1. Map的使用

- 首先是对于map的插入，有一点需要注意，就是通过insert()方法来插入元素的时候，是会先判断要插入的key是否已经存在，如果已经存在，则放弃插入。而如果是通过下标来赋值插入，即m[key]=new_value，会覆盖已经存在的key的元素。

- 使用find函数来查找元素时，如果不存在则返回m.end()，所以判断是否找到应该这样使用：if(m.find(key)!=m.end())

- Map和unordered_map的区别：map内部实现了一个红黑树，该结构具有自动排序的功能，因此map内部的所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素，因此，对于map进行的查找，删除，添加等一系列的操作都相当于是对红黑树进行这样的操作，故红黑树的效率决定了map的效率；而unordered_map：unordered_map内部实现了一个哈希表，因此其元素的排列顺序是杂乱的，无序的。在查找的时候，map的效率要比unordered_map低很多。

- 在map中，由key查找value时，首先要判断map中是否包含key。如果不检查，直接返回map[key]，可能会出现意想不到的行为。如果map包含key，没有问题，如果map不包含key，使用下标有一个危险的副作用，会在map中插入一个key的元素，value取默认值，返回value。也就是说，map[key]不可能返回null。map提供了两种方式，查看是否包含key，m.count(key)，m.find(key)。

2. 使用容器元素、判断奇偶、比较浮点数

- 判断一个整数是否是奇数的时候，用x % 2 != 0，不要用x % 2 == 1，因为x可能是负数。

- 在判断两个浮点数a和b是否相等时，不要用 a == b，应该判断两者之差的绝对值fabs(a-b)是否小于某个阈值，例如 1e-9

- 在需要使用s1.top()之前，一定要先判断下s1是否为空，依次类推在要使用容器的元素的时候，一定先思考下该容器现在的状态是否为空。

3. NULL、0、nullptr之间的区别

C的NULL：在C语言中，我们使用NULL表示空指针，实际上在C语言中，NULL通常被定义为如下：

```
#define NULL ((void *)0)
```

也就是说NULL实际上是一个void *的指针，然后把void *指针赋值给int *和foo_t *的指针的时候，隐式转换成相应的类型。而如果换做一个C++编译器来编译的话是要出错的，因为C++是强类型的，void *是不能隐式转换成其他指针类型的，所以通常情况下，编译器提供的头文件会这样定义NULL：

```
#ifdef __cplusplus ---简称：cpp c++ 文件
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

C++的0：因为C++中不能将void *类型的指针隐式转换成其他指针类型，而又为了解决空指针

的问题，所以C++中引入0来表示空指针（注：0表示，还是有缺陷不完美），这样就有了类似上面的代码来定义NULL。实际上C++的书都会推荐说C++中更习惯使用0来表示空指针而不是NULL。

C++11的nullptr：虽然上面我们说明了0比NULL可以让我们更加警觉，但是我们并没有避免这个问题。这个时候C++ 11的nullptr就很好的解决了这个问题，我们在C++ 11中使用nullptr来表示空指针。如果使用 nullptr 初始化对象，就能避免 0 指针的二义性的问题。

4. max\min函数和INT_MIN\INT_MAX

- max\min函数包含在algorithm头文件中

- INT_MIN\INT_MAX在标准头文件limits.h中定义：

```
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX - 1)
```

使用这两个标志来代表最小整数和最大整数的时候，一定要注意是否存在溢出的问题

- 在涉及到最小值/最大值的时候，一定要考虑到INT_MAX和INT_MIN这两个值，以及max()和min()这两个函数，通过它们可以使代码更加的简单易懂。

5. 判断指针所指向的值之前必须要先判断指针是否为NULL

- 判断二叉树两个结点（p1, p2）的值是否相等的时候，必须按如下顺序进行：

```
if(p1 == NULL && p2 == NULL)     //首先判断p1和p2是否都是NULL
if(p1 == NULL || p2 == NULL)     //然后判断p1和p2之中是否有一个为NULL
if(p1->val == p2->val)           //最后判断p1和p2的值是否相等（此时已经确保p1和p2都不是NULL了）
```

6. 递归和迭代

- 递归和迭代：

1）递归是重复调用函数自身实现循环

2）迭代是函数内某段代码实现循环

3）迭代和普通循环的区别是：迭代时，代码中参与运算的变量同时是保存结果的变量，当前保存的结果作为下一次循环计算的初始值。

7. 位运算的规律总结

一些关于位运算的规律总结：右移永远代表除以二，在不考虑溢出的情况下，左移永远代表乘以二；这里涉及到的一个规律是，二进制负数的左侧实际上有无数个1；二进制正数的左侧实际上有无数个0；

- 与运算：

```
if(a & 1 == 0)   //偶数
if(a & 1 == 1)   //奇数
(n-1) & n        //消去n最后一位的1，可用于统计n比特位上1的个数
~(n-1) & n       //仅保留n最后一位的1，可用于生成相关mask (-n = ~(n-1))
```

- 异或运算：

```
a ^ a = 0;        //相同数异或得0
a ^ 0 = 0;a ^ b ^ a = b    //一个数异或同一个数两次可以还原
//不是用新的变量，交换两个变量的值
//1.基于加减法
a = a + b;
b = a - b;
a = a - b;
//2.基于异或运算
a = a^b;
b = a^b;
a = a^b;
```

- 取反运算：

```
//-1的补码全为1
x + ~x = -1  // 一个数与其反的数相加为-1，推出：-n = ~(n-1)
y + ~y = -1
x + y = 27, 故~x + ~y = -29
```

- 可以通过x&1是否等于1，来判断x的最后一个bit是0还是1
- __builtin_popcount()函数是GCC的一个内建函数，用来计算x中1的个数
- 位逻辑运算符有：&(与)、^(异或)、|(或)、~(取反)
- 位逻辑运算符的优先级高于逻辑运算符，低于比较运算符，且从高到低依次为&、^、|
- 移位运算符的优先级很低，使用时要加括号

8. 一些特殊二进制表示的整数

- 0xaaaaaaaa = 10101010101010101010101010101010 (偶数位为1，奇数位为0)
- 0x55555555 = 01010101010101010101010101010101 (偶数位为0，奇数位为1)
- 0x33333333 = 00110011001100110011001100110011 (1和0每隔两位交替出现)
- 0xcccccccc = 11001100110011001100110011001100 (0和1每隔两位交替出现)
- 0x0f0f0f0f = 00001111000011110000111100001111 (1和0每隔四位交替出现)
- 0xf0f0f0f0 = 11110000111100001111000011110000 (0和1每隔四位交替出现)
- 0x00ff00ff = 00000000111111110000000011111111 (1和0每隔八位交替出现)
- 0xff00ff00 = 11111111000000001111111100000000 (0和1每隔八位交替出现)
- 0x0000ffff = 00000000000000001111111111111111 (1和0每隔十六位交替出现)
- 0xffff0000 = 11111111111111110000000000000000 (0和1每隔十六位交替出现)

9. C++优化之使用emplace

在C++开发过程中，我们经常会用STL的各种容器，比如vector，map，set等，这些容器极大的方便了我们的开发。在使用这些容器的过程中，我们会大量用到的操作就是插入操作，比如vector的push_back，map的insert，set的insert。这些插入操作会涉及到两次构造，首先是对象的初始化

构造，接着在插入的时候会复制一次，会触发拷贝构造。但是很多时候我们并不需要两次构造带来效率的浪费，如果可以在插入的时候直接构造，就只需要构造一次就够了。

因此在C++11中，针对顺序容器(如vector、deque、list)，新标准引入了三个新成员：emplace_front、emplace和emplace_back，这些操作构造而不是拷贝元素。这些操作分别对应push_front、insert和push_back，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

10. C++动态初始化数组

- C++动态初始化数组：一般情况下，声明一个数组的时候是必须要提供一个常量来指明维度的大小，但有时候我们需要动态地来设计数组的大小，遇到这种情况，我们该怎么办呢？
  关于动态数组初始化有如下几点注意事项：
  1）元素只能初始化为元素类型的默认值，而不能像数组变量一样，用初始化列表为数组元素提供各不相同的初始值；
  2）对于内置数据类型元素的数组，必须使用()来显示指定程序执行初始化操作，否则程序不执行初始化操作：

```
int *arr1 = new int[10];          // 每个元素都没有初始化
int *arr2 = new int[10]();        // 每个元素初始化为0
```

  3）对于类类型元素的数组，则无论是否使用()，都会自动调用其构造函数来初始化：

```
string *str1 = new string[10];  // 每个元素调用默认构造函数来初始化
string *str2 = new string[10]() // 每个元素调用默认构造函数来初始化
```

- nth_element()函数：
  头文件： #include<algorithm>
  作用：nth_element作用为求第n大的元素，并把它放在第n位置上，下标是从0开始计数的，也就是说第0小的元素就是最小的数。
  比如：

```
int a[] = {1, 3, 4, 5, 2, 6, 8, 7, 9};
nth_element(a, a+5, a+9);
cout<<"第五大的数："<< a[4]<<endl; // 输出5 下标是从0开始计数的
```

  使用方法：nth_element(a+i, a+k, a+j)，这样它会使a这个数组中区间(i, j)内的第k大的元素处在第k个位置上（相对位置）

  注意：nth_element()函数不过是将第n大的数排好了位置，并不返回值。

11. 优先队列priority_queue和greater\less参数选项

- 关于优先队列priority_queue：优先级队列是一种容器适配器，根据一些严格的弱排序标准，专门设计使其第一个元素始终是它包含的最大元素。

  priority_queue 模板有3个参数，其中两个有默认的参数；第一个参数是存储对象的类型，第二个参数是存储元素的底层容器，第三个参数是函数对象默认是less，它定义了一个用来决定元素顺序的断言。因此模板类型是：

```
template <typename T, typename Container=std::vector<T>, typename
Compare=std::less<T>> class priority_queue
```

  代码例子：

```
#if 1
#include <iostream>
```

```
#include <queue>
using namespace std;
struct Node{
  int x, y;
  Node(int a = 0, int b = 0) :
      x(a), y(b) {}
};
struct cmp{
  bool operator() (Node a, Node b){          //默认是less函数

      //返回true时，a的优先级低于b的优先级（a排在b的后面）
      if (a.x == b.x) return a.y > b.y;
      return a.x> b.x;
  }
};
int main(){
  priority_queue<Node, vector<Node>, cmp> q;
  for (int i = 0; i< 10; ++i)
      q.push(Node(rand(), rand()));
  while (!q.empty()){
      cout << q.top().x << ' ' << q.top().y << endl;
      q.pop();
  }

  system("pause");
  return 0;
}
#endif
```

- greater 和less:在头文件<functional>里面，greater和less都重载了操作符
  我们一般用sort函数的时候，greater和less可以作为函数指针传递下去，不需要单独写比较函数作为函数指针传递给sort函数的第三个参数

```
#if 1
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main(){
    int nums[] = {5, 3, 1, 2, 4};
    int length = sizeof(nums)/sizeof(int);
    std::cout << "nums length is " << length << std::endl;

    sort(nums, nums + length, greater<int>());
    for (int i = 0; i < length; ++i){
        std::cout << nums[i] << "\t";
    }
    std::cout << std::endl;
    sort(nums, nums + length, less<int>());
    for (int i = 0; i < length; ++i){
         std::cout << nums[i] << "\t";
    }
    std::cout << std::endl;
    return 0;
}
#endif
```

12. 处理list中的结点

- 对于单独处理list中的结点的时候，一定要先完全将该结点从list中分离出来。注意，分离的放可以使用node->next = NULL来实现。

- 对于list中结点的反转：node->next->next = node

- 注意：在使用访问node->next->next的时候，一定要先判断node->next是否为nullptr！

13. remove函数、fill函数以及is_sorted函数

- remove(first, last, val):将范围[first，last]转换为另一个范围[first,last_]，其中所有等于val的元素被移除，并将迭代器返回到该范围的新结尾last_，注意：容器的大小并没有发生改变，即[last_, last]还是有元素的。

- fill(first, last, val):将范围[first, last]的元素全都替换为val。

- 判断nums是否已经排好序：is_sorted(nums.begin(), nums.end())

- remove、fill和is_sorted使用：#include<algorithm>