# 一. 数据结构

程序 = 数据结构 + 算法

四大逻辑结构——数据元素之间的关系：集合、线性结构、树结构、图形结构。

存储结构：顺序存储、链式存储、索引存储、散列存储。

## 1.线性结构

### 1.1 链表

物理存储单元非连续，其包括2部分：数据域和下一个节点的指针域。优点：便于增删改查。

链表分为：单向链表，双向链表，循环链表。



单向链表：



（1）链表初始化，创建链表（尾插法）

第一步：创建一个头结点ua,分配内存，初始化各元素，注ua->next = NULL；

第二步：创建一个头指针first，让头指针指向头结点（链表存在头指针中）；

第三步：在尾部插入新节点：在循环内先创建一个新节点p，初始化数据域，让其next为空，即p-> next = ua->next;然后让ua的next指向这个新节点ua->next = p;最后让ua = ua->next;整个链表就存储在头指针first中了。

Eg：

First\_ua\* Init\_link(void)

{

First\_ua\* ua = (First\_ua\*)malloc(sizeof(First\_ua)); //头结点

if (ua == NULL)

{

cout << "内存申请失败" << endl;

return 0;

}

ua->appid = 0;

ua->next = NULL;

First\_ua\* first = ua; //头指针

for (int i = 1; i < 6; i++)

{

First\_ua\* p = (First\_ua\*)malloc(sizeof(First\_ua));

if (ua == NULL)

{

cout << "内存申请失败" << endl;

return 0;

}

p->appid = i;

p->next = ua->next;

ua->next = p;

ua = ua->next;

}

return first;

}

（2）链表输出

将链表（也就是指针）作为输入参数时，要首先判是否为空，其次，不能直接判断prt->next是否为空，因为此时prt是否为空你都不清楚。

Eg：

void input\_link(First\_ua\* prt)

{

while (prt != NULL)

{

cout << "appid :" << prt->appid << endl;

prt = prt->next;

}

}

（3）链表插入结点（在链表i位置插入x）

第一步：传入参数为链表或指针时，先将传参赋给新的局部变量；确定链表中的插入前驱结点；

第二步：创建新节点并进行初始化，先让新节点的next指向链表的尾部，然后让链表的next指向新节点。

Eg：

First\_ua\* insert\_link(First\_ua\* prt, int i, int x)

{

First\_ua\* p = prt;

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

{

p = p->next; /\*链表中的插入前驱结点\*/

}

First\_ua\* node = (First\_ua\*)malloc(sizeof(First\_ua));

node->appid = x;

node->next = p->next;

p->next = node;

return prt;

}

（4）链表删除结点同理。

### 1.2 栈

栈（stack）是一种先进后出的数据结构。栈是一种线性结构，且仅允许由一端进行操作。栈分为数组栈和链表栈。

链表栈



以链表的存储方式，使得先读的数据放到栈底，后读的放入栈顶，有一个top指向栈顶这个链表。

入栈方法：

第一步：创建一个链表的结构体，再创建一个用来限制链表的结构体，通常是一个永远指向栈顶的top指针和计数器count；

第二步：先让ua->next指向top;然后让top指向ua;且count++；这样一来，下一次，就将整个ua给了下一个ua的next。

Eg：

Stack\_Ord \* push\_stack(Stack\_Ord \* prt, int elem[], int len)

{

Stack\_Ord \* p = prt;

for (int i = 0; i < len; i++)to

{

First\_UA \*ua = (First\_UA \*)malloc(sizeof(First\_UA));

if (ua == NULL)

{

cout << "内存分配失败";

exit(0);

}

ua->appid = elem[i];

ua->next = p->top;

p->top = ua;

p->count++;

}

return prt;

}

### 1.3 队列

队列是一个先进先出的

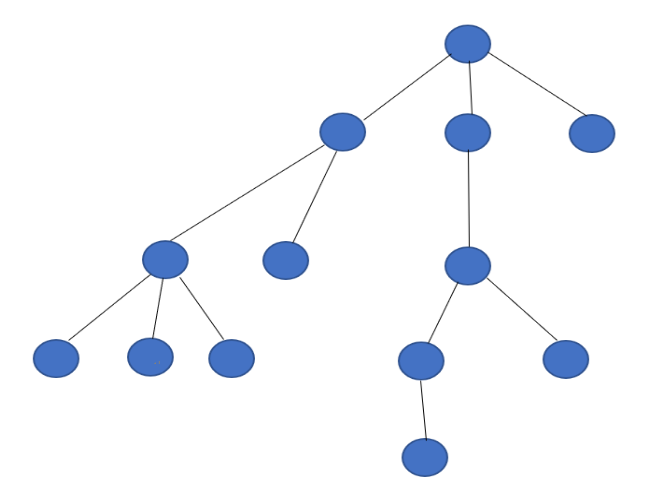
## 2. 树结构

树结构是一种非线性结构。一颗非空树包括一个根节点和多个附加结点，所有节点构成一个多级分层结构。

树的定义：n个结点组成的有限结合。n = 0，空树；n > 0，1个根节点，m个互不相交的有限集，每个子集为根的子树。

结点的层次

第一层



第五层

叶子结点

分支结点

根结点

第二层

第三层

第四层

### 2.1 树的基本术语

（1）结点的度：一个结点所拥有的子树个数；（如上图，根节点的度为3）

（2）树的度：树中各结点度的最大值；（如上图，树的度为3）

（3）分支结点：度不为0的结点；

（4）叶子结点：度为0的结点；

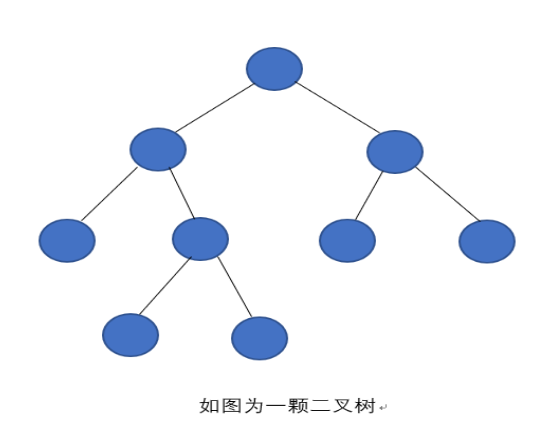
（5）结点的层次：根节点为第一层，以此类推；

（6）树的高度（深度）：树中结点最大的层次；（如上图，树的高度为5）

（7）深林：互不相交的树的集合。

### 2.2 二叉树

二叉树为n（n>=0）个结点的有限集合。该集合或为空集，n=0为空二叉树，或为由一个根节点和2个互不相交的左、右子树组成。二叉树每个结点的度最多为2，即拥有最多2颗子树，且有左右之分，其子树的次序不能颠倒。



（1）二叉树的特点

1）每个结点最多有2颗子树，所以二叉树不存在度大于2的结点；

2）左子树和右子树是有顺序的，次序不能颠倒

3）即使树中某个结点只有一棵树，也是要区分左子树还是右子树。

（2）二叉树性质

1）二叉树第i层的结点最多为2^(i-1)；

2）深度为k的二叉树最多拥有2^k-1个结点；（等比数列求和：a0(1-q^n)/(1-q)）;

3) 包含n个结点的二叉树高度至少为log2(n+1)；

4）任意一颗二叉树中，若叶子结点的个数为n0,则度为2的结点个数为n0-1;

（3）特殊二叉树

1）斜树：所有结点只有左树或右树的二叉树叫左斜树或右斜树。

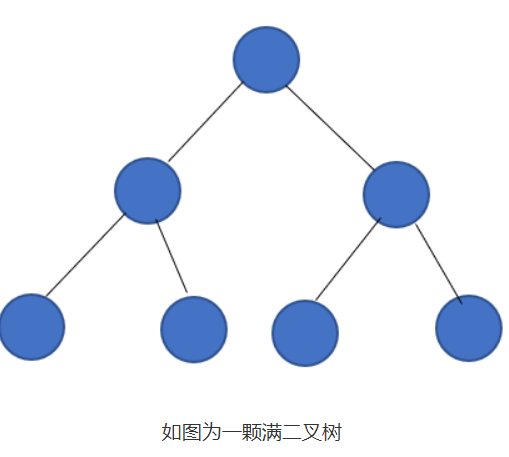
2）满二叉树：在一颗二叉树中，所有分支结点的度都为2，所有叶子结点都在同一层。

满二叉树的特点有：

叶子结点只能出现在最下一层；

所有非叶子结点的度都为2；

在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多。



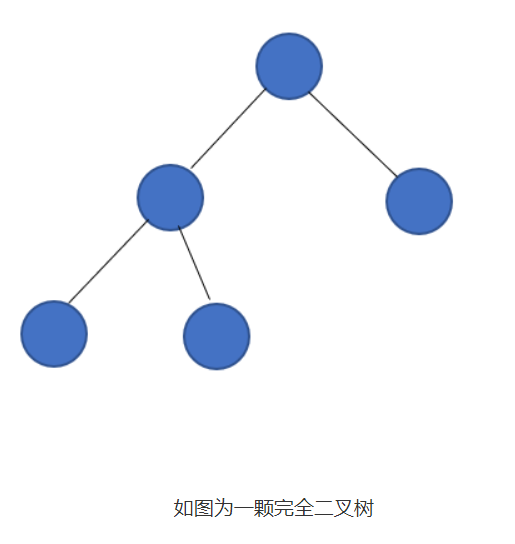
3）完全二叉树：一颗前N个结点与满二叉树的前N个结点位置相同二叉树。

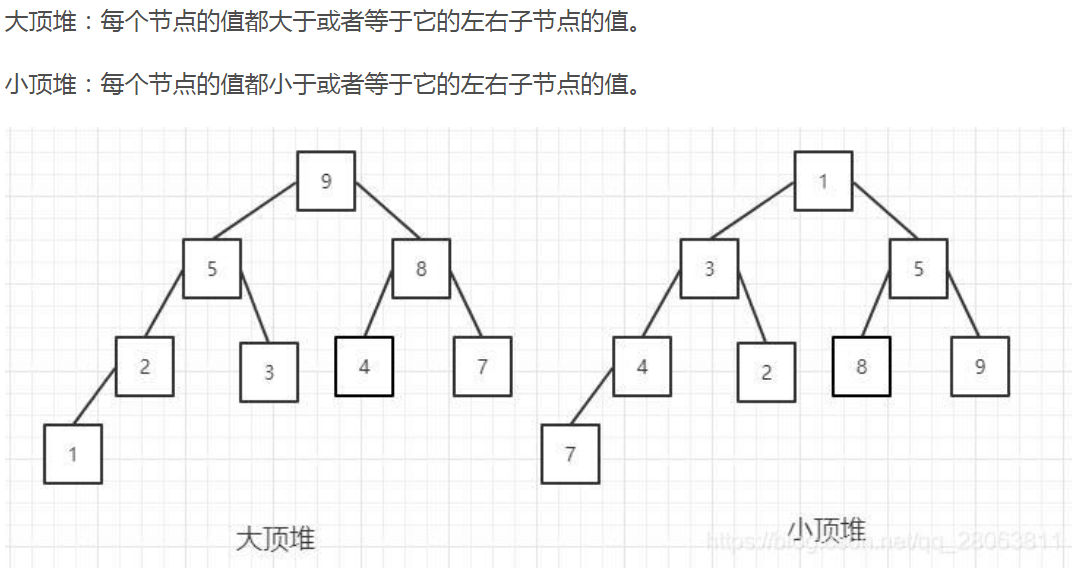
性质：

1如果i > 0，则i结点的双亲结点序号为（i-1）/2(向下取整)，若i = 0，则该结点为根结点；

2 树的结点为n，若2i + 1 < n，则结点i的左孩子结点为2i + 1；若2i + 2 < n，则结点i的右孩子结点为2i + 2；

3 树的结点为n，若2i + 1 >= n，则结点i的无左孩子（无孩子）；若2i + 2 >= n，则结点i的无右孩子；







对于完全二叉树或满二叉树，第一个叶子结点的索引为树的总结点n/2，倒数第一个非叶子结点索引是n/2 -1。（对于上图大顶堆来说，总结点数为8，第一个叶子结点索引8/2=4（根结点索引是0），也就是上图的3，倒数第一个非叶子索引为8/2-1=3，也就是2）

### 2.3 二叉树的存储

树的存储方式分为顺序存储和链式存储。

### 2.4 二叉树的遍历

（1）前序遍历：先根结点，然后左子树，最后右子树（把每一个子节点也看成一棵树）

（2）中序遍历：先左子树，然后根节点，最后右子树；

（3）后序遍历：先左子树，然后右子树，最后根节点。



前序遍历：5,4,1,6,2,9,3,7,8；

中序遍历：6,1,4,2,9,5,3,8,7；

后序遍历：6,1,9,2,4,8,7,3,5；

2.5 二叉树的创建（递归思想）

## 3. 排序

### 3.1冒泡排序（bubble sort）

1）算法时间复杂度

最坏情况：O(n^2)

最好情况：O(n)

平均情况：O(n^2)

2）空间复杂度：S(n)=O(1)

3）稳定性：稳定排序

注：比较相邻元素，如果第二个比第一个大，则交换他们。其中用到了a[i]和a[i+1]，所以循环的时候，n = 数组个数-1；

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

### 3.2简单选择排序

1）算法时间复杂度

最坏情况：O(n^2)

最好情况：O(1) //即不需要排序，本身已是正序

平均情况：O(n^2)

2）空间复杂度：S(n)=O(1)

3）稳定性：不稳定排序

注：相比于冒泡排序的不断交换，简单选择排序在一次循环结束后找到最大（或最小）的index，交换一次就能达到冒泡效果。

### 3.3直接插入排序（从小到大）

1）算法时间复杂度

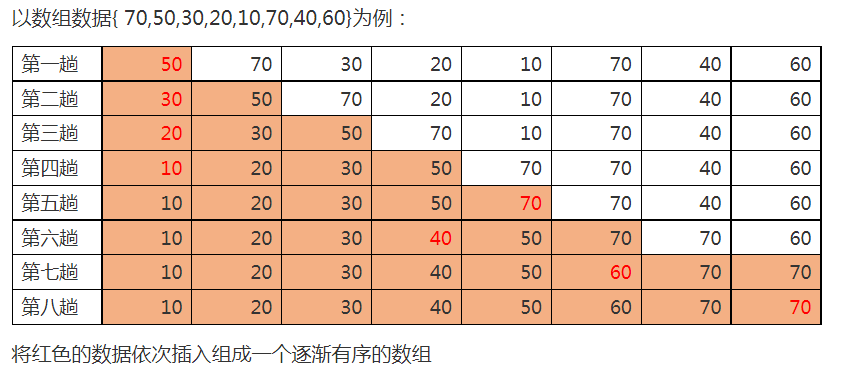
最坏情况：O(N^2)

最好情况：O(N^2)

平均情况：O(N^2)

2）稳定性：稳定排序

注：从无序数组中取第一个，然后与有序数组从后往前作比较，如果比有序数组的最后一个小，则让有序数组的最后一个往后挪一个，这个数再与有序数组中的前一个作比较，以此类推，直到那个数大于它，跳出循环，再将他插入到这个位置。



### 3.4堆排序

1)算法时间复杂度

最坏情况：O(n^2)

最好情况：O(n)

平均情况：O(nlogn)

2)稳定性：不稳定排序

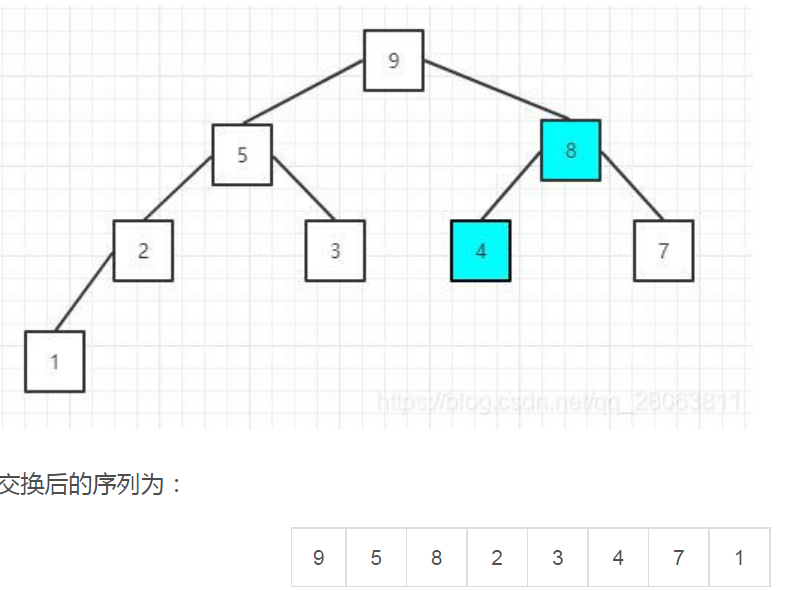
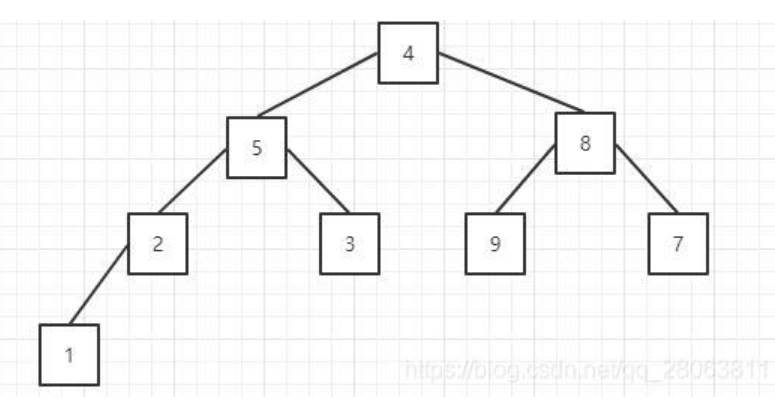
3)堆的特性：

大顶堆——父结点的值都大于或等于子节点的值；

小顶堆——父结点的值都小于或等于子节点的值；

4)基本步骤：

先让数组按照完全二叉树的方式排列成一个无序树，然后找到第一个非叶子结点（树的结点为n，第一个非叶子结点索引为n/2 -1（根结点索引为0）），从下向上比较父结点与最大子节点的值，如果父结点值小于最大子节点的值，则交换两者的值（大顶堆），交换之后要比较，要向下比较子结点值与孙子结点值，直到所有父结点的值都大于或等于子节点的值，这就是堆化的过程；然后将根结点的值与最后一个结点值互换，整个队的结点变为n-1，然后再堆化，一直循环到最后一个结点。



以上是从一个无序二叉树，到一个大顶堆的过程，之后将堆顶的元素值和尾部的元素交换，结点个数变为n-1后，继续堆化即可。

Eg:

void max\_heapify(int \* arr, int start, int end)

{

int dad = start;

int son\_l = dad \* 2 + 1;

while (son\_l <= end)

{

int son\_r = son\_l + 1;

int son\_max;

if (son\_r <= end && arr[son\_r] > arr[son\_l]) //如果有右子结点，且右子结点大于左子结点，则son\_max用右子结点，否则用左子结点

{

son\_max = son\_r;

}

else

{

son\_max = son\_l;

}

if (arr[dad] < arr[son\_max])

{

swap\_t(arr[dad], arr[son\_max]);

dad = son\_max;

son\_l = 2 \* dad + 1;

}

else

{

return;

}

}

}

void sort\_heap(int \* arr, int len)

{

for (int i = len / 2 - 1; i >= 0; i--)

{

max\_heapify(arr, i, len - 1);

}

swap\_t(arr[0], arr[len - 1]);

for (int j = len - 2; j > 0; j--)

{

max\_heapify(arr, 0,j);

swap\_t(arr[0], arr[j]);

}

}

int main()

{

int heap\_arr[7] = {12,15,9,20,6,31,24};

sort\_heap(heap\_arr, 7);

for (int k = 0; k < 7; k++)

{

cout << heap\_arr[k] << endl;

}

return 0;

}

### 3.5快速排序——分治法。

1)算法时间复杂度

最坏情况：O(n^2)

最好情况：O(nlogn)

平均情况：O(nlogn)

2)稳定性：不稳定排序

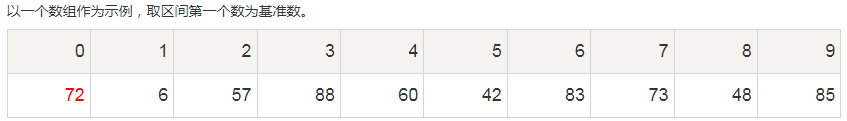
3）基本思想（从小到大排列）：

先从无序数组中随便找一个数作为基准数（这个索引是一个坑）；

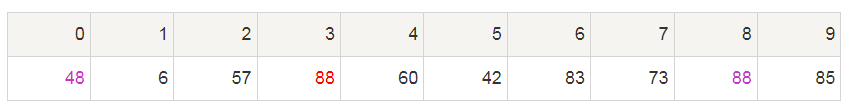
从右向左依次找比基准数小的数，找到后放在那个坑中，他就变成一个坑；然后左索引加1；

从左向右（这个右是第二步中的那个坑，因为再往右已经找过了）依次找比基准数大的数，找到后放到第二步中的那个坑里，它又变成一个坑，以此类推，直到左索引等于右索引，然后将基准数放到左右相等的索引中，此时基准数左边都比基准数小，右边都比基准数大；

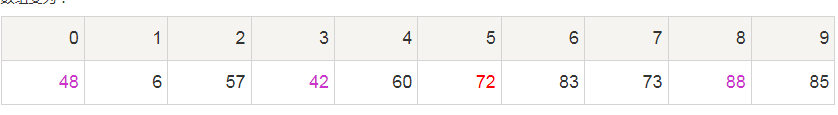
把基准数左右两边分为2个数组，循环执行1-3（用递归）。



基准数a[0]=72,第一步从右向左（9-0）依次找比基准数小的数，找到a[8]后放在索引为0坑中,索引8就变成了坑，从左向右（1-8）依次找比72大的数a[3]，找到后放到上一步中的索引8那个坑里，然后索引3就变成了坑：



从右向左（7-3）依次找比基准数72小的数a[5]，放到索引3的坑里，索引5变成坑，从左向右（4-5）依次找比72大的数，在找的过程中左索引等于右索引等于坑5，所以跳出，然后把基准数72放到a[5]中。



此时0-4为比基准数72小的数，6-9为比基准数72大的数。把基准数左右两边分为2个数组，循环执行以上操作（用递归）。

Eg:

void sort\_quick(int \* arr, int low, int high)

{

int l = low;

int h = high;

int tempt = arr[low];

if (h > l)

{

while (h > l)

{

for (h; h > l; --h) //从右往左找小的

{

if (tempt > arr[h])

{

arr[l++] = arr[h];

break;

}

}

for (l; h > l; l++) //从左往右找大的

{

if (tempt < arr[l])

{

arr[h++] = arr[l];

break;

}

}

}

arr[l] = tempt;

sort\_quick(arr, low, l - 1);

sort\_quick(arr, l + 1, high);

}

}

int main()

{

int quick\_arr[10] = { 72,6,57,88,60,42,83,72,48,85 };

sort\_quick(quick\_arr, 0,9);

for (int k = 0; k < 10; k++)

{

cout << quick\_arr[k] << endl;

}

return 0;

}

### 3.6 排序总结



各个排序面对数据的适用情况与小技巧，选取几个比较有特点有代表性质的排序算法：

（1）快速排序算法的效率体现在序列越乱的时候，效率越高，当数据趋于一个有序状态时（无论是顺序还是逆序），将会退化为冒泡排序，当数据完全处于逆序状态时，快速排序将会消耗极大的时间，因此有些OJ的快速排序模板测试题并不能直接写快排通过【毒瘤数据，超大完全逆序数据】，可以尝试适用rand()产生随机数的方法将整体数据打乱再使用快速排序，可以极大的减少运行时间。

（2）直接插入排序有着稳定和速度快的优点，缺点是比较次数越少，插入点后的数据移动就越多，特别是数据庞大的时候就需要大量的移动数据。

（3）堆排序作为一个相对比较复杂的排序，我们可以更加深入的去借鉴思想，比如有问题要求在n个数据中选出或者排序出前k个数据，利用堆排序的思维可以不将全部的n进行排序而只操作出前k个数据的情况，这个思维是很重要的。

（4）希尔排序最主要的操作是比较而不是交换，因此在小数组的情况下是比快速排序和堆排序要快的，但是涉及大量数据时依旧不如快排。

注：

1.i++和++i

i= 2；

i++:表示先赋值在自增——i++ = i = 2; i = i+1;

++i:表示先自增再赋值——i= i+1 = 3; ++i = i=3

或者：

a=i++——a=i=2;i=i+1=3;

b=++i——i=i+1=3;b=i=3

在for循环中：

for(语句一；语句二；语句三)

{

代码块；

}

语句一在循环（代码块）开始前执行；

语句二定义运行循环（代码块）的条件；

语句三在循环（代码块）已被执行后执行。

for(int i =0; i< 10;i++)和for(int i =0; i< 10;++i)在for循环中结果是一样的，都要等代码块执行完毕才能执行，但是性能不同。在大量数据的时候++i比i++的性能好的原因：

i++由于要先赋值然后自增，所以需要一个临时变量来存储，而++i先自增再赋值，省去了对内存的操作环节，相对而言能够提高性能。

# 二. 计算机内存

## 1. 内存存储



在函数调用时，由于局部变量（非static）存放在栈区，所以在函数运行结束后，其局部变量自动释放，如果返回一个局部变量的char\* p，返回的地址会被接收，但地址中的内容会被释放，所以不能返回局部变量的指针，除非它是一个静态的局部指针变量，这样它会被存储在全局区中。（指针变量就是虽然返回了地址，但是内容被释放了）。

## 2. 计算机存储的大端法和小端法

在计算机内存中，通常以字节byte( 8bit )为存储单位。对于多字节的数据类型（如int 4字节），在内存中存储多字节的常见方法有2种：大端法（big-endian）和小端法（little-endian）。

【注】不管是大端法还是小端法，计算机的存储顺序都是低地址到高地址，所不同的在于是首先读取低地址还是高地址。



大端法：首先读取高地址；

小端法：首先读取低地址；

Eg：

0XFF0A0B0C

大端法存储：以字节为单位，先读取高地址的值



小端法存储：以字节为单位，先读取低地址的值



# 三. 注