**练习练习练习，熟能生巧。practice makes perfect.**

**巧妙的算法令一切变得不同。**

**实现算法要考虑三个问题：1.正确性 2.复杂度 3.可否优化**

**想办法把做过的题目代码保存起来**

有自己想不到的很棒方法的题目 以及 不擅长的题目 序号：不擅长处理二进制串问题

//695(方法不够好) //526(做不出) //413(方法不够好) //647(方法不够好) //462(想不到) //696(想不到，好题) //667(想不到，很棒的题) //448(方法不够好) //653(中序遍历以后，头尾相加，逐渐向中间靠拢) //712(dp，做不出) //732(做不出，非常棒的一道题)

//739(好题) //672(烂题) //477(做不出，好题) //503(方法不够好) //421(O(n)复杂度做不出，需要用到trie) //46(递归生成全排列，好题) //401(方法不够好，参考按位操作方法)

//894(好题) //200(好题)

vector没有.length()，string有，但是size()两者都有，更加通用一些

stack有.pop()和.push()， 而vector相应的成员函数是.push\_back()和.pop\_back()

queue的基本操作有：

1.入队：如q.push(x):将x元素接到队列的末端；

2.出队：如q.pop() 弹出队列的第一个元素，并不会返回元素的值；

3,访问队首元素：如q.front()

4,访问队尾元素，如q.back();

5,访问队中的元素个数，如q.size();

初始化方法 int a = b = 0; 是错误的，因为b没有定义。

没有初始化的时候值是未知的。

struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

TreeNode\* root = new TreeNode(nums[max\_i]);

这个语句中，root是一个结构体指针，指向new出来的那个结构体

在使用t1->left 或者t1->right的时候必须要先确保t1本身不是一个null指针！

给vector排序的函数是什么？怎么用升序？降序？

sort( V.begin() , V.end() ); 这是对vector<int> V的升序排序

sort记得要包含<algorithm>头文件，vector，map/multimap，stack都要包含头文件

int如何转为string？

使用itoa(int)

或者使用stringstream ss; s << a(此处注意箭头的方向！); string s = a.str();

\*\*\*！！！可以用to\_string(int a)，包括double，float之类的都可以

string如何转为int？

使用atoi(const char\*)

注意先用string类的.c\_str()方法化为c字符串，然后使用atoi

vector取末尾的元素：

1.用.back()方法

2.用迭代器，iter = v.end() – 1 \*iter即是最后一个元素

vector的erase用法(两种重载形式)：

iterator erase(   iterator \_Where);

iterator erase(   iterator \_First,   iterator \_Last);

如果是删除指定位置的元素时：

返回值是一个迭代器，指向删除元素下一个元素;

如果是删除某范围内的元素时：返回值也表示一个迭代器，指向最后一个删除元素的下一个元素;

vector清空所有元素：用.clear()方法

vector对于赋值预算符支持良好，但是对于动态分配的类构成的vector数组，

博主认为一定要重写析构函数与复制构造函数以及运算符重载“=”运算符

string 的操作方法

倒转方法：std::string str = "song";

reverse(str.begin(), str.end());

子字符串方法：原型：string substr ( start, length ) const;

功能：获得子字符串。

参数说明：start为起始位置（默认为0），n为子串长度（默认为总长）

string的compare方法： a.compare(b)，如果a<b，返回值<0，a>b，返回值>0

break跳出循环，不再执行循环内的任何语句

continue执行下一次循环，执行for里面的增量表达式

int isalpha(char c) //用来检测一个字符是不是字母（包括大小写）

int isdigit(char c) //用来检测一个字符是不是数字（从0到9）

int isalnum(char c) //检测一个字符是否为数字或者字母（isalpha与isdigit的结合）

int isupper(char c) \ islower(char c) //检测是否大\小写字母

int toupper(char c) //把一个字母化为大写(tolower则为小写)

以上函数如果结果为真，返回非零，否则返回零，参数写成char c 与int c其实没有差别

**strcmp函数**

两个字符串自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇’\0’为止。如：

strcmp(“abcd”,”abcd”)的返回值是 0；

strcmp(“abcd”,”dcba”)的返回值是 -1；

strcmp(“dcba”,”abcd”)的返回值是 1；

find()方法：string s = “hello”; s.find(“xxx”); //如果找到了，返回**第一次出现**的位置(下标)，如果找不到，返回的是string::npos，一个比string对象任何合法位置都要大的一个常量。

erase()方法： s.erase(4, 5); //删掉从位置4开始，长度为5的子字符串

s.erase(4); //删掉从位置4开始，一直到结尾的子字符串

s.erase(); //等于s.clear();

str.substr(0, length) 从位置0开始，往后挪到子字符串长度是length为止，如果不加length，就是从位置0开始挪到末尾。

在string s后面添加一个char ch，可以直接s += ch;

stringstream用法是什么？？

字符串流，需要包含头文件<sstream>，可以识别负数，不符合规则的不能识别

stringstream ss;

int i1 = 0, i2 = 0;

string a1 = “1”, a2 = “1”;

ss << a1; //吸取a1的值 (可以把<< 跟 >> 视作漏斗，便于理解记忆)

ss >> i1; //将a1的值转化到i1里面

ss.clear(); //同一个stringstream对象，多次转换的时候必须记得clear，不然无效！

ss << a2;

ss >> i2;

cout << i1 << " " << i2 << endl;

输出来的会是1，1

push\_back() 是插入到容器的尾部

if() 里面的条件是等于的时候，一定要记得是 == ！！！你如果打成一个等号的话会默认一定为真！！！

传递指针本质上是值传递，指针自身是一个变量，对被调函数的形参指针做修改，影响不到主调函数的指针！！

vector容器作为函数参数传递方式：

void 函数名( vector< int> obj ); //按值传递

void 函数名( vector< int>\* pobj ); //指针传递

void 函数名( const vector< int>\* pobj ); // 在函数内不能让pobj 指向别的对象

void 函数名( vector< int>& obj ); //按引用传递

void 函数名( const vector< int>& obj ); // 在函数内不能改变 obj 对象

在C++中，下面三种情况下，对象需要调用拷贝构造函数：

1) 一个对象作为函数参数，以值传递的方式传入函数体；

2) 一个对象作为函数返回值，以值传递的方式从函数返回；

//return 是用 return 后面的表达式的值初始化一个返回类型的临时变量（如果返回类型是引用，则不是值传递）

3) 一个对象用于给另外一个对象进行初始化（常称为赋值初始化）；

快速初始化一个二维数组：

bool gridgrid[a + 2][b + 2];

memset(gridgrid, false, sizeof(bool) \* (a + 2) \* (b + 2));

或者memset(gridgrid, false, sizeof(gridgrid));

多维数组作为参数的时候，除了第一维可以不指定长度，其余的必须指定

例如count(int a[ ][10])

变量不可以用来声明一个数组，但是可以用const int a = 10，int arr[a][a]

map<key, value> 插入的时候会根据key自动排序成升序，key默认不重复

相关容器：multimap与unordered\_map

map容器的特性：动态分配内存，键唯一，通过主键引用元素，一个主键映射一个元素

要#include<map> 声明： map<int, string> person;

插入： person[1] = “zby” , person[2] = “abc”

map快速初始化方法：

unordered\_map<char, int> T = { { 'I' , 1 },

{ 'V' , 5 },

{ 'X' , 10 },

{ 'L' , 50 },

{ 'C' , 100 },

{ 'D' , 500 },

{ 'M' , 1000 } };

begin()和end()两个成员，分别代表map对象中第一个条目和最后一个条目，这两个数据的类型是iterator.

无序映射表（Unordered Map）容器是一个存储以键值对组合而成的元素的关联容器（Associative container），容器中的元素无特别的次序关系。该容器允许基于主键地快速检索各个元素，其搜索、插入和元素移除拥有平均常数时间复杂度。（无序容器通过 hash 表来组织它们的元素，允许通过主键快速地访问元素）

查找map中是否包含某个关键字条目用find()方法，返回数据所在位置的迭代器，如果没有找到，返回end的位置。或者可以用count方法，对于非multimap，count只能是0或者是1，因为键不允许重复，此方法缺点是不能定位。

//遍历：

map<string,CAgent>::iterator iter;

 for(iter = m\_AgentClients.begin(); iter != m\_AgentClients.end(); ++iter)

cout<<iter->first<<' '<<iter->second<<endl; //first是key，second是value

//判断key是否出现，两种方法

第一种：用count函数来判定关键字是否出现，其缺点是无法定位数据出现位置,由于map的特性，一对一的映射关系，就决定了count函数的返回值只有两个，要么是0，要么是1，出现的情况，当然是返回1了

第二种：用find函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器，如果map中没有要查找的数据，它返回的迭代器等于end函数返回的迭代器

map[1] += 1; 这句代码是有效的，会自动找到key为1的value，然后把该value加1

（这里要说明一下“.”跟”->”的区别，因为在这里iter是一个迭代器，而不是一个实例化对象，如果是pair<int, int> p，应该用p.first而不是p->first）

\*\*\* c++中当定义类对象是指针对象时候，就需要用到->指向类中的成员；当定义一般对象时候时就需要用到"."指向类中的成员。箭头左边应为指针，点左边应为实体

指针指向一个类的实例，而不是实例本身，实例本身访问成员用点”.”

map中key不存在的时候， map[key]++仍然有效， key对应的value为1

同理，map[key]--也有效，对应value为-1

删除键为bfff指向的元素

cmap.erase("bfff");

pair的声明：

pair<int ,int > p(5,6);

pair<int ,int > p1= make\_pair(5,6);

p.first = 5, p.second = 6

可以自己写一个bool comp函数，然后用vector的sort(v.begin(), v.end(), comp)来排序

sort(v.begin(), v.end(),less<int>());//升

sort(v.begin(), v.end(),greater<int>());//降

static bool comp(<T> a, <T> b) // 在比较的时候，a是后面的元素，b是前面的元素，如果你是在a < b的时候返回true，那么排序后的序列就是升序 a < b < c < d 反之，如果你是在a > b的时候返回true，那么排序后的序列就是降序。（可以这样理解，返回true代表后面的元素需要交换到前面的位置）

vector的repalce用法:

void replace (ForwardIterator first, ForwardIterator last,

const T& old\_value, const T& new\_value)

例子：

int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };

std::vector<int> myvector (myints, myints+8); // 10 20 30 30 20 10 10 20

std::replace (myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99

stl中sort算法有个限制，利用sort算法只能对序列容器进行排序，就是线性的（如vector，list，deque）。map也是一个集合容器，它里面存储的元素是pair，但是它不是线性存储的（前面提过，像红黑树），所以利用sort不能直接和map结合进行排序。虽然不能直接用sort对map进行排序，但是我们可以迂回一下，把map中的元素放到序列容器（如vector）中，然后再对这些元素进行排序：

map<int, int> store;

vector<pair<int, int> > vec(store.begin(), store.end());

sort(vec.begin(), vec.end(), comp);

stray‘\302’ in program是字符空格出现了问题

二叉树的遍历：

dfs深度优先搜索二叉树是先访问根结点，然后遍历左子树接着是遍历右子树，因此我们可以利用堆栈的先进后出的特点，先将右子树压栈，再将左子树压栈，这样左子树就位于栈顶

bfs广度优先搜索，利用队列，把左子树压入队列，再将右子树压入队列。

求一棵二叉树的深度：递归，左子树的高度与右子树的高度的更大者+1

二叉树每一层的节点数：第一层的根后面先插入一个NULL，后面每遇到NULL的时候就说明当前层已经打印完毕，如果队列依然非空，继续往队尾加入一个NULL。

subsequence与substring是不一样的，前者是从字符串里面删掉一些字母，而后者是子串，必须是按原字母相邻关系紧紧相连着的。

声明一个变量的时候一定要初始化，很重要，不然很容易出一些不该出现的错。

在解决关于树的问题的时候，一定要考虑到树根为NULL的情况

在<limits.h>里面有宏定义INT\_MAX, INT\_MIN，分别代表int的最大值和最小值

new一个数组的方法：int \*arr=new int[n]; C++声明任何东西都记得初始化！！！

在一个字符集中，任何一个字符的编码都不是另一个字符编码的前缀。

前缀编码不会出现歧义，哈夫曼编码就是一种前缀编码

c.begin() 返回一个迭代器，它指向容器c的第一个元素

c.end() 返回一个迭代器，它指向容器c的最后一个元素的下一个位置

c.rbegin() 返回一个逆序迭代器，它指向容器c的最后一个元素

c.rend() 返回一个逆序迭代器，它指向容器c的第一个元素前面的位置

反向迭代器是一种反向遍历容器的迭代器。也就是，从最后一个元素到第一个元素遍历容器。反向迭代器将自增（和自减）的含义反过来了：对于反向迭代器，++ 运算将访问前一个元素，而 -- 运算则访问下一个元素。

auto关键字能自动匹配类型

for range循环如下：

std::string str = “hello, world”;

for(auto ch : str) {

     std::cout << ch << std::endl;

} //此处ch自动匹配为char类型，然后range为string的头到尾，逐个输出

//此处修改ch对原字符串不起作用！！

for(auto a: appear) {

if(a.second == 1)

ans.push\_back(a.first);

} //此处appear是一个map，但是遍历的时候不是用的iter，如果是iter的话应该iter->second，但是这里却是a.second ？？

//这个C++11新特性，较老的编译器不支持这种用法。

//记得！auto a的时候，a是按值传递，只有auto &a，a才是按引用传递

<cmath>的一些函数：

ceil(1.499900) is 2.000000 上取整

floor(1.499900) is 1.000000 下取整

round(1.499900) is 1.000000 四舍五入

string全体转为大小写：

string word = “hello”

transform(word.begin(), word.end(), word.begin(), ::toupper); //word会转为”HELLO”

第1，2个参数为数据源容器的起始与终止(迭代器)，第3个为目的容器的起始地址,最后一个参数为操作方法

& 按位与

| 按位或

^ 按位异或

<<左移运算符，如1左移1位以后变为10， >>为右移运算符，1010右移一位为101

想确定一个二进制数某位是否为1，可以先右移到该位到末尾，再与1：(num >> j) & 1

例如想知道10101011的倒数第三位，可以先右移2位，成为101010，再跟1与，看结果是什么。101010 & 000001，结果为0，说明原二进制数倒数第三位是0

深度优先遍历可以分为以下三种：区分方法为什么时候访问根节点

preorder、inorder、postorder：traversal 前序中序后序：遍历

广度优先遍历就是一种，按层次遍历

用stack或者queue都可以实现一棵树的前序遍历

mutimap是个很有用的关联容器，它允许一个键对应多个值

但是因为这样，所以插入的时候不能用下标操作符，而只能用insert方法：

.insert(make\_pair("213.108.96.7","cppzone.com"));

equal\_range(k)：该函数查找所有与 k 关联的值。返回迭代指针的 pair，它标记开始和结束范围：

typedef multimap <string, string>::const\_iterator CIT;

typedef pair<CIT, CIT> Range;

Range range=dns.equal\_range("213.108.96.7");

for(CIT i=range.first; i!=range.second; ++i)

cout << i->second << endl;

count(k)： 成员函数返回与给定键关联的值得数量

lower\_bound() 和 upper\_bound()：lower\_bound(k) 查找第一个与键 k 关联的值，而 upper\_bound(k) 是查找第一个键值比 k 大的元素。

//题目609对于multimap有很好的运用

typedef multimap<string, string>::const\_iterator iter; 可以大大减少代码输入量

一些vector初始化方法：

用数组来初始化：

int arr[] = {1,2,3,4,5};

vector<int> varr(arr, arr+5);

同一元素初始化：

vector<int> fromBegin(n, 1);

memset是按字节赋值的。

memset是内存操作，作用是设置每一个“字节”的值。第一个参数是内存地址，第2个参数是设置的值（0x00 - 0xff),第3个参数是字节数

除了将整型数组设置为0，否则用memset对非字符型数组赋初值是不可取的！！

memset一般用于字符型数组或者布尔数组！！！

（一个字节byte等于8个位bit！！！！！！）

int与float 4字节，char与bool 1字节，double 8字节，在64位的操作系统里面指针大小都是8字节

数组不可越界！越界并不会报错！但是会导致程序崩溃。

引用不是独立的变量，编译系统不给它单独分配存储单元，因此在建立引用时只有声明没有定义，只是声明它与原有的某一变量的关系。

声明与定义的关系？？？

一个别名只能对应一个原始变量，但是一个原始变量可以有多个别名，而且别名也可以

拥有自己的别名

数组名与指针的关系：

(1) 数组名的内涵在于其指代实体是一种数据结构，这种数据结构就是数组；

　　(2) 数组名的外延在于其可以转换为指向其指代实体的指针，而且是一个指针常量；

(3) 指向数组的指针则是另外一种变量类型（在64位平台下，长度为8），仅仅意味着数组的存放地址（与首元素的地址相同）

(4) 数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；

(5) 在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改

int poker[15];

int (\*p)[15] = &poker; //此时的p就是一个int[15]数组的指针

int \*p[15] 则是一个指针数组，里面存放15个指向int类型的指针

如果希望函数只接收特定数量元素的数组，可以考虑用引用形参func (int (&array)[15])

数组作为形参时是讨厌的，可以的话尽量用stl容器。

做题目时容易出现的问题：1.使用while(k--)的时候没有考虑到k在变化，后面用到k的时候出问题，而用for(int i = 0; i < k; i++)就不会出现这种问题

2.ListNode\* cur = root; cur->next = NULL;这个语句会直接导致root->next也变成NULL，因为cur跟root这时候指向同一个表头

3.一些数字边界没弄清楚会导致wrong answer，还有指针为空的问题会导致出现runtime error，程序崩溃。

通用的程序开头：

#include <iostream>

#include <cmath>

#include <cstring>

#include <vector>

#include <map>

#include <algorithm>

#include <limits.h>

using namespace std;

if(i > j+ k) 在括号里面的判断条件，先计算j + k，然后再比较。

C++新型类型转换： 旧版强制转换(char)(10)，C++中风格是static\_cast<type>(content)，C++风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么，更加安全。

static\_cast:可以实现C++中内置基本数据类型之间的相互转换，可以将基类的指针或者引用转为派生类的指针或引用

const\_cast: 不能在不同的种类间转换。它仅仅把一个它作用的表达式转换成常量。它可以使一个本来不是const类型的数据转换成const类型的，或者把const属性去掉。

reinterpret\_cast: 从位(bit)的角度来看待一个对象。它可以转化任何内置的数据类型为其他任何的数据类型，也可以转化任何指针类型为其他的类型。它甚至可以转化内置的数据类型为指针，无须考虑类型安全或者常量的情形。不到万不得已绝对不用。

引用跟指针的区别：1.引用不能为空，指针可以为NULL

2.引用必须初始化，而指针可以不初始化，值为随机

3.引用被初始化以后，在其生命期内只会指向初始化它的对象，无法改变

4.sizeof 引用，得到的是所指向变量的大小；sizeof 指针，得到的是指针的大小；

5.指针++，是指【指针的地址】自增；引用++是指【所指变量】自增；

6.引用是类型安全的，引用过程会进行类型检查；指针不会进行安全检查；

map有find成员函数， 但vector没有find成员函数，但是可以使用泛型find函数，需要加上头文件#include <algorithm>

vector<int>::iterator result = find( L.begin( ), L.end( ), 3 ); 找不到就返回.end()位置

invalid use of non-static member function:

类成员函数跟类成员变量的地址对于每一个实例化的类对象来说都是不同的，但是如果是静态static则是共用一个地址的。

仿函数functor:

class MyClass {

struct Less {

Less(const MyClass& c) : myClass(c) {}

bool operator () ( const int & i1, const int & i2 ) {// use 'myClass'}

MyClass& myClass;

};

doSort() { std::sort(arr,arr+someSize, Less(\*this)); }

}

语法上和函数一样，用起来也跟函数一样，但是它不是函数。

比如 int add(int a, int b) { return a + b; }

你也可以

struct Add {

int operator()(int a, int b) {

return a + b;

}

};

之后就可以声明一个这个类实例：

Add add;

然后就和上面那个add一样用了：

int x = add(1, 2);

仿函数的好处是里面可以带有一些自己的状态。

传统情况下一个函数要有状态，就是里面用static声明变量，不然就只能送一个结构体指针进来，在那个结构体里保存状态。但是现在，因为它不是函数了，它是对象，对象可以有成员变量，于是这些成员变量就用来保存状态。

system(“pause”); //VS2015用来暂停当前行为

runtime error有可能是数组边界出了问题。

moore‘s voting 算法： 从一个数组中找出出现半数以上的元素

reservoir sampling 算法： 给出一个数据流，这个数据流的长度很大或者未知，并且对该数据流中数据只能访问一次，要使所有数据被选中的概率相等

Fisher–Yates shuffle 洗牌算法：

function randomize(A)

n=A.length

for i=1 to n

j=random(i,n)

swap A[i],A[j]

生成随机数函数：

#include <cstdlib>

#include <time.h>

rand()用来获取随机数，但是是伪随机数，默认种子为1，不改变种子的话，每次产生随机数相同。

可以用 srand()来修改种子，多数是改成以时间为种子，srand( (unsigned)time( NULL ) );

快速复制一个数组的方法: memcpy(\* dst, \* src, int size); 把src指向的地址，长度为size字节的内容拷贝到dst指向的地址。

const int M = matrix.size(), N = matrix[0].size();

//当matrix为空时，matrix[0].size()会出现问题，因为你用了null指针

当你要声明多个指针的时候，注意格式，每个变量前都要带\*

ListNode\* l1 = head;

ListNode \*l1 = head, \*l2 = head->next;

用变量名字的时候不要用到一些关键字，比如long，short

枚举类型：

枚举类型是unsigned int型常量的集合，其中的常量称为枚举数

enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};

默认情况下，枚举数的值从0开始，每次加1

difficulty current = EASY; //current被置为1

enum shipCost {FIGHTER\_COST = 25, BOMBER\_COST, CRUISER\_COST = 50};

//此处BOMBER\_COST被初始化为26（前一个枚举数的值加1）

vector的insert用法example:

std::vector<int> myvector (3,100);

std::vector<int>::iterator it; //100, 100, 100

it = myvector.begin();

it = myvector.insert ( it , 200 ); //200, 100, 100, 100

myvector.insert (it,2,300); //300, 300, 200, 100, 100, 100

// "it" no longer valid, get a new one:

it = myvector.begin();

std::vector<int> anothervector (2,400);

myvector.insert (it+2,anothervector.begin(),anothervector.end());

//300, 300, 400, 400, 200, 100, 100, 100

int myarray [] = { 501,502,503 };

myvector.insert (myvector.begin(), myarray, myarray+3);

//501 502 503 300 300 400 400 200 100 100 100

调用push\_back如果发生内存重分配，则所有的迭代器都失效，否则仍有效

迭代器失效的类型：

1.由于容器元素整体“迁移”导致存放原容器元素的空间不再有效，从而使得指向原空间的迭代器失效。

2.由于删除元素使得某些元素次序发生变化使得原本指向某元素的迭代器不再指向希望指向的元素。

//所以，记得insert或者erase以后，更新迭代器！！

//insert可能导致全部失效(重分配)，erase导致被删以及后面元素失效

find()算法包含在algorithm库中

find()算法的参数为起点迭代器，终点迭代器，和要查找的值。所查找的范围从起点开始，到终点迭代器的前一个元素。如果找不到，返回终点迭代器。

capacity()返回vector向量的容量，与size()是不同的概念

程序每次给vector重分配内存时，会增大成为原先的两倍

容器适配器：

STL提供了三种容器适配器：stack, queue, priority\_queue

容器适配器是对容器的一种再封装，它们并不是正式的容器，但是与容器类似(它们并没有提供与元素的保存形式有关的真正数据结构实现，即它们实际上使用底层容器来实现的，并且适配器不支持迭代器)

适配器的优点是：能够使程序员选择一种合适的底层数据结构。

这三个适配器类都提供了成员函数push和pop，能够在每个适配器数据结构中正确地插入和删除元素。

1、 stack适配器

stack类允许在底层数据结构的一端执行插入和删除操作（先入后出）。堆栈能够用任何序列容器实现：vector、list、deque。默认情况下，堆栈是用deque实现的。

2、 queue适配器

queue类允许在底层数据结构的末尾插入元素，也允许从前面删除元素（先入先出）。

队列能够用STL数据结构的list和deque实现，默认情况下队列是用deque实现的。

3、 priority\_queue适配器

priority\_queue类，能够按照有序的方式在底层数据结构中执行插入操作，也能从底层数据结构的前面执行删除操作。

priority\_queue能够用STL的序列容器vector和deque实现。默认情况下使用vector作为底层容器的。当元素添加到priority\_queue时，它们按优先级顺序插入。

这样，具有最高优先级的元素，就是从priority\_queue中首先被删除的元素。通常这是利用堆排序来实现的。堆排序总是将最大值（即优先级最高的元素）放在数据结构的前面。这种数据结构称为heap

默认情况下，元素的比较是通过比较器函数对象less<T>执行的。

为了获取最佳性能，使用vector作为priority\_queue的底层容器

给容器装上适配器的方法：

vector<int> v;

stack< int, vector<int> > s(v);

序列型(顺序型)容器：deque, list, vector

关联型容器:map, set, multimap, multiset

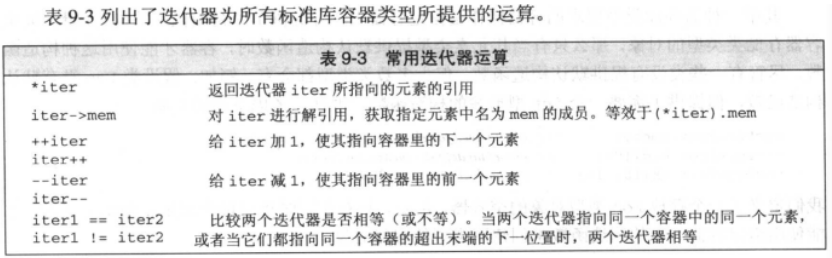
容器适配器:stack, queue, priority\_queue

reserve()函数可以把向量的容量扩充到给定的实参的大小，并且允许程序员自己控制重新分配额外内存的时机。

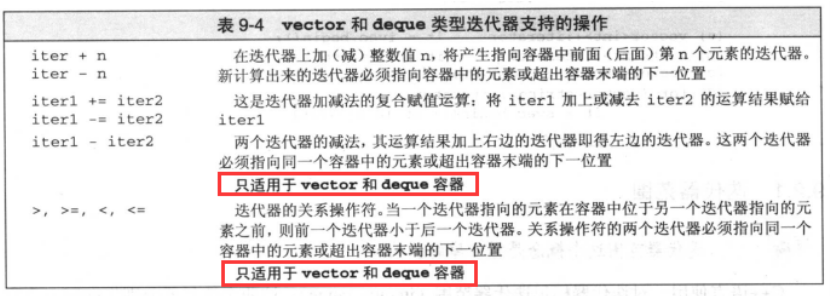
各种容器各有优缺点，要根据自己的需要来选择相应的容器类型。如果序列比较小，或者插入和删除操作比较少，那么最好的选择是使用向量。

容器迭代器的运算类型：

通用型：



vector和deque容器的迭代器提供额外的运算：



list 的迭代器不支持算术运算，也不支持关系运算。它只支持前置和后置的自增、自减运算以及相等（不相等）运算

vector的swap函数本质上并不是真正交换两个容器内部的元素，而是相当于两个容器名字互换(即v1与v2交换以后，v1指向v2，v2指向v1，地址变了值没变)，原先的迭代器都仍然有效，只不过现在指向另一个容器对应的位置(t1原来指向v1.begin()，现在就指向v2.begin()，其值是一样的)，但是注意因为end迭代器不关联任何元素，所以可能会失效(即原先v1.end()的值跟交换以后v2.end()的值可能不一样)

C++中有一些编译器自带函数，如\_\_gcd(int a, int b)求最大公因数，\_\_builtin\_popcount(int a)用来求一个数的二进制里1的个数

顶层const：表示某对象本身是个常量

底层const：表示指针所指对象或者引用所绑定对象是一个常量

可以把一个非常量或者非常量的引用赋给一个常量引用，但不可以把一个常量赋给一个非常量引用（因为非常量引用允许通过其修改引用对象，但被引用对象不允许被修改），也可以让一个指向常量的指针指向一个非常量对象。此时不能通过常量引用和指向常量的指针来修改被引用或被指对象，但可以通过其他的途径来修改。

当一个常量引用被绑定到一个另外的类型时，实际上是绑定了一个临时量。（临时量对象是当编译器需要一个空间来暂存表达式的求值结果时，临时创建的一个未命名的对象）。如果一个临时性对象被绑定于一个reference，对象将残留，直到被初始化之reference的生命结束，或直到临时对象的生命范畴（scope）结束

auto定义的变量必须有初始值，auto可以一条语句中声明多个变量，但是该语句所有变量的初始基本数据类型都必须一样。

踩到的坑：

queue不支持遍历操作，没有迭代器，如果要遍历就不该使用queue这个容器

使用迭代器遍历元素的过程中，如果元素有增加或者减少，迭代器可能会失效

set没有push方法，只有insert方法，insert进去的元素会自动排列，而且不允许有重复，如果insert了一个重复的元素，多余的insert是无效的，set里面仍然只有一个该元素。

如果要使用逆序的迭代器，要用reverse\_iterator iter = v.rbegin()，此时自增自减的含义相反（详见上方笔记）

Vector也有insert函数。

对deque的使用不够熟练。

对链表的使用不够熟练。（ListNode\* t = head，接着对t进行操作，是不是等于对head进行操作？即我创建的t是head的复制品还是head本身，只是head的另一个别名？）

Next\_permutation使用的时候要先排序，从小到大。