Day 1

C++结构

一个典型的C++程序

```cpp
#include <iostream>

int main(){
    std::cout << "hello world"<< std::endl;
    return 0;
}
```

C++相较于C增加的特性

- 库
- 命名空间（面向对象语言新增的结构）
  - using namespace std;
- 类（面向对象语言新增的结构）
  - std::ios
- ....

命名空间：在大规模软件中，程序需要被分为多个模块进行设计，并相互之间进行调用，因此引入命名空间（namespace）,类似于java中的包（package）。语法： using namespace 命名空间名;
类：面向对象程序设计中抽象个体的结构（class）。
在算法竞赛中，这两项都不会用到，仅作说明。
在调用命名空间或者是类的成员时，需要用到访问运算符“::”，对于命名空间，如果在程序上方已经引用了该命名空间，则可以直接调用其成员，而不需要再通过命名空间逐级访问，但在大型系统设计中，这样容易造成变量冲突。
后续介绍的各种STL中，它们定义在各个头文件里，但由于它们的域限定在std命名空间中，因此程序开头会有一句using namespace std;或者是形如std::cin等。
C++中引用头文件的方式与C不一样：对于C含有的头文件，建议引用方式为去掉后面的.h，在前面加上一个c。如stdio.h->cstdio；对于C++的头文件，有.h和不含.h两种版本，.h为较老版本的C++中的引用方式，现已摒弃。如iostream.h
C++中访问结构体变量的方式与C不同，参见后一页。

C++与C的语法区别

- 库
  - #include <cstdio>
  - #include <iostream>
  - c语言原有的头文件以c开头作为标识，比如stdio.h->cstdio ； 此外c++ 增加了很多新的头文件
- 结构体定义

```
struct node{
    int v;
};

node node1;
```

- ……

引用类型

引用类型指定一个变量的别名，但是引用类型是常量，指定后不可改变，常用来代替函数中的指针

```
int a = 30;
int & p = a;
```

布尔类型

布尔类型表示逻辑的“真”与“假”。
也可用数字赋值，0代表假，非0代表真。

```
bool istrue = true;// true false
```

在C99及之后，C语言中通过引用stdbool.h也可使用bool类型。

运算符也有重载方式，对于自定义的结构体，有时候我们需要实现它的一些运算，方法如下一页：

结构体
运算符重载

```
struct node{
    int value;
    bool operator < (const node & n1) const{
        return value < n1.value;
    }
};
```

STL简介

STL的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但这种分离确实使得STL变得非常通用。例如，由于STL的sort()函数是完全通用的，你可以用它来操作几乎任何数据集合，包括链表，容器和数组。

STL另一个重要特性是它不是面向对象的。为了具有足够通用性，STL主要依赖于模板而不是封装，继承和虚函数（多态性）。你在STL中找不到任何明显的类继承关系。这好像是一种倒退，但这正好是使得STL的组件具有广泛通用性的底层特征。另外，由于STL是基于模板，内联函数的使用使得生成的代码短小高效。

STL的头文件与其他头文件不同，它是“开源”的。如果你想深入了解STL到底是怎么实现的，最好的办法是写个简单的程序，将程序中涉及到的模板源码给复制下来，稍作整理，就能看懂了。

STL组件主要包括容器（Container）、迭代器（Iterator）、算法（Algorithm）等。容器是管理某类对象的集合，迭代器用于在对象集合上的遍历，而算法用于处理集合内的元素。

STL组件都存在于std命名空间中，使用时不要忘记“using namespace std;”。

字符串类

C++中引入了新的字符串类，用于表示字符串，并支持通过运算符操作字符串。

字符串类的方法

```
//构造函数
string s1();  // si = ""
string s2("Hello");  // s2 = "Hello"
string s3(4, 'K');  // s3 = "KKKK"
string s4("12345", 1, 3);  //s4 = "234",
即 "12345" 的从下标 1 开始，长度为 3 的子串
```

```
//求字符串的长度
s1.length();
s1.size();
```

```
//对 string 对象赋值
string s1;
s1 = "Hello";  // s1 = "Hello"
s2 = 'K';  // s2 = "K”
//string 类还有 assign 成员函数，可以用来对 string 对象
赋值。assign 成员函数返回对象自身的引用。例如:
string s1("12345"), s2;
s3.assign(s1);  // s3 = s1
s2.assign(s1, 1, 2);  // s2 = "23", 即 s1 的子串(1, 2)
s2.assign(4, 'K');  // s2 = "KKKK"
s2.assign("abcde", 2, 3);  // s2 = "cde",
即 "abcde" 的子串(2, 3)
```

```
//string对象中字符串的连接
string s1("123"), s2("abc");
s1.append(s2);  // s1 = "123abc"
s1.append(s2, 1, 2);  // s1 = "123abcbc"
s1.append(3, 'K');  // s1 = "123abcbcKKK"
s1.append("ABCDE", 2, 3);  // s1 = "123abcbcKKKCDE", 添加 "ABCDE" 的子串(2, 3)
//string对象的比较
<、<=、==、!=、>=、>
s1.compare(s2);
//小于 0 表示当前的字符串小;
//等于 0 表示两个字符串相等;
//大于 0 表示另一个字符串小。

//例如:
string s1("hello"), s2("hello, world");
int n = s1.compare(s2);
n = s1.compare(1, 2, s2, 0, 3);  //比较s1的子串 (1,2) 和s2的子串 (0,3)
n = s1.compare(0, 2, s2);  // 比较s1的子串 (0,2) 和 s2
n = s1.compare("Hello");
n = s1.compare(1, 2, "Hello");  //比较 s1 的子串(1,2)和"Hello”
n = s1.compare(1, 2, "Hello", 1, 2);  //比较 s1 的子串(1,2)和 "Hello" 的子串(1,2)
```

```
        //求 string 对象的子串
        string s1 = "this is ok";
        string s2 = s1.substr(2, 4);  // s2 = "is i"
        s2 = s1.substr(2);  // s2 = "is is ok"
        //交换两个string对象的内容
        string s1("West"), s2("East");
        s1.swap(s2);  // s1 = "East", s2 = "West"

//删除子串
//erase 成员函数可以删除 string 对象中的子串，返回值为对象自身的引用。例如：
string s1("Real Steel");
s1.erase(1, 3);  //删除子串(1, 3)，此后 s1 = "R Steel"
s1.erase(5);  //删除下标5及其后面的所有字符，此后 s1 = "R Ste"
//插入字符串
//insert 成员函数可以在 string 对象中插入另一个字符串，返回值为对象自身的引用。
例如：
string s1("Limitless"), s2("00");
s1.insert(2, "123");  //在下标 2 处插入字符串"123"，s1 = "Li123mitless"
s1.insert(3, s2);  //在下标 2 处插入 s2 ，s1 = "Li10023mitless"
s1.insert(3, 5, 'X');  //在下标 3 处插入 5 个 'X'，
s1 = "Li1XXXXX0023mitless"
```

```cpp
//将 string 对象作为流处理
//使用流对象 istringstream 和 ostringstream，可以将 string 对象当作一个流进行输入输出。使用这两个类需要包含头文件 sstream。
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    string src("Avatar 123 5.2 Titanic K");
    istringstream istrStream(src); //建立src到istrStream的联系
    string s1, s2; int n;  double d;  char c;
    istrStream >> s1 >> n >> d >> s2 >> c; //把src的内容当做输入流进行读取
    ostringstream ostrStream;
    ostrStream << s1 << endl << s2 << endl << n << endl << d << endl << c <<endl;
    cout << ostrStream.str();
    return 0;
}
```

程序的输出结果是：

```
Avatar
Titanic
123
5.2
K
```

用 STL 算法操作 string 对象
string 对象也可以看作一个顺序容器，它支持随机访问迭代器(后面内容)，也有 begin 和 end 等成员函数。STL 中的许多算法也适用于 string 对象。下面是用 STL 算法操作 string 对象的程序示例。
纯文本复制

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main()
{
    string s("afgcbed");
    string::iterator p = find(s.begin(), s.end(), 'c');
    if (p!= s.end())
        cout << p - s.begin() << endl;  //输出 3
    sort(s.begin(), s.end());
    cout << s << endl;  //输出 abcdefg
    next_permutation(s.begin(), s.end());
    cout << s << endl; //输出 abcdegf
    return 0;
}
```

查找子串和字符

string 类有一些查找子串和字符的成员函数，它们的返回值都是子串或字符在 string 对象字符串中的位置（即下标）。如果查不到，则返回 string::npos。 string::npos 是在 string 类中定义的一个静态常量。这些函数如下：

find：从前往后查找子串或字符出现的位置。

rfind：从后往前查找子串或字符出现的位置。

find_first_of：从前往后查找何处出现另一个字符串中包含的字符。例如：

```
s1.find_first_of("abc");  //查找s1中第一次出现"abc"中任一字符的位置
```

find_last_of：从后往前查找何处出现另一个字符串中包含的字符。

find_first_not_of：从前往后查找何处出现另一个字符串中没有包含的字符。

find_last_not_of：从后往前查找何处出现另一个字符串中没有包含的字符。

replace 成员函数可以对 string 对象中的子串进行替换，返回值为对象自身的引用。例如：

```
string s1("Real Steel");
s1.replace(1, 3, "123456", 2, 4); //用 "123456" 的子串(2,4) 替换 s1 的子串(1,3)
cout << s1 << endl;  //输出 R3456 Steel
string s2("Harry Potter");
s2.replace(2, 3, 5, '0');  //用 5 个 '0' 替换子串(2,3)
cout << s2 << endl;  //输出 Ha00000 Potter
int n = s2.find("00000");  //查找子串 "00000" 的位置，n=2
s2.replace(n, 5, "XXX");  //将子串(n,5)替换为"XXX"
cout << s2 < < endl;  //输出 HaXXX Potter
```

标准输入输出流

```
#include <iostream>

std::cin >> a; 输入一个a
cin >> a >> b; 连续读入两个变量
std::cout << a << “ ” << b <<  std::endl; 输出“a
b”并换行
std::ios::sync_with_stdio(false);  关闭与C的流同步，这样C++
输入的速度可以提高，但关闭后两种输入输出方式不能再混用

istream& getline (char* s, streamsize n );
        iostream的两种读入一行的方式，
istream& getline (char* s, streamsize n, char delim );
#include <string>
    istream& getline (istream&  is, string& str, char delim);
    istream& getline (istream&& is, string& str, char delim);
    istream& getline (istream&  is, string& str);
    istream& getline (istream&& is, string& str);
string中读入一行的方式
例如:string s;
    std:: getline(cin,s);
```

文件重定向

```
#include <cstdio>
    freopen("read.in","r",stdin);
    freopen("write.out","w",stdout);
```

文件重定向可以把标准输入输出重定向到文件里面（对C++的流也适用），上面第一行为输入，第二行为输出
文件重定向可方便调试程序，特别是对于机房的win7计算机每次需要多个步骤去粘贴代码。

STL
STL的算法部分

Standard Template Library，缩写：STL。一个C++软件库，大量影响了C++标准程序库但并非是其的一部分。其中包含4个组件，分别为算法、容器、函数、迭代器。

C++中的标准程序库（Standard Library）是类和函数的集合，其使用核心语言写成。标准程序库的特性声明于std命名空间之中。

模板：C++实现泛型（参数化类型）功能的机制，由函数模板、类模板组成。

以下已默认引用namespace命名空间。

排序算法

```
#include <algorithm>
    int a[100];
    std::sort(a,a+100);//默认升序

    bool cmp(const int& a, const int& b) {
                return a > b; }//为true时不交换
    std::sort(a,a+100,cmp);//逆序

   #include <<functional>
       std::sort(a,a+100,greater<int>()); //升序
       std::sort(a,a+100,less<int>());   //降序
```

运算符重载：小于运算符

稳定的排序：stable_sort

交换算法

```
#include <algorithm>

    template <class T> void swap ( T& a, T& b )
    {
    T c(a); a=b; b=c;
    }
```

例如 int a = 1,b=2;
swap(a,b);//a = 2,b = 1

万能头文件

```
#include <bits/stdc++.h>
```

右为该头文件的内容，适合参加算法竞赛的同学使用。

```
// C++ includes used for precompiling -*- C++ -*-

// Copyright (C) 2003-2014 Free Software Foundation, Inc.
//
// This file is part of the GNU ISO C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
// Free Software Foundation; either version 3, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// Under Section 7 of GPL version 3, you are granted additional
// permissions described in the GCC Runtime Library Exception, version
// 3.1, as published by the Free Software Foundation.

// You should have received a copy of the GNU General Public License and
// a copy of the GCC Runtime Library Exception along with this program;
// see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see
// <http://www.gnu.org/licenses/>.

/** @file stdc++.h
 *  This is an implementation file for a precompiled header.
 */

// 17.4.1.2 Headers

// C
#ifndef _GLIBCXX_NO_ASSERT
#include <cassert>
#endif
#include <cctype>
#include <cerrno>
#include <cfloat>
#include <ciso646>
#include <climits>
#include <clocale>
#include <cmath>
#include <csetjmp>
#include <csignal>
#include <cstdarg>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>

#if __cplusplus >= 201103L
#include <ccomplex>
#include <cfenv>
#include <cinttypes>
#include <cstdalign>
#include <cstdbool>
#include <cstdint>
#include <ctgmath>
#include <cwchar>
#include <cwctype>
#endif

// C++
#include <algorithm>
#include <bitset>
#include <complex>
#include <deque>
#include <exception>
#include <fstream>
#include <functional>
#include <iomanip>
#include <ios>
#include <iosfwd>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <streambuf>
#include <string>
#include <typeinfo>
#include <utility>
#include <valarray>
#include <vector>

#if __cplusplus >= 201103L
#include <array>
#include <atomic>
#include <chrono>
#include <condition_variable>
#include <forward_list>
#include <future>
#include <initializer_list>
#include <mutex>
#include <random>
#include <ratio>
#include <regex>
#include <scoped_allocator>
#include <system_error>
#include <thread>
#include <tuple>
#include <typeindex>
#include <type_traits>
#include <unordered_map>
#include <unordered_set>
#endif
```

模拟算法

模拟算法

有些问题难以找到公式与规律来解决，只能依旧问题的叙述一步一步不停地做下去，才能最终得到结果。这样的问题用计算机来模拟解决是最有效的。

所以计算机模拟算法，即使程序完整的按题目所叙述的方式运行，最终得出答案。

其实，模拟算法也就是将整个过程完完整整的走一遍。

题目怎么叙述的，程序就怎么运行。

例：神奇的幻方

1．神奇的幻方

(magic.cpp/c/pas)

【问题描述】

幻方是一种很神奇的 $N*N$ 矩阵：它由数字 $1,2,3,\ldots\ldots,N*N$ 构成，且每行、每列及两条对角线上的数字之和都相同。

当 N 为奇数时，我们可以通过以下方法构建一个幻方：

首先将 1 写在第一行的中间。

之后，按如下方式从小到大依次填写每个数 $K(K=2,3,\ldots,N*N)$：

1. 若 $(K-1)$ 在第一行但不在最后一列，则将 K 填在最后一行，$(K-1)$ 所在列的右一列；
2. 若 $(K-1)$ 在最后一列但不在第一行，则将 K 填在第一列，$(K-1)$ 所在行的上一行；
3. 若 $(K-1)$ 在第一行最后一列，则将 K 填在 $(K-1)$ 的正下方；
4. 若 $(K-1)$ 既不在第一行，也不在最后一列，如果 $(K-1)$ 的右上方还未填数，则将 K 填在$(K-1)$的右上方，否则将 K 填在 $(K-1)$ 的正下方。

现给定 N，请按上述方法构造 $N*N$ 的幻方。

【输入格式】

输入文件只有一行，包含一个整数 N，即幻方的大小。

【输出格式】

输出文件包含 N 行，每行 N 个整数，即按上述方法构造出的 $N*N$ 的幻方。相邻两个整数之间用单个空格隔开。

【输入输出样例 1】

magic.in	magic.out
3	8 1 6 3 5 7 4 9 2

【数据规模与约定】

对于 100% 的数据，$1 \le N \le 39$ 且 N 为奇数。

```cpp
#include <iostream>

int a[40][40] = { 0 };

int main() {
    int n;
    std::cin >> n;
    int step = 1;      // 填到几了
    int posx, posy; // 上一个点坐标
    while (step <= n * n) {
        if (step == 1)
            a[posx = 1][posy = n / 2 + 1] = step++;
        else if (posx == 1 && posy != n)
            a[posx = n][++posy] = step++;
        else if (posx != 1 && posy == n)
            a[--posx][posy = 1] = step++;
        else if (posx == 1 && posy == n)
            a[++posx][posy] = step++;
        else if (posx != 1 && posy != n)
            if (a[posx - 1][posy + 1] == 0)
                a[--posx][++posy] = step++;
            else
                a[++posx][posy] = step++;
    }
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j)
            std::cout << a[i][j] << " ";
        std::cout << std::endl;
    }
    return 0;
}
```

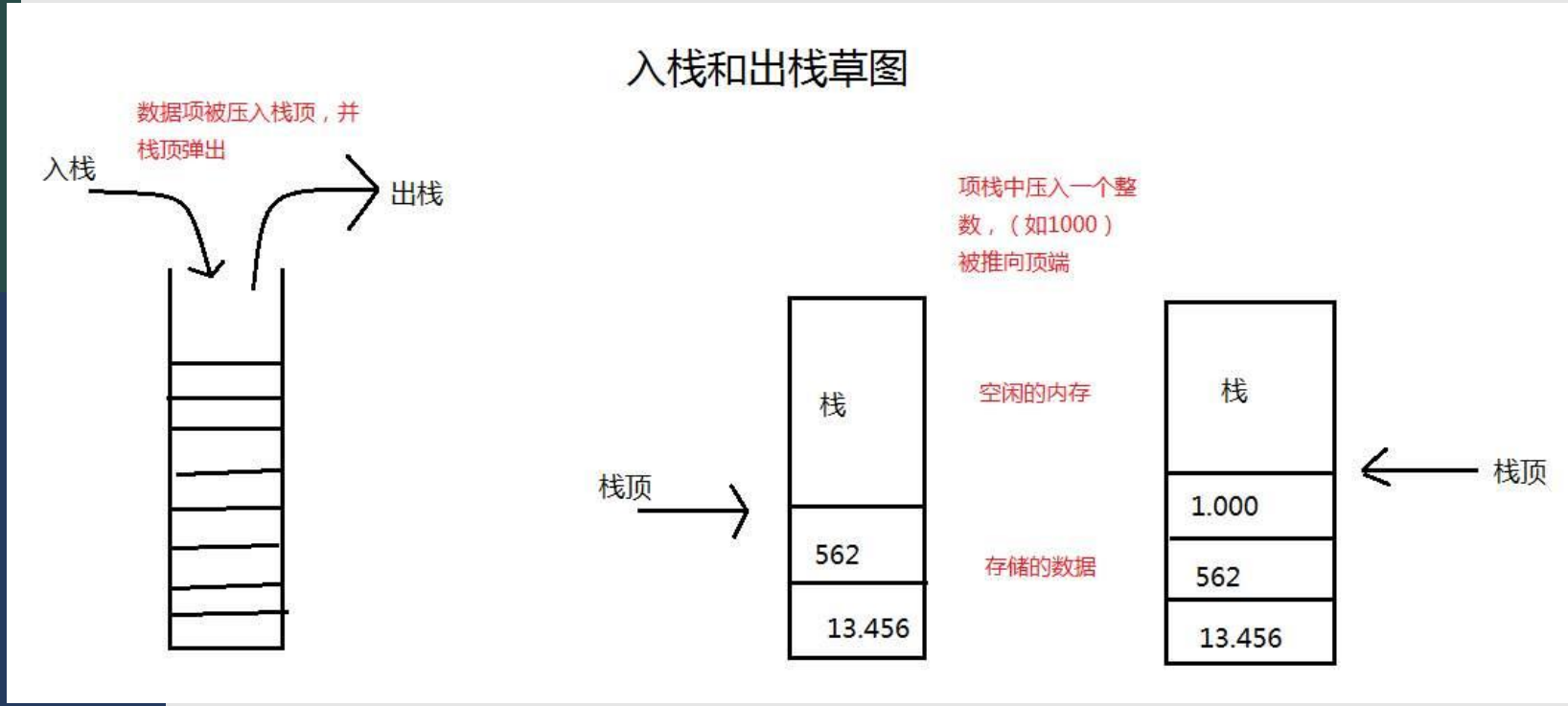
1. 若 $(K-1)$ 在第一行但不在最后一列，则将 K 填在最后一行，$(K-1)$ 所在列的右一列；
2. 若 $(K-1)$ 在最后一列但不在第一行，则将 K 填在第一列，$(K-1)$ 所在行的上一行；
3. 若 $(K-1)$ 在第一行最后一列，则将 K 填在 $(K-1)$ 的正下方；
4. 若 $(K-1)$ 既不在第一行，也不在最后一列，如果 $(K-1)$ 的右上方还未填数，则将 K 填在$(K-1)$的右上方，否则将 K 填在 $(K-1)$ 的正下方。

栈、队列

栈

栈是管理数据的一种结构。体现数据“后进先出”的特性。对于栈和队列，一般不允许访问中间元素。

原理图如下，左为STL实现方式




```cpp
#include <stack>
using namespace std;

stack<int> s;

s.push(1);//入栈
s.pop();//弹出
int stacktop = s.top();//取栈首
bool isempty = s.empty();//栈为空的判断
int stacksize = s.size();//栈内元素数量
```

栈

```
int stack[N], top=0;// top表示栈顶位置。
void push(int a) { stack[top++]=a; }//入栈
int pop() { return stack[--top]; }//出栈
bool empty() { return top<0; }//栈空的条件
```

如果两个栈有相反的需求，可以用这种方法节省空间：用一个数组表示两个栈。分别用top1、top2表示栈顶的位置，令top1从0开始，top2从N-1开始。

队列也是管理数据的一种结构。体现数据“先进先出”的特性，插入在一端，删除在另一端。就像排队一样，刚来的人入队（push）要排在队尾(rear)，每次出队(pop)的都是队首(front)的人。

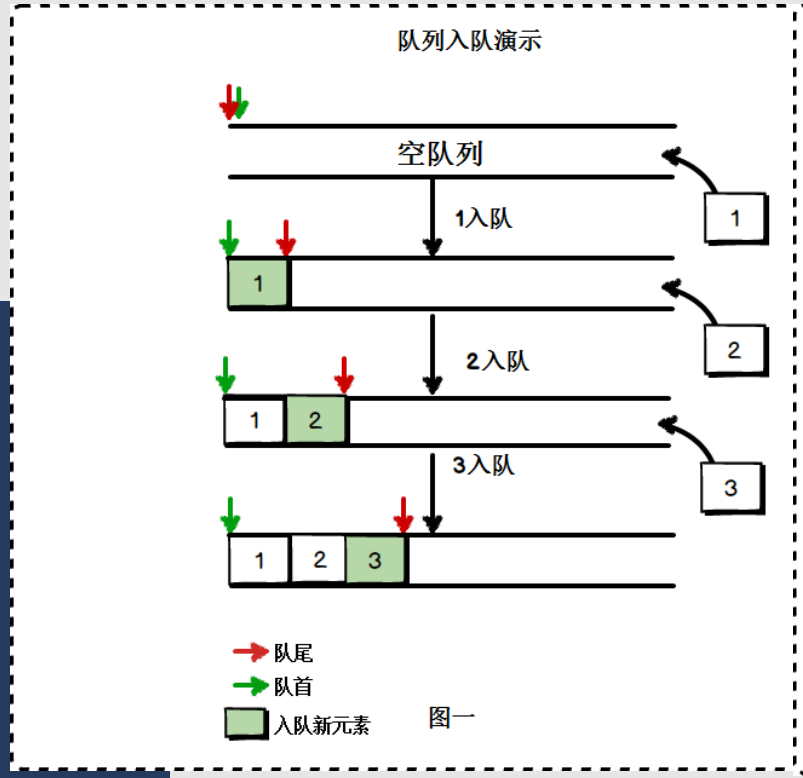
原理图如下，左为STL实现方式

队列

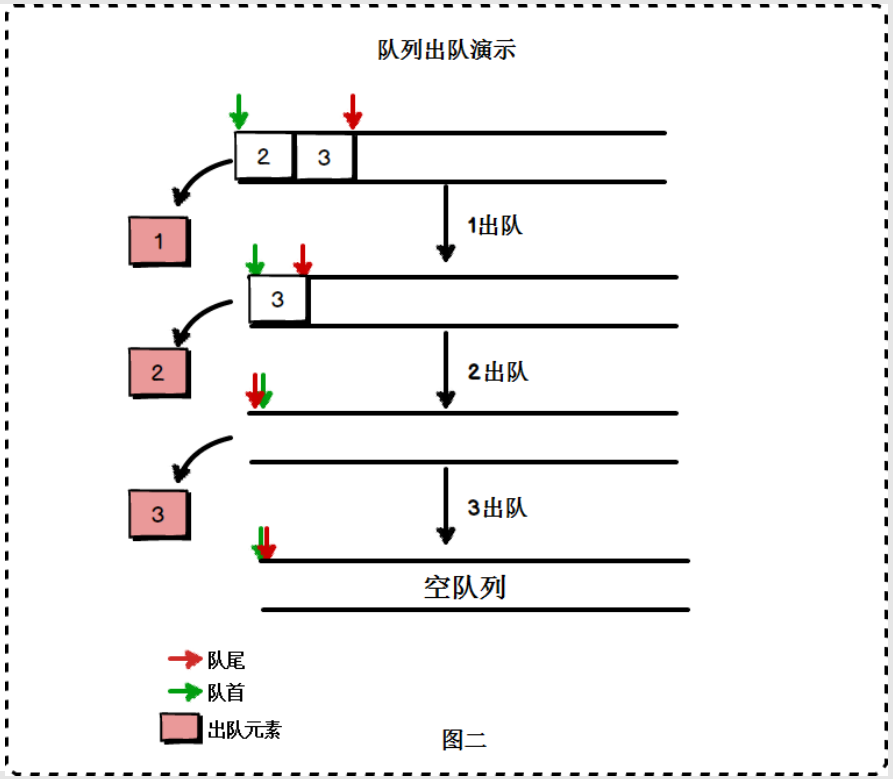
```
#include <queue>
using namespace std;

queue<int> q;

q.push(1);//入队
q.pop();//弹出
int qfront = q.front();//取队首
int qback = q.back();//取队尾
bool isempty = q.empty();//队为空的判断
int qsize = q.size();//队内元素数量
```



图一



图二

队列

顺序队列

```
int queue[N], front=0, rear=0;
//front指向队列首个元素，rear指向队列尾部元素的右侧。
void push(int a) { queue[rear++]=a; }//入队
int pop() { return queue[front++]; }//出队
bool empty() { return front==rear; }//队空的条件
```

循环队列

```
int queue[N], front=0, rear=0;
//front指向队列首个元素，rear指向队列尾部元素的右侧。
void push(int a) { queue[rear]=a; rear=(rear+1)%N; }
//入队
int pop() { int t=queue[front]; front=(front+1)%N; ret
urn t; }//出队
bool empty() { return front==rear; }//队满或队空的条件
```

队满和队空都符合上述条件。怎么把它们区分开呢？

第一种方法：令队列的大小是N+1，然后只使用N个元素。
这样队满和队空的条件就不一样了。
第二种方法：在入队和出队同时记录队列元素个数。这样，
直接检查元素个数就能知道队列是空还是满。

优先队列

本质

优先队列可以按照特定的优先级弹出元素（如按照数据的大小）。

优先队列本质是二叉堆。

二叉堆，是完全二叉树，它可以分为最大值堆和最小值堆。

- 最大（小）值堆中，结点一定不小（大）于两个儿子的值。
- 在堆中，两兄弟的大小没有必然联系。
- 最大（小）值堆的根结点是整个树中的最大（小）值。

STL版本

```cpp
#include <queue>

priority_queue<int> pq;//默认是大顶堆，从大到小排
priority_queue<int, vector<int>, greater<int> >
 pq1;//小顶堆
```

此外可以重载小于运算符，当小于运算符为true时，元素会往后移动

```cpp
pq.push(1);//入队
pq.pop();//出队
pq.top();//返回第一个元素
pq.size();//返回个数
pq.empty();//判断为空
```

STL中常用的容器

STL通用操作

与大小有关的操作

v.size()：返回v中的元素数量。

v.empty()：判断v中元素是否为空。

v.max_size()：返回v能容纳的最大元素数量。

比较：==、!=、<、<=、>、>=

比较操作两端的容器必须是同一类型。比较运算将按字典序比较两个容器元素，当所有元素按序相等时，两容器才相等。

赋值和交换

a=b：用b中元素取代a。

a.swap(b)或swap(a,b)：交换a、b中的元素。

元素操作

v.insert(pos, e)：将元素e的拷贝安插于迭代器pos所指的位置。

v.erase(pos)：移除pos处的元素。

v.erase(begin, end)：移除[begin, end)区间内的所有元素。

v.clear()：移除所有元素。

迭代器

迭代器按照定义方式分成以下四种。

1) 正向迭代器，定义方法如下：

容器类名::iterator 迭代器名;

2) 常量正向迭代器，定义方法如下：

容器类名::const_iterator 迭代器名;

3) 反向迭代器，定义方法如下：

容器类名::reverse_iterator 迭代器名;

4) 常量反向迭代器，定义方法如下：

容器类名::const_reverse_iterator 迭代器名;

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;  //v是存放int类型变量的可变长数组，开始时没有元素
    for (int n = 0; n<5; ++n)
        v.push_back(n);  //push_back成员函数在vector容器尾部添加一个元素
    vector<int>::iterator i;  //定义正向迭代器
    for (i = v.begin(); i != v.end(); ++i) {  //用迭代器遍历容器
        cout << *i << " ";  //*i 就是迭代器i指向的元素
        *i *= 2;  //每个元素变为原来的2倍
    }
    cout << endl;
    //用反向迭代器遍历容器
    for (vector<int>::reverse_iterator j = v.rbegin(); j != v.rend(); ++j)
        cout << *j << " ";
    return 0;
}
```

向量

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。

```cpp
#include<vector>

vector<int> v;

v.push_back(1);//向量尾部增加元素
v.insert(1);//向量前面增加一个元素
v.erase(iterator ite);//删除迭代器指向元素
v.pop_back();//删除向量最后一个元素
v.clear();//清空向量
v.size();//返回向量中元素个数
v.empty();//判断向量为空
v.front();//返回首元素引用
v.back();//返回尾元素引用
//begin end头尾指针
```

映射

映射（map）是关联容器，它按照特定顺序存储由键值和映射值的组合形成的元素。
本质：红黑树

```
#include <map>
map<int,string> m;
```

查询

```
m.at(3)或m[3]//返回一个引用，指向键为3时的对应值。注意，//它和数组下标完全不是一回事儿!
//如果元素不存在，map会自动建立这个元素。
m.count(3)
//返回s中键为3的具体数目。但对于map来说，返回值不是0就是1
m.find(3)//返回指向键为3的元素的迭代器。如果不存在，则返回//m.end()。
m.empty()//判断映射是否为空映射。
m.size()//返回映射的元素数量。
```

插入和删除

```
m.insert(pair)//将元素插入到set中。pair的first是键，second是值
#include <utility>
        可以定义一个pair: pair <int, string> p(10, "Hello");
也可以用make_pair(<utility>)建立一个pair make_pair(10, "Hello");
m.insert(begin, end)//将区间[begin, end)中的值插入到s中。该区间应该是map类型的。
m.erase(e)//将键为e的元素删除。返回值为被删除的e的数量（对于multimap来说，被删除的可能不止一个元素）。
m.erase(pos)//将pos处的元素删除。
m.erase(begin, end)//将[begin, end)处的元素删除。
```

集合

集合（set）是按照特定顺序存储唯一元素的容器。

```
#include <set>
```

定义

```
set< int, less<int> > s; //内部升序排列
set< int, greater<int> > s; //内部降序排列
```

set也可以用预定义的区间来初始化。

查询

```
s.count(10);
//返回s中值为10的具体数目。但对于set来说，返回值不是0就是1。
s.empty();//判断集合是否为空集。
s.size();//返回集合的元素数量。
```

插入和删除

```
s.insert(e);//将e插入到set中。
```

返回值是一个pair，其first是指向插入后元素的迭代器，second表示插入是否成功（如果其second为false，说明元素已经存在）。

```
s.insert(begin, end);
//将区间[begin, end)中的值插入到s中。
s.erase(e);
//将e删除。返回值为e的数量（对于multiset来说，被删除的可能不止一个数）。
s.erase(pos);
//将pos处的元素删除。
s.erase(begin, end);
//将[begin, end)处的元素删除。
```

迭代器：cbegin、cend、crbegin、crend返回只读迭代器。

双端队列

deque和vector类似，但向deque两端添加或删除元素的开销很小。

deque的内存管理比较复杂，随机访问的性能不如vector，插入、删除的性能不如list。如果不需要快速地从容器头部插入和删除数据，最好还是用vector。

不要对deque使用迭代器，因为：

- 在deque中增加任何元素都将使deque的所有迭代器失效。
- 在deque的中间删除元素将使所有的迭代器失效。
- 在deque的头或（d1.begin(),d1.end()）删除元素时，只有指向该元素的迭代器失效。

```
#include<deque>
```

deque特有的操作如下：

```
    d[i]: //返回d中下标为i的元素的引用。
    d.front(): //返回第一个元素的引用。
    d.back(): //返回最后一个元素的引用。
    d.pop_back(): //删除尾部的元素。该函数没有返回值。
    d.pop_front(): //删除头部的元素。该函数没有返回值。
    d.push_back(e): //在尾部添加一个元素e的副本。
    d.push_front(e): //在头部添加一个元素e的副本。
```

表

```
#include <list>

list<int>l;
```

list使用双向链表管理元素。显然list不支持随机存取，也不能对list使用“[]”运算符，但是元素的插入和删除速度很快。

list的其他操作如下：

1. 元素存取
   l.front(): //返回第一个元素。不检查第一个元素是否存在。
   l.back(): //返回最后一个元素。不检查最后一个元素是否存在。
2. 插入元素
   l.insert(pos, e): //在pos位置插入元素e的副本，并返回新元素位置。
   l.insert(pos, n, e): //在pos位置插入n个元素e的副本。
   l.insert(pos, begin, end): //在pos位置插入区间[begin, end)内所有元素的副本。
   l.push_back(e): //在尾部添加一个元素e的副本。
   l.push_front(e): //在头部添加一个元素e的副本。

3. 移除元素
   l.pop_back(): //移除最后一个元素。没有返回值。
   l.pop_front(): //移除第一个元素。没有返回值。
   l.erase(pos): //删除pos位置的元素，返回下一个元素的位置。
   l.erase(begin, end): //删除区间[begin, end)内所有元素，返回下一个元素的位置。
   l.remove(val): //移除所有值为val的元素。
   l.remove_if(op): //移除所有满足“op(val)==true”的元素。
   l.clear(): //移除所有元素，清空容器。
   l.resize(num): //将元素数量改为num（增加的元素用默认构造函数产生，多余的元素被删除）。
   l.resize(num,e): //将元素数量改为num（增加的元素是e的副本）。
4. 其他操作
   l.unique(): //移除重复元素。
   l.unique(op): //移除满足“op(val) == true”的重复元素。
   l1.splice(pos, l2): //将l2内的所有元素转移到l1的迭代器之前。
   l1.splice(pos, l2, l2pos): //将l2内l2pos所指元素转移到l1内的pos之前。
   l1.splice(pos, l2, l2begin, l2end): //将l2内[l2begin, l2end)区间内所有元素转移到l1的pos之前。
   l.sort(): //以operator <为准则对所有元素排序。
   l.sort(op): //以op（定义“小于”关系）为准则对所有元素排序。
   l1.merge(l2): //假设l1和l2都已排序，将l2全部元素转移到l1并保证合并后仍是有序表。
   l.reverse(): //将所有元素反序

无论是安插，还是删除，指向其他元素的指针、引用和迭代器都不会失效。

比较

	vector	deque	list	map/multimap	set/multiset
内部结构	动态数组	数组的数组	双向链表	平衡二叉树	平衡二叉树
元素形式	值	值	值	键—值	值
元素是否可以重复	√	√	√	map：×（键） multimap：√	set：× multiset：√
可随机存取	√	√	×	map：√（键） multimap：×	×
迭代器类型	随机存取	随机存取	双向	双向 键被视为常数	双向 值被视为常数
元素搜寻速度	慢	慢	非常慢	对键来说快	快
在哪里安插、移除速度快	尾部	头尾两端	任何位置	×	×
何时安插、移除会导致迭代器失效	重新分配时	任何时候	不会	不会	不会

高精度

有的时候，数字会大到连long long都不能承受的程度。这时，我们可以自己模拟大数的各种运算。

所谓压位存储，就是在高精度数内部采用10000进制（即每四位放到一个数中）进行存储。它与10进制（即一个数位对应一个数）相比速度要快一些。

高精度数内部也可以采用100000000进制，但是这样就不能计算乘除法了。

编程时这样做——假设hp是高精度类型。

先用宏定义：#define hp long long ，然后集中精力编写算法代码。

最后直接删除这条宏定义，把真正的高精度算法写出来。这样做的好处是无需再修改算法代码，减小了维护代价。

定义

```
const int MAX=100;
struct hp
{
  int num[MAX];

  hp & operator = (const char*);
  hp & operator = (int);
  hp();
  hp(int);

  // 以下运算符可以根据实际需要来选择。
  bool operator > (const hp &) const;
  bool operator < (const hp &) const;
  bool operator <= (const hp &) const;
  bool operator >= (const hp &) const;
  bool operator != (const hp &) const;
  bool operator == (const hp &) const;


  hp operator + (const hp &) const;
  hp operator - (const hp &) const;
  hp operator * (const hp &) const;
  hp operator / (const hp &) const;
  hp operator % (const hp &) const;

  hp & operator += (const hp &);
  hp & operator -= (const hp &);
  hp & operator *= (const hp &);
  hp & operator /= (const hp &);
  hp & operator %= (const hp &);
};
```

```
// num[0]用来保存数字位数。另外，利用10000进制可以节省空间和时间。
hp & hp::operator = (const char* c)
{
  memset(num,0,sizeof(num));
  int n=strlen(c),j=1,k=1;
  for (int i=1;i<=n;i++)
  {
    if (k==10000) j++,k=1;         // 10000进制，4个数字才算1位。
    num[j]+=k*(c[n-i]-'0');
    k*=10;
  }
  num[0]=j;
  return *this;
}

hp & hp::operator = (int a)
{
  char s[MAX];
  sprintf(s,"%d",a);
  return *this=s;
}
hp::hp() {memset(num,0,sizeof(num)); num[0]=1;} // 目的：声明hp时无需显式初始化。
hp::hp (int n) {*this = n;}                     // 目的：支持“hp a=1;”之类的代码。
```

```cpp
// 如果位数不等，大小是可以明显看出来的。如果位数相等，就需要逐位比较。
bool hp::operator > (const hp &b) const
{
  if (num[0]!=b.num[0]) return num[0]>b.num[0];
  for (int i=num[0];i>=1;i--)
    if (num[i]!=b.num[i])
      return (num[i]>b.num[i]);
  return false;
}
bool hp::operator < (const hp &b) const {return b>*this;}
bool hp::operator <= (const hp &b) const {return !(*this>b);}
bool hp::operator >= (const hp &b) const {return !(b>*this);}
bool hp::operator != (const hp &b) const {return (b>*this)||(*this>b);}
bool hp::operator == (const hp &b) const {return !(b>*this)&&!(*this>b);}
```

```
// 注意：最高位的位置和位数要匹配。
hp hp::operator + (const hp &b) const
{
  hp c;
  c.num[0] = max(num[0], b.num[0]);
  for (int i=1;i<=c.num[0];i++)
  {
    c.num[i]+=num[i]+b.num[i];
    if (c.num[i]>=10000) // 进位
    {
      c.num[i]-=10000;
      c.num[i+1]++;
    }
  }
  if (c.num[c.num[0]+1]>0) c.num[0]++;
// 9999+1，计算完成后多了一位
  return c;
}
```

```
hp hp::operator - (const hp &b) const
{
  hp c;
  c.num[0] = num[0];
  for (int i=1;i<=c.num[0];i++)
  {
    c.num[i]+=num[i]-b.num[i];
    if (c.num[i]<0) // 退位
    {
      c.num[i]+=10000;
      c.num[i+1]--;
    }
  }
  while (c.num[c.num[0]]==0&&c.num[0]>1)
       c.num[0]--;  // 100000000-99999999
  return c;
}
```

```
hp & hp::operator += (const hp &b) {return *this=*this+b;}
hp & hp::operator -= (const hp &b) {return *this=*this-b;}
```

```cpp
hp hp::operator * (const hp &b) const
{
  hp c;
  c.num[0] = num[0]+b.num[0]+1;
  for (int i=1;i<=num[0];i++)
  {
    for (int j=1;j<=b.num[0];j++)
    {
      c.num[i+j-1]+=num[i]*b.num[j];      // 和小学竖式的算法一模一样
      c.num[i+j]+=c.num[i+j-1]/10000;     // 进位
      c.num[i+j-1]%=10000;
    }
  }
  while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--;  // 99999999*0

  return c;
}

hp & hp::operator *= (const hp &b) {return *this=*this*b;}
```

```
hp hp::operator / (const hp &b) const
{
  hp c, d;
  c.num[0] = num[0]+b.num[0]+1;
  d.num[0] = 0;
  for (int i=num[0];i>=1;i--)
  {
    // 以下三行的含义是：d=d*10000+num[i];
    memmove(d.num+2, d.num+1, sizeof(d.num)-sizeof(int)*2);
    d.num[0]++;
    d.num[1]=num[i];

    // 以下循环的含义是：c.num[i]=d/b; d%=b;
    while (d >= b)
    {
      d-=b;
      c.num[i]++;
    }
  }
  while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--;
// 99999999/99999999
  return c;
}
```

```
hp hp::operator % (const hp &b) const
{
  ……      // 和除法的代码一样。唯一不同的地方是返回值：
return d;
}
```

```
hp & hp::operator /= (const hp &b) {return *this=*this/b;}
hp & hp::operator %= (const hp &b) {return *this=*this%b;}
```

```
ostream & operator << (ostream & o, hp &n)
{
  o<<n.num[n.num[0]];
  for (int i=n.num[0]-1;i>=1;i--)
  {
    o.width(4);
    o.fill('0');
    o<<n.num[i];
  }
  return o;
}

istream & operator >> (istream & in, hp &n)
{
  char s[MAX];
  in>>s;
  n=s;
  return in;
}
```