目　录

1 STL 简介

http://www.cplusplus.com/reference/stl/更加详细的资料

C++ STL (Standard Template Library标准模板库) 是通用类模板和算法的集合，它提供给程序员一些标准的数据结构的实现如 queues(队列), lists(链表), 和 stacks(栈)等.

C++ STL 提供给程序员以下三类数据结构的实现：

标准容器类

顺序性容器

vector 从后面快速的插入与删除，直接访问任何元素

deque 从前面或后面快速的插入与删除，直接访问任何元素

list 双链表，从任何地方快速插入与删除

关联容器

set 快速查找，不允许重复值

multiset 快速查找，允许重复值

map 一对多映射，基于关键字快速查找，不允许重复值

multimap 一对多映射，基于关键字快速查找，允许重复值

容器适配器

stack 后进先出

queue 先进先出

priority_queue 最高优先级元素总是第一个出列

程序员使用复杂数据结构的最困难的部分已经由STL完成. 如果程序员想使用包含int数据的stack, 他只要写出如下的代码:

stack<int> myStack;

接下来, 他只要简单的调用 push() 和 pop() 函数来操作栈. 借助 C++ 模板的威力, 他可以指定任何的数据类型，不仅仅是int类型. STL stack实现了栈的功能，而不管容纳的是什么数据类型.

2 顺序性容器

2.1 C++ Vector（向量容器）

是一个线性顺序结构。相当于数组，但其大小可以不预先指定，并且自动扩展。它可以像数组一样被操作，由于它的特性我们完全可以将vector 看作动态数组。

在创建一个vector 后，它会自动在内存中分配一块连续的内存空间进行数据

存储，初始的空间大小可以预先指定也可以由vector 默认指定，这个大小即capacity （）函数的返回值。当存储的数据超过分配的空间时vector 会重新分配一块内存块，但这样的分配是很耗时的，在重新分配空间时它会做这样的动作：

首先，vector 会申请一块更大的内存块；

然后，将原来的数据拷贝到新的内存块中；

其次，销毁掉原内存块中的对象（调用对象的析构函数）；

最后，将原来的内存空间释放掉。

如果vector 保存的数据量很大时，这样的操作一定会导致糟糕的性能（这也是vector 被设计成比较容易拷贝的值类型的原因）。所以说vector 不是在什么情况下性能都好，只有在预先知道它大小的情况下vector 的性能才是最优的。

vector 的特点：

(1) 指定一块如同数组一样的连续存储，但空间可以动态扩展。即它可以像数组一样操作，并且可以进行动态操作。通常体现在push_back() pop_back() 。

(2) 随机访问方便，它像数组一样被访问，即支持[ ] 操作符和vector.at()

(3) 节省空间，因为它是连续存储，在存储数据的区域都是没有被浪费的，但是要明确一点vector 大多情况下并不是满存的，在未存储的区域实际是浪费的。

(4) 在内部进行插入、删除操作效率非常低，这样的操作基本上是被禁止的。Vector 被设计成只能在后端进行追加和删除操作，其原因是vector 内部的实现是按照顺序表的原理。

(5) 只能在vector 的最后进行push 和pop ，不能在vector 的头进行push 和pop 。

(6) 当动态添加的数据超过vector 默认分配的大小时要进行内存的重新分配、拷贝与释放，这个操作非常消耗性能。 所以要vector 达到最优的性能，最好在创建vector 时就指定其空间大小。

Vectors 包含着一系列连续存储的元素,其行为和数组类似。访问Vector中的任意元素或从末尾添加元素都可以在常量级时间复杂度内完成，而查找特定值的元素所处的位置或是在Vector中插入元素则是线性时间复杂度。

1.Constructors 构造函数

```
vector<int> v1; //构造一个空的vector
vector<int> v1( 5, 42 ); //构造了一个包含5个值为42的元素的Vector
```

2.Operators 对vector进行赋值或比较

C++ Vectors能够使用标准运算符: ==, !=, <=, >=, <, 和 >.

要访问vector中的某特定位置的元素可以使用 [] 操作符.

两个vectors被认为是相等的,如果:

1.它们具有相同的容量

2.所有相同位置的元素相等.

vectors之间大小的比较是按照词典规则.

3.assign() 对Vector中的元素赋值

语法：

```
void assign( input_iterator start, input_iterator end );
// 将区间[start, end)的元素赋到当前vector
void assign( size_type num, const TYPE &val );
// 赋num个值为val的元素到vector中,这个函数将会清除掉为vector赋值以前的内容.
```

4.at() 返回指定位置的元素

语法：

TYPE at(size_type loc);//差不多等同v[i];但比v[i]安全;

5.back() 返回最末一个元素

6.begin() 返回第一个元素的迭代器

7.capacity() 返回vector所能容纳的元素数量(在不重新分配内存的情况下）

8.clear() 清空所有元素

9.empty() 判断Vector是否为空（返回true时为空）

10.end() 返回最末元素的迭代器(译注:实指向最末元素的下一个位置)

11.erase() 删除指定元素

语法：

iterator erase(iterator loc);//删除loc处的元素

iterator erase(iterator start, iterator end);//删除start和end之间的元素

12.front() 返回第一个元素的引用

13.get_allocator() 返回vector的内存分配器

14.insert() 插入元素到Vector中

语法：

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

//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,

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

//在指定位置loc前插入num个值为val的元素

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

//在指定位置loc前插入区间[start, end)的所有元素

15.max_size() 返回Vector所能容纳元素的最大数量（上限）

16.pop_back() 移除最后一个元素

17.push_back() 在Vector最后添加一个元素

18.rbegin() 返回Vector尾部的逆迭代器

19.rend() 返回Vector起始的逆迭代器

20.reserve() 设置Vector最小的元素容纳数量

//为当前vector预留至少共容纳size个元素的空间

21.resize() 改变Vector元素数量的大小

语法:

void resize(size_type size, TYPE val);

//改变当前vector的大小为size,且对新创建的元素赋值val

22.size() 返回Vector元素数量的大小

23.swap() 交换两个Vector

语法：

void swap(vector &from);

2.2 C++ List（双向链表）

是一个线性链表结构，它的数据由若干个节点构成，每一个节点都包括一个信息块（即实际存储的数据）、一个前驱指针和一个后驱指针。它无需分配指定的内存大小且可以任意伸缩，这是因为它存储在非连续的内存空间中，并且由指

针将有序的元素链接起来。

由于其结构的原因，list 随机检索的性能非常的不好，因为它不像vector 那样直接找到元素的地址，而是要从头一个一个的顺序查找，这样目标元素越靠后，它的检索时间就越长。检索时间与目标元素的位置成正比。

虽然随机检索的速度不够快，但是它可以迅速地在任何节点进行插入和删除操作。因为list 的每个节点保存着它在链表中的位置，插入或删除一个元素仅对最多三个元素有所影响，不像vector 会对操作点之后的所有元素的存储地址都有所影响，这一点是vector 不可比拟的。

list 的特点：

(1) 不使用连续的内存空间这样可以随意地进行动态操作；
(2) 可以在内部任何位置快速地插入或删除，当然也可以在两端进行push和pop 。
(3) 不能进行内部的随机访问，即不支持[ ] 操作符和vector.at() ；

Lists将元素按顺序储存在链表中，与向量(vectors)相比，它允许快速的插入和删除，但是随机访问却比较慢.

1.assign() 给list赋值

语法:

```
void assign( input_iterator start, input_iterator end );
//以迭代器start和end指示的范围为list赋值
void assign( size_type num, const TYPE &val );
//赋值num个以val为值的元素。
```

2.back() 返回最后一个元素的引用
3.begin() 返回指向第一个元素的迭代器
4.clear() 删除所有元素
5.empty() 如果list是空的则返回true
6.end() 返回末尾的迭代器
7.erase() 删除一个元素

语法：

```
iterator erase( iterator loc );//删除loc处的元素
iterator erase( iterator start, iterator end ); //删除start和end之间的元素
```

8.front() 返回第一个元素的引用
9.get_allocator() 返回list的配置器
10.insert() 插入一个元素到list中

语法：

```
iterator insert( iterator loc, const TYPE &val );
//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,
void insert( iterator loc, size_type num, const TYPE &val );
//定位置loc前插入num个值为val的元素
void insert( iterator loc, input_iterator start, input_iterator end );
//在指定位置loc前插入区间[start, end)的所有元素
```

11.max_size() 返回list能容纳的最大元素数量
12.merge() 合并两个list

语法:

void merge(list &lst);//把自己和lst链表连接在一起

void merge(list &lst, Comp compfunction);

//指定compfunction，则将指定函数作为比较的依据。

13.pop_back() 删除最后一个元素

14.pop_front() 删除第一个元素

15.push_back() 在list的末尾添加一个元素

16.push_front() 在list的头部添加一个元素

17.rbegin() 返回指向第一个元素的逆向迭代器

18.remove() 从list删除元素

语法:

void remove(const TYPE &val);

//删除链表中所有值为val的元素

19.remove_if() 按指定条件删除元素

20.rend() 指向list末尾的逆向迭代器

21.resize() 改变list的大小

语法:

void resize(size_type num, TYPE val);

//把list的大小改变到num。被加入的多余的元素都被赋值为val22.

22.reverse() 把list的元素倒转

23.size() 返回list中的元素个数

24.sort() 给list排序

语法:

void sort();//为链表排序，默认是升序

void sort(Comp compfunction);//采用指定函数compfunction来判定两个元素的大小。

25.splice() 合并两个list

语法:

void splice(iterator pos, list &lst);//把lst连接到pos的位置

void splice(iterator pos, list &lst, iterator del);//插入lst中del所指元素到现链表的pos上

void splice(iterator pos, list &lst, iterator start, iterator end);//用start和end指定范围。

26.swap() 交换两个list

语法：

void swap(list &lst);// 交换lst和现链表中的元素

27.unique() 删除list中重复的元素

语法:

void unique();//删除链表中所有重复的元素

void unique(BinPred pr);// 指定pr，则使用pr来判定是否删除。

2.3 C++ Deque(双向队列)

是一种优化了的、对序列两端元素进行添加和删除操作的基本序列容器。它允许较为快速地随机访问，但它不像vector 把所有的对象保存在一块连续的内存块，而是采用多个连续的存储块，并且在一个映射结构中保存对这些块及其顺序的跟踪。向deque 两端添加或删除元素的开销很小。它不需要重新分配空间，所

以向末端增加元素比vector 更有效。

实际上，deque 是对vector 和list 优缺点的结合，它是处于两者之间的一种容器。

deque 的特点：

(1) 随机访问方便，即支持[ ] 操作符和vector.at() ，但性能没有vector 好；

(2) 可以在内部进行插入和删除操作，但性能不及list ；

(3) 可以在两端进行push 、pop ；

(4) 相对于verctor 占用更多的内存。

双向队列和向量很相似，但是它允许在容器头部快速插入和删除（就像在尾部一样）。

1.Constructors 创建一个新双向队列

语法:

```
deque();//创建一个空双向队列
deque( size_type size );// 创建一个大小为size的双向队列
deque( size_type num, const TYPE &val ); //放置num个val的拷贝到队列中
deque( const deque &from );// 从from创建一个内容一样的双向队列
deque( input_iterator start, input_iterator end );
// start 和 end - 创建一个队列，保存从start到end的元素。
```

2.Operators 比较和赋值双向队列

```
//可以使用[]操作符访问双向队列中单个的元素
```

3.assign() 设置双向队列的值

语法:

```
void assign( input_iterator start, input_iterator end);
//start和end指示的范围为双向队列赋值
void assign( Size num, const TYPE &val );//设置成num个val。
```

4.at() 返回指定的元素

语法:

```
reference at( size_type pos ); 返回一个引用，指向双向队列中位置pos上的元素
```

5.back() 返回最后一个元素

语法:

```
reference back();//返回一个引用，指向双向队列中最后一个元素
```

6.begin() 返回指向第一个元素的迭代器

语法:

```
iterator begin();//返回一个迭代器，指向双向队列的第一个元素
```

7.clear() 删除所有元素

8.empty() 返回真如果双向队列为空

9.end() 返回指向尾部的迭代器

10.erase() 删除一个元素

语法:

```
iterator erase( iterator pos ); //删除pos位置上的元素
iterator erase( iterator start, iterator end ); //删除start和end之间的所有元素
//返回指向被删除元素的后一个元素
```

11.front() 返回第一个元素的引用

12.get_allocator() 返回双向队列的配置器
13.insert() 插入一个元素到双向队列中

语法:

iterator insert(iterator pos, size_type num, const TYPE &val); //pos前插入num个val值

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

//插入从start到end范围内的元素到pos前面

14.max_size() 返回双向队列能容纳的最大元素个数
15.pop_back() 删除尾部的元素
16.pop_front() 删除头部的元素
17.push_back() 在尾部加入一个元素
18.push_front() 在头部加入一个元素
19.rbegin() 返回指向尾部的逆向迭代器
20.rend() 返回指向头部的逆向迭代器
21.resize() 改变双向队列的大小
22.size() 返回双向队列中元素的个数
23.swap() 和另一个双向队列交换元素

语法：

void swap(deque &target);// 交换target和现双向队列中元素

2.4 三者比较

vector 是一段连续的内存块，而deque 是多个连续的内存块， list 是所有数据元素分开保存，可以是任何两个元素没有连续。

vector 的查询性能最好，并且在末端增加数据也很好，除非它重新申请内存段；适合高效地随机存储。

list 是一个链表，任何一个元素都可以是不连续的，但它都有两个指向上一元素和下一元素的指针。所以它对插入、删除元素性能是最好的，而查询性能非常差；适合大量地插入和删除操作而不关心随机存取的需求。

deque 是介于两者之间，它兼顾了数组和链表的优点，它是分块的链表和多个数组的联合。所以它有被list好的查询性能，有被vector好的插入、删除性能。如果你需要随即存取又关心两端数据的插入和删除，那么deque是最佳之选。

3 关联容器

3.1 特点

set, multiset, map, multimap 是一种非线性的树结构，具体的说采用的是一种比较高效的特殊的平衡检索二叉树—— 红黑树结构。（至于什么是红黑树，我也不太理解，只能理解到它是一种二叉树结构）

因为关联容器的这四种容器类都使用同一原理，所以他们核心的算法是一致的，但是它们在应用上又有一些差别，先描述一下它们之间的差别。

set 又称集合，实际上就是一组元素的集合，但其中所包含的元素的值是唯一的，且是按一定顺序排列的，集合中的每个元素被称作集合中的实例。因为其内部是通过链表的方式来组织，所以在插入的时候比vector 快，但在查找和末尾添加上比vector 慢。

multiset 是多重集合，其实现方式和set 是相似的，只是它不要求集合中的元素是唯一的，也就是说集合中的同一个元素可以出现多次。

map 提供一种“键- 值”关系的一对一的数据存储能力。其“键”在容器中不可重复，且按一定顺序排列（其实我们可以将set 也看成是一种键- 值关系的存储，只是它只有键没有值。它是map 的一种特殊形式）。由于其是按链表的方式存储，它也继承了链表的优缺点。

multimap 和map 的原理基本相似，它允许“键”在容器中可以不唯一。

关联容器的特点是明显的，相对于顺序容器，有以下几个主要特点：

1、其内部实现是采用非线性的二叉树结构，具体的说是红黑树的结构原理实现的；

2、set 和map 保证了元素的唯一性，mulset 和mulmap 扩展了这一属性，可以允许元素不唯一；

3、元素是有序的集合，默认在插入的时候按升序排列。

基于以上特点，

1、关联容器对元素的插入和删除操作比vector 要快，因为vector 是顺序存储，而关联容器是链式存储；比list 要慢，是因为即使它们同是链式结构，但list 是线性的，而关联容器是二叉树结构，其改变一个元素涉及到其它元素的变动比list 要多，并且它是排序的，每次插入和删除都需要对元素重新排序；

2、关联容器对元素的检索操作比vector 慢，但是比list 要快很多。vector 是顺序的连续存储，当然是比不上的，但相对链式的list 要快很多是因为list 是逐个搜索，它搜索的时间是跟容器的大小成正比，而关联容器 查找的复杂度基本是Log(N) ，比如如果有1000 个记录，最多查找10 次，1,000,000 个记录，最多查找20 次。容器越大，关联容器相对list 的优越性就越能体现；

3、在使用上set 区别于vector,deque,list 的最大特点就是set 是内部排序的，这在查询上虽然逊色于vector ，但是却大大的强于list 。

4、在使用上map 的功能是不可取代的，它保存了“键- 值”关系的数据，而这种键值关系采用了类数组的方式。数组是用数字类型的下标来索引元素的位置，而map 是用字符型关键字来索引元素的位置。在使用上map 也提供了一种类数组操作的方式，即它可以通过下标来检索数据，这是其他容器做不到的，当然也包括set 。（STL 中只有vector 和map 可以通过类数组的方式操作元素，即如同ele[1] 方式）

3.2 C++ Sets & MultiSets

集合(Set)是一种包含已排序对象的关联容器。多元集合(MultiSets)和集合(Sets)相像，只不过支持重复对象,其用法与set基本相同。

1.begin() 返回指向第一个元素的迭代器

2.clear() 清除所有元素

3.count() 返回某个值元素的个数

4.empty() 如果集合为空，返回true
5.end() 返回指向最后一个元素的迭代器
6.equal_range() 返回第一个>=关键字的迭代器和>关键字的迭代器
语法：
pair <iterator,iterator>equal_range(const key_type &key);
//key是用于排序的关键字
Set<int> ctr;
例如：
Pair<set<int>::iterator,set<int>::iterarot>p;
For(i=0;i<=5;i++) ctr.insert(i);
P=ctr.equal_range(2);
那么*p.first==2;*p.second==3;
7.erase() 删除集合中的元素
语法:
iterator erase(iterator i); //删除i位置元素
iterator erase(iterator start, iterator end);
//删除从start开始到end(end为第一个不被删除的值)结束的元素
size_type erase(const key_type &key);
//删除等于key值的所有元素（返回被删除的元素的个数）
//前两个返回第一个不被删除的双向定位器,不存在返回末尾
//第三个返回删除个数
8.find() 返回一个指向被查找到元素的迭代器
语法:
iterator find(const key_type &key);
//查找等于key值的元素，并返回指向该元素的迭代器;
//如果没有找到,返回指向集合最后一个元素的迭代器
9.get_allocator() 返回集合的分配器
10.insert() 在集合中插入元素
语法:
iterator insert(iterator i, const TYPE &val); //在迭代器i前插入val
void insert(input_iterator start, input_iterator end);
//将迭代器start开始到end（end不被插入）结束返回内的元素插入到集合中
pair insert(const TYPE &val);
//插入val元素，返回指向该元素的迭代器和一个布尔值来说明val是否成功被插入
//应该注意的是在集合(Sets中不能插入两个相同的元素)
11.lower_bound() 返回指向大于（或等于）某值的第一个元素的迭代器
语法:
iterator lower_bound(const key_type &key);
//返回一个指向大于或者等于key值的第一个元素的迭代器
12.key_comp() 返回一个用于元素间值比较的函数
语法:
key_compare key_comp();
//返回一个用于元素间值比较的函数对象
13.max_size() 返回集合能容纳的元素的最大限值

14.rbegin() 返回指向集合中最后一个元素的反向迭代器

示例：

```
Set<int> ctr;
Set<int>::reverse_iterator rcp;
For(rcp=ctr.rbegin();rcp!=ctr.rend();rcp++)
Cout<<*rcp<<” ”;
```

15.rend() 返回指向集合中第一个元素的反向迭代器

16.size() 集合中元素的数目

17.swap() 交换两个集合变量

语法:

```
void swap( set &object ); //交换当前集合和object集合中的元素
```

18.upper_bound() 返回大于某个值元素的迭代器

语法:

```
iterator upwer_bound( const key_type &key );
//返回一个指向大于key值的第一个元素的迭代器
```

19.value_comp() 返回一个用于比较元素间的值的函数

语法:

```
iterator upper_bound( const key_type &key );//返回一个用于比较元素间的值的函数对象
```

3.3 C++ Maps & MultiMaps

C++ Maps是一种关联式容器，包含“关键字/值”对。

C++ Multimaps和maps很相似，但是MultiMaps允许重复的元素。

1.begin() 返回指向map头部的迭代器

2.clear() 删除所有元素

3.count() 返回指定元素出现的次数

语法:

```
size_type count( const KEY_TYPE &key );
//返回map中键值等于key的元素的个数
```

4.empty() 如果map为空则返回true

5.end() 返回指向map末尾的迭代器

6.equal_range() 返回特殊条目的迭代器对

语法:

```
pair equal_range( const KEY_TYPE &key );
返回两个迭代器,指向第一个键值为key的元素和指向最后一个键值为key的元素
```

7.erase() 删除一个元素

语法:

```
void erase( iterator i ); //删除i元素
void erase( iterator start, iterator end ); //删除从start开始到end（不包括end）结束的元素
size_type erase( const key_type &key );
//删除等于key值的所有元素（返回被删除的元素的个数）
```

8.find() 查找一个元素

语法:

```
iterator find( const key_type &key );
```

//查找等于key值的元素，并返回指向该元素的迭代器;
//如果没有找到,返回指向集合最后一个元素的迭代器.

9.get_allocator() 返回map的配置器

10.insert() 插入元素

语法:

```
iterator insert( iterator pos, const pair<KEY_TYPE,VALUE_TYPE> &val );
//插入val到pos的后面，然后返回一个指向这个元素的迭代器
void insert( input_iterator start, input_iterator end );
//插入start到end的元素到map中
pair<iterator, bool> insert( const pair<KEY_TYPE,VALUE_TYPE> &val );
//只有在val不存在时插入val。返回指向被插入元素的迭代器和描述是否插入的bool值
```

11.key_comp() 返回比较元素key的函数

语法:

```
key_compare key_comp();
//返回一个用于元素间值比较的函数对象
```

12.lower_bound() 返回键值>=给定元素的第一个位置

语法:

```
iterator lower_bound( const key_type &key );
//返回一个指向大于或者等于key值的第一个元素的迭代器
```

13.max_size() 返回可以容纳的最大元素个数

14.rbegin() 返回一个指向map尾部的逆向迭代器

15.rend() 返回一个指向map头部的逆向迭代器

16.size() 返回map中元素的个数

17.swap() 交换两个map

语法:

```
void swap( map &obj );
//swap()交换obj和现map中的元素
```

18.upper_bound() 返回键值>给定元素的第一个位置

语法:

```
iterator upwer_bound( const key_type &key );
//返回一个指向大于key值的第一个元素的迭代器
```

19.value_comp() 返回比较元素value的函数

语法:

```
value_compare value_comp();
//返回一个用于比较元素value的函数
```

4 容器适配器

4.1 特点

STL 中包含三种适配器：栈stack 、队列queue 和优先级priority_queue 。适配器是容器的接口，它本身不能直接保存元素，它保存元素的机制是调用

另一种顺序容器去实现，即可以把适配器看作“它保存一个容器，这个容器再保存所有元素”。

STL 中提供的三种适配器可以由某一种顺序容器去实现。默认下stack 和queue 基于deque 容器实现，priority_queue 则基于vector 容器实现。当然在创建一个适配器时也可以指定具体的实现容器，创建适配器时在第二个参数上指定具体的顺序容器可以覆盖适配器的默认实现。

由于适配器的特点，一个适配器不是可以由任一个顺序容器都可以实现的。

栈stack 的特点是后进先出，所以它关联的基本容器可以是任意一种顺序容器，因为这些容器类型结构都可以提供栈的操作有求，它们都提供了push_back 、pop_back 和back 操作。

队列queue 的特点是先进先出，适配器要求其关联的基础容器必须提供pop_front 操作，因此其不能建立在vector 容器上。

4.2 C++ Stacks（堆栈）

C++ Stack（堆栈） 是一个容器类的改编，为程序员提供了堆栈的全部功能，——也就是说实现了一个先进后出（FILO）的数据结构。

1.empty() 堆栈为空则返回真
2.pop() 移除栈顶元素
3.push() 在栈顶增加元素
4.size() 返回栈中元素数目
5.top() 返回栈顶元素

4.3 C++ Queues(队列)

C++队列是一种容器适配器，它给予程序员一种先进先出(FIFO)的数据结构。

1.back() 返回一个引用，指向最后一个元素
2.empty() 如果队列空则返回真
3.front() 返回第一个元素
4.pop() 删除第一个元素
5.push() 在末尾加入一个元素
6.size() 返回队列中元素的个数

4.4 C++ Priority Queues(优先队列)

C++优先队列类似队列，但是在这个数据结构中的元素按照一定的断言排列有序。

1.empty() 如果优先队列为空，则返回真
2.pop() 删除第一个元素

3.push() 加入一个元素
4.size() 返回优先队列中拥有的元素的个数
5.top() 返回优先队列中有最高优先级的元素

5 迭代器

5.1 解释

迭代器是一种对象，它能够用来遍历STL容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址。迭代器修改了常规指针的接口，所谓迭代器是一种概念上的抽象：那些行为上象迭代器的东西都可以叫做迭代器。然而迭代器有很多不同的能力，它可以把抽象容器和通用算法有机的统一起来。

迭代器提供一些基本操作符：*、++、==、！=、=。这些操作和C/C++“操作array元素”时的指针接口一致。不同之处在于，迭代器是个所谓的smart pointers，具有遍历复杂数据结构的能力。其下层运行机制取决于其所遍历的数据结构。因此，每一种容器型别都必须提供自己的迭代器。事实上每一种容器都将其迭代器以嵌套的方式定义于内部。因此各种迭代器的接口相同，型别却不同。这直接导出了泛型程序设计的概念：所有操作行为都使用相同接口，虽然它们的型别不同。

5.2 功能特点

迭代器使开发人员不必整个实现类接口。只需提供一个迭代器，即可遍历类中的数据结构，可被用来访问一个容器类的所包函的全部元素，其行为像一个指针，但是只可被进行增加(++)或减少(--)操作。举一个例子，你可用一个迭代器来实现对vector容器中所含元素的遍历。
如下代码对vector容器对象生成和使用了迭代器：

```
vector<int> the_vector;
vector<int>::iterator the_iterator;
for( int i=0; i < 10; i++ )
    the_vector.push_back(i);
int total = 0;
the_iterator = the_vector.begin();
while( the_iterator != the_vector.end() ) {
    total += *the_iterator;
    the_iterator++;
}cout << "Total=" << total << endl;
```

提示：通过对一个迭代器的解引用操作（*），可以访问到容器所包含的元素。

6 C++标准库总结

6.1 容器

6.1.1 序列

vector=========================<vector>

list==========================<list>

deque=========================<deque>

6.1.2 序列适配器

stack:top,push,pop=============<stack>

queue:front,back,push,pop======<queue>

priority_queue:top,push,pop====<queue>

6.1.3 关联容器

map===========================<map>

multimap======================<map>

set===========================<set>

multiset======================<set>

6.1.4 拟容器

string========================<string>

valarray======================<valarray>

bitset========================<bitset>

6.2 算法

http://www.cplusplus.com/reference/algorithm/详细

6.2.1 非修改性序列操作

<algorithm>

for_each()====================对序列中每个元素执行操作

find()========================在序列中找某个值的第一个出现

find_if()=====================在序列中找符合某谓词的第一个元素

find_first_of()===============在序列中找另一序列里的值

adjust_find()=================找出相邻的一对值

count()=======================在序列中统计某个值出现的次数

count_if()====================在序列中统计与某谓词匹配的次数

mismatch()====================找使两序列相异的第一个元素

equal()=======================如果两个序列对应元素都相同则为真

search()======================找出一序列作为子序列的第一个出现位置

find_end()====================找出一序列作为子序列的最后一个出现位置

search_n()====================找出一序列作为子序列的第 n 个出现位置

6.2.2 修改性的序列操作

<algorithm>

transform()==========将操作应用于序列中的每个元素
copy()==========从序列的第一个元素起进行复制
copy_backward()==========从序列的最后元素起进行复制
swap()==========交换两个元素
iter_swap()==========交换由迭代器所指的两个元素
swap_ranges()==========交换两个序列中的元素
replace()==========用一个给定值替换一些元素
replace_if()==========替换满足谓词的一些元素
replace_copy()==========复制序列时用一个给定值替换元素
replace_copy_if()==========复制序列时替换满足谓词的元素
fill()==========用一个给定值取代所有元素
fill_n()==========用一个给定值取代前 n 个元素
generate()==========用一个操作的结果取代所有元素
generate_n()==========用一个操作的结果取代前 n 个元素
remove()==========删除具有给定值的元素
remove_if()==========删除满足谓词的元素
remove_copy()==========复制序列时删除给定值的元素
remove_copy_if()==========复制序列时删除满足谓词的元素
unique()==========删除相邻的重复元素
unique_copy()==========复制序列时删除相邻的重复元素
reexample()==========反转元素的次序
reexample_copy()==========复制序列时反转元素的次序
rotate()==========循环移动元素
rotate_copy()==========复制序列时循环移动元素
random_shuffle()==========采用均匀分布随机移动元素

6.2.3 序列排序

<algorithm>

sort()==========以很好的平均次序排序
stable_sort()==========排序且维持相同元素原有的顺序
partial_sort()==========将序列的前一部分排好序
partial_sort_copy()==========复制的同时将序列的前一部分排好序
nth_element()==========将第 n 个元素放到它的正确位置
lower_bound()==========找到某个值的第一个出现
upper_bound()==========找到大于某个值的第一个出现
equal_range()==========找出具有给定值的一个子序列
binary_search()==========在排好序的序列中确定给定元素是否存在
merge()==========归并两个排好序的序列
inplace_merge()==========归并两个接续的排好序的序列
partition()==========将满足某谓词的元素都放到前面
stable_partition()==========将满足某谓词的元素都放到前面且维持原顺序

6.2.4 集合算法

<algorithm>

include()==========如果一个序列是另一个的子序列则为真
set_union()==========构造一个已排序的并集

set_intersection()==========构造一个已排序的交集

set_difference()==========构造一个已排序序列，包含在第一个序列但不在第二个序列的元素

set_symmetric_difference()=====构造一个已排序序列，包括所有只在两个序列之一中的元素

6.2.5 堆操作

<algorithm>

make_heap()==========将序列高速得能够作为堆使用

push_heap()==========向堆中加入一个元素

pop_heap()==========从堆中去除元素

sort_heap()==========对堆排序

6.2.6 最大和最小

<algorithm>

min()==========两个值中较小的

max()==========两个值中较大的

min_element()==========序列中的最小元素

max_element()==========序列中的最大元素

lexicographic_compare()========两个序列中按字典序的第一个在前

6.2.7 排列

<algorithm>

next_permutation()==========按字典序的下一个排列

prev_permutation()==========按字典序的前一个排列

6.2.8 通用数值算法

<numeric>

accumulate()==========积累在一个序列中运算的结果(向量的元素求各的推广)

inner_product()==========积累在两个序列中运算的结果(内积)

partial_sum()==========通过在序列上的运算产生序列(增量变化)

adjacent_difference()==========通过在序列上的运算产生序列(与 partial_sum 相反)

6.2.9 C 风格算法

<cstdlib>

qsort()==========快速排序，元素不能有用户定义的构造，拷贝赋值和析构函数

bsearch()==========二分法查找，元素不能有用户定义的构造，拷贝赋值和析构函数

6.3 函数对象

6.3.1 基类

template<class Arg, class Res> struct unary_function

template<class Arg, class Arg2, class Res> struct binary_function

6.3.2 谓词

返回 bool 的函数对象。

<functional>

equal_to==========二元，arg1 == arg2

not_equal_to==========二元，arg1 != arg2

greater==========二元，arg1 > arg2

less==========二元，arg1 < arg2

greater_equal==========二元，arg1 >= arg2

less_equal==========二元，arg1 <= arg2

logical_and==========二元，arg1 && arg2

logical_or==========二元，arg1 || arg2

logical_not==========一元，!arg

6.3.3 算术函数对象

<functional>

plus==========二元，arg1 + arg2

minus==========二元，arg1 - arg2

multiplies==========二元，arg1 * arg2

divides==========二元，arg1 / arg2

modulus==========二元，arg1 % arg2

negate==========一元，-arg

6.3.4 约束器，适配器和否定器

<functional>

bind2nd(y)

binder2nd==========以 y 作为第二个参数调用二元函数

bind1st(x)

binder1st==========以 x 作为第一个参数调用二元函数

mem_fun()

mem_fun_t==========通过指针调用 0 元成员函数

mem_fun1_t==========通过指针调用一元成员函数

const_mem_fun_t==========通过指针调用 0 元 const 成员函数

const_mem_fun1_t==========通过指针调用一元 const 成员函数

mem_fun_ref()

mem_fun_ref_t==========通过引用调用 0 元成员函数

mem_fun1_ref_t==========通过引用调用一元成员函数

const_mem_fun_ref_t========通过引用调用 0 元 const 成员函数

const_mem_fun1_ref_t=======通过引用调用一元 const 成员函数

ptr_fun()

pointer_to_unary_function==调用一元函数指针

ptr_fun()

pointer_to_binary_function=调用二元函数指针

not1()

unary_negate==========否定一元谓词

not2()

binary_negate==========否定二元谓词

6.4 迭代器

6.4.1 分类

Output: *p= , ++

Input: =*p , -> , ++ , == , !=

Forward: *p= , =*p , -> , ++ , == , !=

Bidirectional: *p= , =*p -> , [] , ++ , -- , == , !=

Random: += , -= , *p= , =*p -> , [] , ++ , -- , + , - , == , != , < , > , <= , >=

6.4.2 插入器

template<class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);

template<class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);

template<class Cont, class Out> insert_iterator<Cont> back_inserter(Cont& c, Out p);

6.4.3 反向迭代器

reexample_iterator==========rbegin(), rend()

6.4.4 流迭代器

ostream_iterator==========用于向 ostream 写入

istream_iterator==========用于向 istream 读出

ostreambuf_iterator==========用于向流缓冲区写入

istreambuf_iterator==========用于向流缓冲区读出

6.5 分配器

<memory>

template<class T> class std::allocator

6.6 数值

6.6.1 数值的限制

<limits>

numeric_limits<>

<climits>

CHAR_BIT

INT_MAX

...

<cfloat>

DBL_MIN_EXP

FLT_RADIX

LDBL_MAX

...

6.6.2 标准数学函数

<cmath>

double abs(double)==========绝对值(不在 C 中)，同 fabs()

double fabs(double)==========绝对值
double ceil(double d)==========不小于 d 的最小整数
double floor(double d)==========不大于 d 的最大整数
double sqrt(double d)==========d 在平方根，d 必须非负
double pow(double d, double e)=d 的 e 次幂
double pow(double d, int i)====d 的 i 次幂
double cos(double)==========余弦
double sin(double)==========正弦
double tan(double)==========正切
double acos(double)==========反余弦
double asin(double)==========反正弦
double atan(double)==========反正切
double atan2(double x,double y) //atan(x/y)
double sinh(double)==========双曲正弦
double cosh(double)==========双曲余弦
double tanh(double)==========双曲正切
double exp(double)==========指数，以 e 为底
double log(double d)==========自动对数(以 e 为底),d 必须大于 0
double log10(double d)==========10 底对数，d 必须大于 0
double modf(double d,double*p)=返回 d 的小数部分，整数部分存入*p
double frexp(double d, int* p)=找出[0.5,1)中的 x,y,使 d=x*pow(2,y),返回 x 并将 y 存入*p
double fmod(double d,double m)=浮点数余数，符号与 d 相同
double ldexp(double d, int i)==d*pow(2,i)

<cstdlib>

int abs(int)==========绝对值
long abs(long)==========绝对值(不在 C 中)
long labs(long)==========绝对值
struct div_t { implementation_defined quot, rem; }
struct ldiv_t { implementation_defined quot, rem; }
div_t div(int n, int d)========用 d 除 n，返回(商，余数)
ldiv_t div(long n, long d)=====用 d 除 n，返回(商，余数)(不在 C 中)
ldiv_t ldiv(long n, long d)====用 d 除 n，返回(商，余数)

6.6.3 向量算术

<valarray>

valarray

6.6.4 复数算术

<complex>

template<class T> class std::complex;

6.6.5 通用数值算法

见 6.2.8