[**C++ STL快速入门**](https://www.cnblogs.com/skyfsm/p/6934246.html)

在数月之前的机试中第一次体验到STL的威力，因为自己本来一直在用C语言做开发，很多数据结构都是自己造的，比如链表、队列等，第一次接触C++ STL后发现这些数据结构都已经给我提供好了，我直接拿去调用就好了，真是超级方便。最近的项目中也遇到了STL一些容器，所以现在自己好好总结一下STL中一些最常用的容器的使用方法，方便自己日后查阅。

C++ STL中最基本以及最常用的类或容器无非就是以下几个：

* string
* vector
* set
* list
* map

下面就依次介绍它们，并给出一些最常见的最实用的使用方法，做到快速入门。

**string**

首先看看我们C语言一般怎么使用字符串的

char\* s1 = "Hello SYSU!"; //创建指针指向字符串常量，这段字符串我们是不能修改的

//想要创建 可以修改的字符串，我们可以使用数组分配空间

char s2[20] = "Hello SYSU!";

//或者这样

char s3[] = "Hello SYSU!";

//当然我们也可以动态分配内存

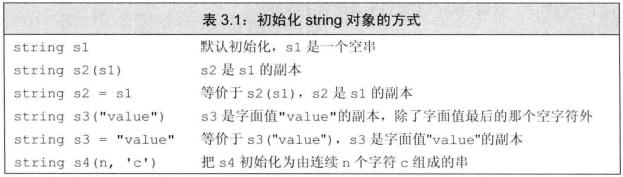
char\* s4 = (char\*)malloc(20）;

gets(s4);

C++ 标准库中的string表示可变长的字符串，它在头文件string里面。

#include <string>

用string初始化字符串分两类：用“=”号就是拷贝初始化，否则就是直接初始化。



string s1;//初始化字符串，空字符串

string s2 = s1; //拷贝初始化，深拷贝字符串

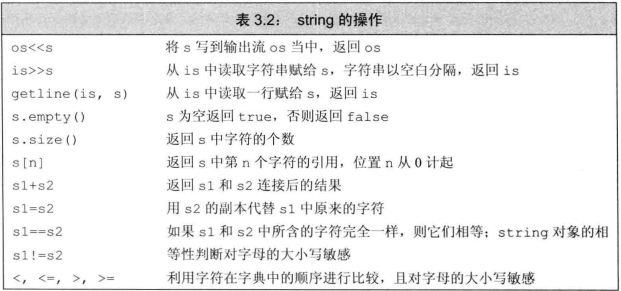
string s3 = "I am Yasuo"; //直接初始化，s3存了字符串

string s4(10, 'a'); //s4存的字符串是aaaaaaaaaa

string s5(s4); //拷贝初始化，深拷贝字符串

string s6("I am Ali"); //直接初始化

string s7 = string(6, 'c'); //拷贝初始化，cccccc



#include <iostream>

#include <string>

using namespace std;

int main()

{

string s1;//初始化字符串，空字符串

string s2 = s1; //拷贝初始化，深拷贝字符串

string s3 = "I am Yasuo"; //直接初始化，s3存了字符串

string s4(10, 'a'); //s4存的字符串是aaaaaaaaaa

string s5(s4); //拷贝初始化，深拷贝字符串

string s6("I am Ali"); //直接初始化

string s7 = string(6, 'c'); //拷贝初始化，cccccc

//string的各种操作

string s8 = s3 + s6;//将两个字符串合并成一个

s3 = s6;//用一个字符串来替代另一个字符串的对用元素

cin >> s1;

cout << s1 << endl;

cout << s2 << endl;

cout << s3 << endl;

cout << s4 << endl;

cout << s5 << endl;

cout << s6 << endl;

cout << s7 << endl;

cout << s8 << endl;

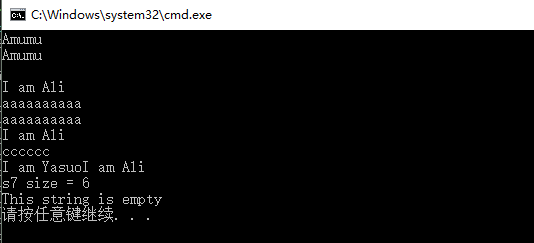
cout << "s7 size = " << s7.size() << endl; //字符串长度，不包括结束符

cout << (s2.empty() ? "This string is empty" : "This string is not empty") << endl;;

system("pause");

return 0;

}



**string的IO操作**

使用cin读入字符串时，遇到空白就停止读取。比如程序输入的是

" Hello World"

那么我们得到的字符串将是"Hello"，前面的空白没了，后面的world也读不出来。

如果我们想把整个hello world读进来怎么办？那就这样做

cin>>s1>>s2;

hello存在s1里，world存在s2里了。

有时我们想把一个句子存下来，又不想像上面那样创建多个string来存储单词，怎么办？

那就是用getline来获取一整行内容。

string str;

getline(cin, str);

cout << str << endl;

当把string对象和字符面值及字符串面值混在一条语句中使用时，必须确保+的两侧的运算对象至少有一个是string

string s1 = s2 + ", "; //正确

string s3 = "s " + ", "; //错误

string s4 = "hello" + ", " + s1; //错误

string s5 = s1 + "hello " + ", "; //改一下顺序，s1放前头，正确了，注意理解=号右边的运算顺序

**处理string中的字符**

访问字符串的每个字符

for (int i = 0; i < s3.size(); i++)

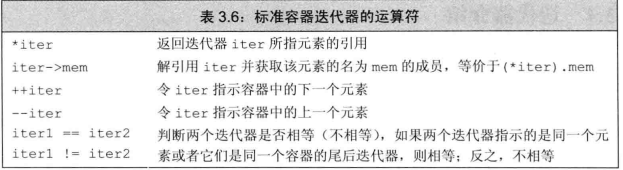
{

cout << s3[i] << endl;

s3[i] = 's';

}

在C语言中我都是用下标或者指针来访问数组元素，而在C++里，有个新奇的东西叫做迭代器iterator，我们可以使用它来访问容器元素。



string str("hi sysu");

for (string::iterator it = str.begin(); it != str.end(); it++)

{

cout << \*it << endl;

}

我们也可以是使用const\_iterator使得访问元素时是能读不能写，这跟常量指针意思差不多。

string str2("hi sysu");

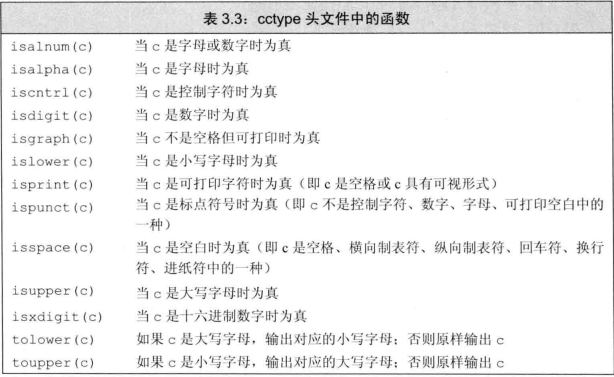
for (string::const\_iterator it = str2.begin(); it != str2.end(); it++)

{

cout << \*it << endl;

\*it = 'l'; //这是错误的，不能写

}



string还有一些很好用的函数，比如找子串

string sq("heoolo sdaa ss");

cout << s.find("aa", 0) << endl; //返回的是子串位置。第二个参数是查找的起始位置，如果找不到，就返回string::npos

if (s.find("aa1", 0) == string::npos)

{

cout << "找不到该子串！" << endl;

}

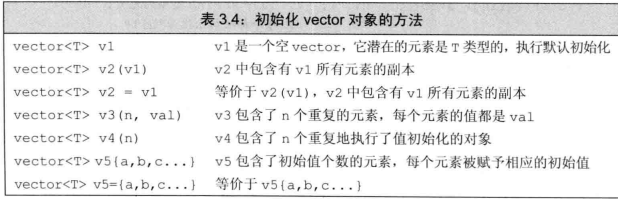
**vector**

C++ STL中的verctor好比是C语言中的数组，但是vector又具有数组没有的一些高级功能。与数组相比，vector就是一个可以不用再初始化就必须制定大小的边长数组，当然了，它还有许多高级功能。

要想用vector首先得包含头文件vector。

#include <vector>

怎么初始化？



如果vector的元素类型是int，默认初始化为0；如果vector元素类型为string，则默认初始化为空字符串。

vector<int> v1;

vector<father> v2;

vector<string> v3;

vector<vector<int> >; //注意空格。这里相当于二维数组int a[n][n];

vector<int> v5 = { 1,2,3,4,5 }; //列表初始化,注意使用的是花括号

vector<string> v6 = { "hi","my","name","is","lee" };

vector<int> v7(5, -1); //初始化为-1,-1,-1,-1,-1。第一个参数是数目，第二个参数是要初始化的值

vector<string> v8(3, "hi");

vector<int> v9(10); //默认初始化为0

vector<int> v10(4); //默认初始化为空字符串

如何向vector添加元素？

请使用push\_back加入元素，并且这个元素是被加在数组尾部的。

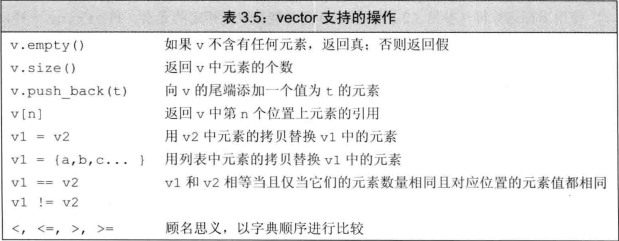
for (int i = 0; i < 20; i++)

{

v1.push\_back(i);

}

vector其他的操作



访问和操作vector中的每个元素

for (int i = 0; i < v1.size(); i++)

{

cout << v1[i] << endl;

v1[i] = 100;

cout << v1[i] << endl;

}

注意：只能对已存在的元素进行赋值或者修改操作，如果是要加入新元素，务必使用push\_back。push\_back的作用有两个：告诉编译器为新元素开辟空间、将新元素存入新空间里。

比如下面的代码是错误的，但是编译器不会报错，就像是数组越界。

vector<int> vec;

vec[0] = 1; //错误！

当然我们也可以选择使用迭代器来访问元素

vector<string> v6 = { "hi","my","name","is","lee" };

for (vector<string>::iterator iter = v6.begin(); iter != v6.end(); iter++)

{

cout << \*iter << endl;

//下面两种方法都行

cout << (\*iter).empty() << endl;

cout << iter->empty() << endl;

}

上面是正向迭代，如果我们想从后往前迭代该如何操作？  
使用反向迭代器

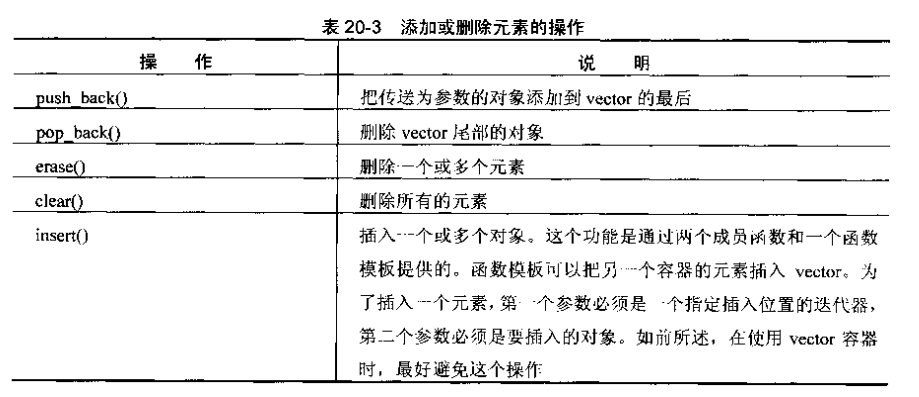
for (vector<string>::reverse\_iterator iter = v6.rbegin(); iter != v6.rend(); iter++)

{

cout << \*iter << endl;

}

vector最常用的增删操作



#include <iostream>

#include <vector>

#include <string>

using namespace std;

template <typename T>

void showvector(vector<T> v)

{

for (vector<T>::iterator it = v.begin(); it != v.end(); it++)

{

cout << \*it;

}

cout << endl;

}

int main()

{

vector<string> v6 = { "hi","my","name","is","lee" };

v6.resize(3); //重新调整vector容量大小

showvector(v6);

vector<int> v5 = { 1,2,3,4,5 }; //列表初始化,注意使用的是花括号

cout << v5.front() << endl; //访问第一个元素

cout << v5.back() << endl; //访问最后一个元素

showvector(v5);

v5.pop\_back(); //删除最后一个元素

showvector(v5);

v5.push\_back(6); //加入一个元素并把它放在最后

showvector(v5);

v5.insert(v5.begin()+1,9); //在第二个位置插入新元素

showvector(v5);

v5.erase(v5.begin() + 3); //删除第四个元素

showvector(v5);

v5.insert(v5.begin() + 1, 7,8); //连续插入7个8

showvector(v5);

v5.clear(); //清除所有内容

showvector(v5);

system("pause");

return 0;

}



注意：虽然vertor对象可以动态增长，但是也或有一点副作用：已知的一个限制就是不能再范围for循环中向vector对象添加元素。另外一个限制就是任何一种可能改变vector对象容量的操作，不如push\_back，都会使该迭代器失效。

总而言之就是：但凡使用了迭代器的循环体，都不要向迭代器所属的容器添加元素！

C++中push\_back和insert两个有什么区别?

顾名思义push\_back把元素插入容器末尾，insert把元素插入任何你指定的位置。  
不过push\_back速度一般比insert快。如果能用push\_back尽量先用push\_back。

**set**

set跟vector差不多，它跟vector的唯一区别就是，set里面的元素是有序的且唯一的，只要你往set里添加元素，它就会自动排序，而且，如果你添加的元素set里面本来就存在，那么这次添加操作就不执行。要想用set先加个头文件set。

#include <set>

#include <iostream>

#include <set>

#include <string>

using namespace std;

template <typename T>

void showset(set<T> v)

{

for (set<T>::iterator it = v.begin(); it != v.end(); it++)

{

cout << \*it;

}

cout << endl;

}

int main()

{

set<int> s1{9,8,1,2,3,4,5,5,5,6,7,7 }; //自动排序，从小到大,剔除相同项

showset(s1);

set<string> s2{ "hello","sysy","school","hello" }; //字典序排序

showset(s2);

s1.insert(9); //有这个值了，do nothing

showset(s1);

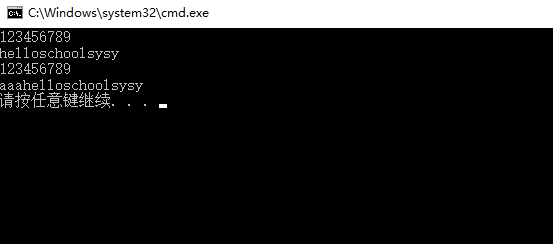
s2.insert("aaa"); //没有这个字符串，添加并且排序

showset(s2);

system("pause");

return 0;

}



**list**

list就是链表，在C语言中我们想使用链表都是自己去实现的，实现起来倒不难，但是如果有现成的高效的链表可以使用的话，我们就不需要重复造轮子了。STL就提供了list容器给我们。

list是一个双向链表，而单链表对应的容器则是foward\_list。

list即双向链表的优点是插入和删除元素都比较快捷，缺点是不能随机访问元素。

初始化方式就大同小异了，跟vector基本一样。要想用list先加个头文件list。

#include <list>

#include <iostream>

#include <list>

#include <string>

using namespace std;

template <typename T>

void showlist(list<T> v)

{

for (list<T>::iterator it = v.begin(); it != v.end(); it++)

{

cout << \*it;

}

cout << endl;

}

int main()

{

list<int> l1{ 1,2,3,4,5,5,6,7,7 };

showlist(l1);

list<double> l2;

list<char> l3(10);

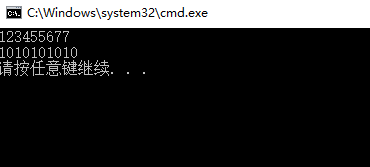
list<int> l4(5, 10); //将元素都初始化为10

showlist(l4);

system("pause");

return 0;

}



值得注意的是，list容器不能调用algorithm下的sort函数进行排序，因为sort函数要求容器必须可以随机存储，而list做不到。所以，list自己做了一个自己用的排序函数，用法如下：

list<int> l1{ 8,5,7,6,1,2,3,4,5,5,6,7,7 };

l1.sort();

**map**

map运用了哈希表地址映射的思想，也就是key-value的思想，来实现的。

首先给出map最好用也最最常用的用法例子，就是用字符串作为key去查询操作对应的value。

要使用map得先加个头文件map。

#include <map>

#include <iostream>

#include <map>

#include <string>

using namespace std;

void showmap(map<string, int> v)

{

for (map<string, int>::iterator it = v.begin(); it != v.end(); it++)

{

cout << it->first << " " << it->second << endl; //注意用法，不是用\*it来访问了。first表示的是key，second存的是value

}

cout << endl;

}

int main()

{

map<string, int> m1; //<>里的第一个参数表示key的类型,第二个参数表示value的类型

m1["Kobe"] = 100;

m1["James"] = 99;

m1["Curry"] = 98;

string s("Jordan");

m1[s] = 90;

cout << m1["Kobe"] << endl;

cout << m1["Jordan"] << endl;

cout << m1["Durant"] << endl; //不存在这个key，就显示0

m1.erase("Curry");//通过关键字来删除

showmap(m1);

m1.insert(pair<string, int>("Harris", 89)); //也可以通过insert函数来实现增加元素

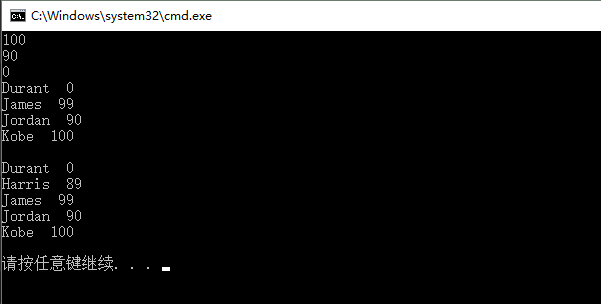
showmap(m1);

m1.clear(); //清空全部

system("pause");

return 0;

}



如果想看看某个存不存在某个key，可以用count来判断

if (m1.count("Lee"))

{

cout << "Lee is in m1!" << endl;

}

else

{

cout << "Lee do not exist!" << endl;

}

用迭代器来访问元素

for (map<string, int>::iterator it = m1.begin(); it != m1.end(); it++)

{

cout << it->first<<" "<<it->second << endl; //注意用法，不是用\*it来访问了。first表示的是key，second存的是value

}

C++ 标准模板库(STL)  
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:

顺序结构   
C++ Vectors   
C++ Lists C++ Double-Ended Queues

容器适配器   
C++ Stacks   
C++ Queues   
C++ Priority Queues   
联合容器   
C++ Bitsets   
C++ Maps   
C++ Multimaps   
C++ Sets   
C++ Multisets   
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.

C++ Bitsets  
C++ Bitsets给程序员提供一种位集合的数据结构。Bitsets使用许多二元操作符，比如逻辑和，或等。

Constructors 创建新bitsets   
Operators 比较和赋值bitsets   
any() 如果有任何一个位被设置就返回true   
count() 返回被设置的位的个数   
flip() 反转bits中的位   
none() 如果没有位被设置则返回true   
reset() 清空所有位   
set() 设置位   
size() 返回可以容纳的位的个数   
test() 返回指定位的状态   
to\_string() 返回bitset的字符串表示   
to\_ulong() 返回bitset的整数表示

C++ Double Ended Queues(双向队列)  
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）。

Constructors 创建一个新双向队列   
Operators 比较和赋值双向队列   
assign() 设置双向队列的值   
at() 返回指定的元素   
back() 返回最后一个元素   
begin() 返回指向第一个元素的迭代器   
clear() 删除所有元素   
empty() 返回真如果双向队列为空   
end() 返回指向尾部的迭代器   
erase() 删除一个元素   
front() 返回第一个元素   
get\_allocator() 返回双向队列的配置器   
insert() 插入一个元素到双向队列中   
max\_size() 返回双向队列能容纳的最大元素个数   
pop\_back() 删除尾部的元素   
pop\_front() 删除头部的元素   
push\_back() 在尾部加入一个元素   
push\_front() 在头部加入一个元素   
rbegin() 返回指向尾部的逆向迭代器   
rend() 返回指向头部的逆向迭代器   
resize() 改变双向队列的大小   
size() 返回双向队列中元素的个数   
swap() 和另一个双向队列交换元素

C++ Lists（链表）  
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.

assign() 给list赋值   
back() 返回最后一个元素   
begin() 返回指向第一个元素的迭代器   
clear() 删除所有元素   
empty() 如果list是空的则返回true   
end() 返回末尾的迭代器   
erase() 删除一个元素   
front() 返回第一个元素   
get\_allocator() 返回list的配置器   
insert() 插入一个元素到list中   
max\_size() 返回list能容纳的最大元素数量   
merge() 合并两个list   
pop\_back() 删除最后一个元素   
pop\_front() 删除第一个元素   
push\_back() 在list的末尾添加一个元素   
push\_front() 在list的头部添加一个元素   
rbegin() 返回指向第一个元素的逆向迭代器   
remove() 从list删除元素   
remove\_if() 按指定条件删除元素   
rend() 指向list末尾的逆向迭代器   
resize() 改变list的大小   
reverse() 把list的元素倒转   
size() 返回list中的元素个数   
sort() 给list排序   
splice() 合并两个list   
swap() 交换两个list   
unique() 删除list中重复的元素

C++ Maps  
C++ Maps是一种关联式容器，包含“关键字/值”对

begin() 返回指向map头部的迭代器   
clear() 删除所有元素   
count() 返回指定元素出现的次数   
empty() 如果map为空则返回true   
end() 返回指向map末尾的迭代器   
equal\_range() 返回特殊条目的迭代器对   
erase() 删除一个元素   
find() 查找一个元素   
get\_allocator() 返回map的配置器   
insert() 插入元素   
key\_comp() 返回比较元素key的函数   
lower\_bound() 返回键值>=给定元素的第一个位置   
max\_size() 返回可以容纳的最大元素个数   
rbegin() 返回一个指向map尾部的逆向迭代器   
rend() 返回一个指向map头部的逆向迭代器   
size() 返回map中元素的个数   
swap() 交换两个map   
upper\_bound() 返回键值>给定元素的第一个位置   
value\_comp() 返回比较元素value的函数

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。

back() 返回最后一个元素   
empty() 如果队列空则返回真   
front() 返回第一个元素   
pop() 删除第一个元素   
push() 在末尾加入一个元素   
size() 返回队列中元素的个数

C++ Sets

集合(Set)是一种包含已排序对象的关联容器

begin()  
返回指向第一个元素的迭代器  
clear()  
清除所有元素  
count()  
返回某个值元素的个数  
empty()  
如果集合为空，返回true  
end()  
返回指向最后一个元素的迭代器  
equal\_range()  
返回集合中与给定值相等的上下限的两个迭代器  
erase()  
删除集合中的元素  
find()  
返回一个指向被查找到元素的迭代器  
get\_allocator()  
返回集合的分配器  
insert()  
在集合中插入元素  
lower\_bound()  
返回指向大于（或等于）某值的第一个元素的迭代器  
key\_comp()  
返回一个用于元素间值比较的函数  
max\_size()  
返回集合能容纳的元素的最大限值  
rbegin()  
返回指向集合中最后一个元素的反向迭代器  
rend()  
返回指向集合中第一个元素的反向迭代器  
size()  
集合中元素的数目  
swap()  
交换两个集合变量  
upper\_bound()  
返回大于某个值元素的迭代器  
value\_comp()  
返回一个用于比较元素间的值的函数

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。

操作比较和分配堆栈   
empty() 堆栈为空则返回真   
pop() 移除栈顶元素   
push() 在栈顶增加元素   
size() 返回栈中元素数目   
top() 返回栈顶元素

C++ Vectors  
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。  
Constructors 构造函数   
Operators 对vector进行赋值或比较   
assign() 对Vector中的元素赋值   
at() 返回指定位置的元素   
back() 返回最末一个元素   
begin() 返回第一个元素的迭代器   
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）   
clear() 清空所有元素   
empty() 判断Vector是否为空（返回true时为空）   
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)   
erase() 删除指定元素   
front() 返回第一个元素   
get\_allocator() 返回vector的内存分配器   
insert() 插入元素到Vector中   
max\_size() 返回Vector所能容纳元素的最大数量（上限）   
pop\_back() 移除最后一个元素   
push\_back() 在Vector最后添加一个元素   
rbegin() 返回Vector尾部的逆迭代器   
rend() 返回Vector起始的逆迭代器   
reserve() 设置Vector最小的元素容纳数量   
resize() 改变Vector元素数量的大小   
size() 返回Vector元素数量的大小   
swap() 交换两个Vector

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

begin() 返回指向第一个元素的迭代器   
clear() 删除所有元素   
count() 返回一个元素出现的次数   
empty() 如果multimap为空则返回真   
end() 返回一个指向multimap末尾的迭代器   
equal\_range() 返回指向元素的key为指定值的迭代器对   
erase() 删除元素   
find() 查找元素   
get\_allocator() 返回multimap的配置器   
insert() 插入元素   
key\_comp() 返回比较key的函数   
lower\_bound() 返回键值>=给定元素的第一个位置   
max\_size() 返回可以容纳的最大元素个数   
rbegin() 返回一个指向mulitmap尾部的逆向迭代器   
rend() 返回一个指向multimap头部的逆向迭代器   
size() 返回multimap中元素的个数   
swap() 交换两个multimaps   
upper\_bound() 返回键值>给定元素的第一个位置   
value\_comp() 返回比较元素value的函数

C++ MultiSets

多元集合(MultiSets)和集合(Sets)相像，只不过支持重复对象。

begin()  
返回指向第一个元素的迭代器  
clear()  
清除所有元素  
count()  
返回指向某个值元素的个数  
empty()  
如果集合为空，返回true  
end()  
返回指向最后一个元素的迭代器  
equal\_range()  
返回集合中与给定值相等的上下限的两个迭代器  
erase()  
删除集合中的元素  
find()  
返回一个指向被查找到元素的迭代器  
get\_allocator()  
返回多元集合的分配器  
insert()  
在集合中插入元素  
key\_comp()  
返回一个用于元素间值比较的函数  
lower\_bound()  
返回指向大于（或等于）某值的第一个元素的迭代器  
max\_size()  
返回集合能容纳的元素的最大限值  
rbegin()  
返回指向多元集合中最后一个元素的反向迭代器  
rend()  
返回指向多元集合中第一个元素的反向迭代器  
size()  
多元集合中元素的数目  
swap()  
交换两个多元集合变量  
upper\_bound()  
返回一个大于某个值元素的迭代器  
value\_comp()  
返回一个用于比较元素间的值的函数

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。

empty() 如果优先队列为空，则返回真   
pop() 删除第一个元素   
push() 加入一个元素   
size() 返回优先队列中拥有的元素的个数   
top() 返回优先队列中有最高优先级的元素