排序的算法

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。
- 容器分为三大类：

1) 顺序容器

vector：后部插入/删除，直接访问
deque：前/后部插入/删除，直接访问
list：双向链表，任意位置插入/删除

2)关联容器

set：快速查找，无重复元素
multiset：快速查找，可有重复元素
map：一对一映射，无重复元素，基于关键字查找
multimap：一对一映射，可有重复元素，基于关键字查找

前2者合称为第一类容器

3)容器适配器

stack：LIFO
queue：FIFO
priority_queue：优先级高的元素先出

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，所以，放入容器的对象所属的类，还应该实现 == 和 < 运算符。

顺序容器简介

1) vector 头文件 <vector>

实际上就是个动态数组。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。

2) deque 头文件 <deque>

也是个动态数组，随机存取任何元素都能在常数时间完成(但性能次于vector)。在两端增删元素具有较佳的性能。

3) list 头文件 <list>

双向链表，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。

上述三种容器称为顺序容器，是因为元素的插入位置同元素的值无关，只跟插入的时机有关。

关联容器简介

- 关联式容器内的元素是排序的，插入任何元素，都按相应的排序准则来确定其位置。关联式容器的特点是在查找时具有非常好的性能。

1) 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。

2) map/multimap:　头文件 <map>

map与set的不同在于map中存放的是成对的key/value。

并根据key对元素进行排序，可快速地根据key来检索元素

map同multimap的不同在于是否允许多个元素有相同的key值。

上述4种容器通常以平衡二叉树方式实现，插入、查找和删除的时间都是 O(logN)

容器适配器简介

1) 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。即按照后进先出的原则

2) 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。按照先进先出的原则。

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

容器的共有成员函数

1) 所有标准库容器共有的成员函数：

- 相当于按词典顺序比较两个容器大小的运算符：
  =, < , <= , > , >=, == , !=
- empty : 判断容器中是否有元素
- max_size: 容器中最多能装多少元素
- size: 容器中元素个数
- swap: 交换两个容器的内容

比较两个容器的例子

比较两个容器的例子：

```
#include <vector>
#include <iostream>
int main()
{
        std::vector<int> v1;
        std::vector<int> v2;
        v1.push_back (5);
        v1.push_back (1);
        v2.push_back (1);
         v2.push_back (2);
        v2.push_back (3);
        std::cout << (v1 < v2);
        return 0;
}
```

输出：

0

- 若两容器长度相同、所有元素相等，则两个容器就相等，否则为不等。
- 若两容器长度不同，但较短容器中所有元素都等于较长容器中对应的元素，则较短容器小于另一个容器
- 若两个容器均不是对方的子序列，则取决于所比较的第一个不等的元素

容器的成员函数

2) 只在第一类容器中的函数：

begin 返回指向容器中第一个元素的迭代器

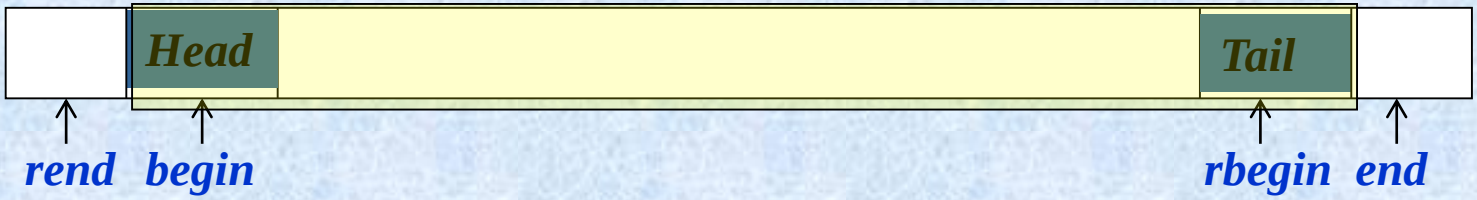
end 返回指向容器中最后一个元素后面的位置的迭代器

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



迭代器

- 用于指向第一类容器中的元素。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。迭代器用法和指针类似。
- 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，则迭代器变成past-the-end值。
- 使用一个past-the-end值的迭代器来访问对象是非法的，就好像使用NULL或未初始化的指针一样。

例如：

```
#include <vector>
#include <iostream>
using namespace std;
int main()  {
    vector<int> v; //一个存放int元素的向量，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;
```

```
    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,

迭代器

- 不同容器上支持的迭代器功能强弱有所不同。
- 容器的迭代器的功能强弱，决定了该容器是否支持STL中的某种算法。
  - 例1：只有第一类容器能用迭代器遍历。
  - 例2：排序算法需要通过随机迭代器来访问容器中的元素，那么有的容器就不支持排序算法。

STL 中的迭代器

- STL 中的迭代器按功能由弱到强分为5种：
  1. 输入：Input iterators 提供对数据的只读访问。
  1. 输出：Output iterators 提供对数据的只写访问
  2. 正向：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器。
  3. 双向：Bidirectional iterators提供读写操作，并能一次一个地向前和向后移动。
  4. 随机访问：Random access iterators提供读写操作，并能在数据中随机移动。

- 编号大的迭代器拥有编号小的迭代器的所有功能，能当作编号小的迭代器使用。

不同迭代器所能进行的操作(功能）

- 所有迭代器： ++p, p ++
- 输入迭代器： * p, p = p1, p == p1 , p!= p1
- 输出迭代器： * p, p = p1
- 正向迭代器： 上面全部
- 双向迭代器： 上面全部，--p, p --,
- 随机访问迭代器： 上面全部，以及：
  - p+= i, p -= i,
  - p + i： 返回指向 p 后面的第i个元素的迭代器
  - p - i： 返回指向 p 前面的第i个元素的迭代器
  - p[i]: p 后面的第i个元素的引用
  - p < p1, p <= p1, p > p1, p>= p1

容器所支持的迭代器类别

容器	迭代器类别
vector	随机
deque	随机
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

例如，vector的迭代器是随机迭代器，所以遍历 vector 可以有以下几种做法：

```
vector<int> v(100);
int i;
for(i = 0;i < v.size() ; i ++)
        cout << v[i];
vector<int>::const_iterator 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;
}
```

而 list 的迭代器是双向迭代器，所以以下代码可以：

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

以下代码则不行：

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

算法简介

- STL中提供能在各种容器中通用的算法，比如插入，删除，查找，排序等。大约有70种标准算法。
  - 算法就是一个个函数模板。
  - 算法通过迭代器来操纵容器中的元素。许多算法需要两个参数，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C语言的数组

算法分类

- 变化序列算法
  - copy ,remove,fill,replace,random_shuffle,swap, …..
  - 会改变容器
- 非变化序列算法：
  - adjacent-find, equal, mismatch,find ,count, search, count_if, for_each, search_n
- 以上函数模板都在<algorithm> 中定义
- 此外还有其他算法，比如<numeric>中的算法

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。
  - 这个区间是个左闭右开的区间，即区间的起点是位于查找范围之中的，而终点不是
- val参数是要查找的元素的值
- 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，则该迭代器指向查找区间终点。

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()  {
        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;
```

```
        p = find(v.begin(),v.end(),9);
        if( p == v.end())
                cout << "not found " << endl;
        p = find(v.begin()+1,v.end()-2,1);
        if( p != v.end())
                cout << * p << endl;
        int * pp = find( array,array+4,20);
        cout << * pp << endl;
}
```

输出:

```
3
not found
3
20
```

顺序容器

- 除前述共同操作外，顺序容器还有以下共同操作：
  - 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

vector

- 支持随机访问迭代器，所有STL算法都能对vector操作。
- 随机访问时间为常数。在尾部添加速度很快，在中间插入慢。实际上就是动态数组。

例1：

```
int main()  {    int i;
        int a[5] = {1,2,3,4,5 };    vector<int>  v(5);
        cout << v.end()  - v.begin() << endl;
        for( i = 0;i < v.size();i ++ )  v[i] = i;
        v.at(4) = 100;
        for( i = 0;i < v.size();i ++ )
                cout << v[i] << "," ;
        cout << endl;
        vector<int> v2(a,a+5);  //构造函数
        v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置插入 13
        for( i = 0;i < v2.size();i ++ )
                cout << v2[i] << "," ;
        return 0;
}
```

输出：

5

0,1,2,3,100,

1,2,13,3,4,5,

例2：

```
int main()  {
        const int SIZE = 5;
        int a[SIZE] = {1,2,3,4,5 };
        vector<int> v (a,a+5);  //构造函数
        try {
                v.at(100) = 7;
        }
        catch( out_of_range e) {
                cout << e.what() << endl;
        }
        cout << v.front() << “,” << v.back() << endl;
        v.erase(v.begin());
        ostream_iterator<int> output(cout ,“*");
        copy (v.begin(),v.end(),output);
        v.erase( v.begin(),v.end());  //等效于 v.clear();
```

```
        if( v.empty ())
                cout << "empty" << endl;
        v.insert (v.begin(),a,a+SIZE);
        copy (v.begin(),v.end(),output);
}

// 输出:
invalid vector<T> subscript
1,5
2*3*4*5*empty
1*2*3*4*5*
```

算法解释

- ostream_iterator<int> output(cout ,“*");
  - 定义了一个 ostream_iterator 对象，可以通过cout输出以 * 分隔的一个个整数
- copy (v.begin(),v.end(),output);
  - 导致v的内容在 cout上输出
- copy 函数模板(算法）：

  template<class InIt, class OutIt>

  OutIt copy(InIt first, InIt last, OutIt x);
  - 本函数对每个在区间[0, last - first)中的N执行一次 *(x+N) = *(first + N)，返回 x + N
- 对于copy (v.begin(),v.end(),output);
  - first 和 last 的类型是 vector<int>::const_iterator
  - output 的类型是 ostream_iterator<int>

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
  - 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

deque 容器

- 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

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
  - map关联数组：元素通过键来存储和读取
  - set大小可变的集合，支持通过键实现的快速读取
  - multimap支持同一个键多次出现的map类型
  - multiset支持同一个键多次出现的set类型
- 与顺序容器的本质区别
  - 关联容器是通过键(key)存储和读取元素的
  - 而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

关联容器

- 除了各容器都有的函数外，还支持以下成员函数：设m表容器，k表键值
  - m.find(k)：如果容器中存在键为k的元素，则返回指向该元素的迭代器。如果不存在，则返回end()值。
  - m.lower_bound(k)：返回一个迭代器，指向键不小于k的第一个元素
  - m.upper_bound(k)：返回一个迭代器，指向键大于k的第一个元素
  - m.count(k)：返回m中k的出现次数
  - 插入元素用 insert

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。

pair模板

- pair模板类用来绑定两个对象为一个新的对象，该类型在<utility>头文件中定义。
- 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
  - p1 < p2字典次序：如果p1.first<p2.first或者!(p2.first < p1.first)&& p1.second<p2.second，则返回true
  - p1 == p2：如果两个pair对象的first和second成员依次相等，则这两个对象相等。
  - p.first：返回p中名为first的（公有）数据成员
  - p.second：返回p中名为second的（公有）数据成员

```
#include <set>
#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;
    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;
    return 0; }
```

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

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

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”

map

- 可以用pairs[key] 访形式问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
- 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pair对象
    cout << "2) " << pairs.count(15) << endl;
    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  << ",";
        return 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),

思考题

- 如何用程序用来统计一篇英文文章中单词出现的频率（为简单起见，假定依次从键盘输入该文章）

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, int> wordCount;
    string word;
    while (cin >> word)
        ++wordCount[word];

    for (map<string, int>::iterator it = wordCount.begin(); it !=
                wordCount.end(); ++it)
        cout<<"Word: "<<(*it).first<<" \tCount: "<<(*it).second<<endl;

    return 0;
}
```

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 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

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

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。即执行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;
int main()  {
    priority_queue<double> priorities;
    priorities.push(3.2);
    priorities.push(9.8);
    priorities.push(5.4);
    while( !priorities.empty() ) {
        cout << priorities.top() << " ";    priorities.pop();
    }
    return 0;
}
//输出结果：9.8 5.4 3.2
```

排序和查找算法

- Sort

  template<class RanIt>
  void sort(RanIt first, RanIt last);

- find

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

- binary_search 折半查找，要求容器已经有序

  template<class FwdIt, class T>
  bool binary_search(FwdIt first, FwdIt last, const T& val);

```
int main()  {
    const int SIZE = 10;
    int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
    vector<int> v(a1,a1+SIZE);
    vector<int>::iterator location;
    location = find(v.begin(),v.end(),10);
    if( location != v.end()) {
        cout << endl << "1) " << location - v.begin();
    }
     sort(v.begin(),v.end());
     if( binary_search(v.begin(),v.end(),9))
        cout << endl << “2) " << "9 found";
    else
       cout << endl << " 2) " << " 9 not found";
    return 0;
}
```

输出：

1) 8

2) 9 found

sort

- sort 实际上是快速排序，时间复杂度 O(n*log(n));
  - 平均性能最优。但是最坏的情况下，性能可能非常差。

  如果要保证“最坏情况下”的性能，那么可以使用stable_sort
- stable_sort
  - stable_sort 实际上是归并排序（将两个已经排序的序列合并成一个序列），特点是能保持相等元素之间的先后次序

- stable_sort 用法和 sort相同
  - 排序算法要求随机存取迭代器的支持，所以list 不能使用排序算法，要使用list::sort

小结

- C++模板
  - 函数模板
- STL库
  - 各种容器
  - 迭代器
  - 算法