目录

[变量的申明 0](#_Toc1229278)

[库函数 0](#_Toc1229279)

[关于预处理的define指令 1](#_Toc1229280)

[Printf的格式化输出 2](#_Toc1229281)

[堆与栈 3](#_Toc1229282)

[printf中的修饰符： 3](#_Toc1229283)

[问题：为什么在VS编译器中使用scanf会报错？ 4](#_Toc1229284)

[Goto语句 5](#_Toc1229285)

[运算符优先级（从高到低） 5](#_Toc1229286)

[算法 9](#_Toc1229287)

# 变量的申明

Int 整形定义函数，int x叫做申明x这个变量。赋值只能为整数数字  
long int 长整形  
long long int  长长整形  
double float 双精度  
float 非整形赋值，浮点数单精度  
undesigned前缀省略，省略正负号（使赋值增加一倍大小，因为多了一位二进制数的空位）其实这边涉及到了原码反码补码的知识，因为计算机是以二进制存储数据的，正负号在计算机存储数据的方式补码中是作为第一位数字的（0/1），用了undesigned作为前缀，就是省下了补码的一个二进制单位，能够表达的数字大小自然而然也就大了一倍。

# 库函数

printf输出函数和scanf 输入函数都是在标准库函数<stdio.h>里面的，c语言其实并没有输入和输出函数，因为在标准库函数里面定义了这两个函数，才能在cmd控制台上输入参数出我们想要的字符。  
函数strlen()来获取字符长度，但是如果要使用strlen（）函数需要库函数<string.h>  
记住与字符串处理的函数大多和库函数<string.h>有关，比如字符串衔接函数strcat，字符串的赋值函数strcpy，字符串字典序排列函数strcmp。  
Strcat函数在使用的时候要保证母数组的单位要大于原来的母函数带有的字母数加子数列的字母数。  
Strcmp函数：strcmp（a,b）;如果a的字典序在b的前面，那么返回-1（false），如果a的字典序在b的后面，那么返回正数1（true），如果数组a,b相等，那么返回值为0；其实，字典序是根据ASCⅡ来排列的，比如‘A’<‘B’

字符串是字符的序列。双引号不是字符串的一部分，双引号仅告知编译器他括起来的是哪一个字符串，一般单引号用来标志一个字符。对于我现在接受的教育，切记scanf时的%s和%c是有很大的区别的，因为使用字符串“”默认在最后面加上了/0这个东西，会自动使用掉一个单位  
字符串都被储存在char类型的数组中，数组由连续的存储单元组成，每个单元存储一个字符，但是最后一个单元必定是/0，如果空了一大堆存储单元都会变成/0，也就是说数组的容量必须比带储存字符串多一位  
运算对象是类型时，圆括号必不可少，但是对于特定量时，可有可无。尽管如此，还是建议所有的情况下都使用圆括号。

# 关于预处理的define指令

#define指令

如#define RATE 0.001 那么在源码中所有的RATE都会被替换成0.001

他的通用格式是#define 符号常量名 符号常量值

那么，为什么用RATE呢，因为这样子看到大写的单词就知道是符号常量而非变量。大写只是为了提高程序的可读性（小写大写根据个人意愿）

如果要加入一个句子，应在句子的前后加上双引号。#define能够定义字符段，即“”格式，对于数字，是不需要特别说说明的

Const限定符：限定一个申明的量仅为为可读，比如const int MONTH=12；不再更改这个已经申明的量的数值。就像static限定符限定变量为静态变量

一些符号被称为转换说明：（一般情况下一个字符代表着八个字节）  
在已命名变量前面加（），（）里面加的是数据类型  
%d打印有符号的十进制整数  
%c打印单个字符  
%e，%E打印浮点数，e记数法  
%f 浮点数，十进制记数法  
在使用double类型的变量的时候，切记，一定要在scanf里面用%lf  
1、 在c语言中，使用printf函数输出float和double数据类型都可以使用%f。事实上，在用printf输出float数据类型的变量的时候也都可以使用%lf。所以在同时有float和double数据类型的变量的时候似乎在printf函数里面可以都时候%lf更加保险。  
2、 Scanf当中若是对double数据类型的变量赋值是必须用%lf的，对float数据类型的变量赋值是能且只能用%f的。在scanf里的规定就比较严格，不可以混淆使用 。

%g用的比较少，和科学记数法相关很大。我个人感觉好好使用在实际使用的时候用处很大。  
%g用来输出实数，它根据数值的大小，自动选f格式或e格式（选择输出时占宽度较小的一种），且不输出无意义的0（自动的进行科学记数法了）。即%g是根据结果自动选择科学记数法还是一般的小数记数法。但是%g仅对于指数小于-4（比如-5）或者大于给定精度的数值（正数）并且.2g之类的表示有效数字位数为2位,这些按照%e的控制输出,否则按照%f的控制输出。下面举了几个例子  
找到的4个例子（注释是我后来自己添的）  
printf("%g\n", 0.0000520); //小数点后有4个0。结果是5.20e-005  
printf("%g\n", 0.0001314);//小数点后有3个0。结果是0.0001314  
printf("%.2g\n", 123.45);//结果是1.2e+002  
printf("%.2g\n", 65.43);//结果是65

# Printf的格式化输出

对于printf的格式化输出，有一张表格，如下：  
％a     浮点数、十六进制数字和p-记数法（C99标准）  
%A    浮点数、十六进制数字和p-记法（C99标准）  
%c    一个字符   
%d    有符号十进制整数   
%e    浮点数、e-记数法  
%E    浮点数、E-记数法  
%f    浮点数、十进制记数法    
%g    根据数值不同自动选择％f或％e．  
%G    根据数值不同自动选择％f或％e.  
%i      有符号十进制数（与％d相同）  
%o    无符号八进制整数  
%p    指针      
%s    字符串  
%u    无符号十进制整数  
%x    使用十六进制数字0f的无符号十六进制整数   
%X    使用十六进制数字0f的无符号十六进制整数  
%%    打印一个百分号

在c语言中，strlen函数和sizeof关键字使用时结果很相似：  
strlen所作的是一个计数器的工作，它从内存的某个位置（可以是字符串开头，或者字符串中间某个位置，甚至是某个不确定的内存区域）开始扫描，直到碰到第一个字符串结束符为'\0'（也可以理解为碰到空格）停止，然后返回计数器值(长度不包含'\0')。  
  
区别sizeof  
对比strlen：strlen(char\*）函数求的是字符串的实际长度，它求得方法是从开始到遇到第一个'\0'，然后返回其中的字符数量（不包括‘\0’）.如果你只定义没有给它赋初值，这个结果是不定的，它会从首地址一直找下去，直到遇到'\0'停止。  
而sizeof：sizeof是取字节运算符（关键字），strlen是函数。  
sizeof可以用类型做参数，strlen只能用char\*做参数，且必须是以'\0'结尾的。  
sizeof还可以用函数做参数  
下面讲到的关于sizeof的变得高深一点了：  
数组做sizeof的参数不退化，但是数组传递给strlen就退化为指针了。  
大部分编译程序，在编译的时候就把sizeof计算过了，会把sizeof作为类型或是变量的长度。这就是sizeof(数组名)可以用来定义数组维数的原因  
strlen（）是用来计算字符串的长度，sizeof（）是类型占内存的大小。  
注：sizeof后如果是类型必须加括弧，如果是变量名可以不加括弧。这是因为sizeof是个操作符不是个函数。（我认为都加括号不容易失误，但是这个知识点难保什么时候会用到）但是sizeof 操作符不能返回动态地被分派了的数组或外部的数组的尺寸  
事实上在传递一个数组名到一个函数中时，它会完全退化为一个指针。数组作为参数传给函数时传的是指针而不是数组，传递的是数组的首地址。

其实理解 sizeof 只需要抓住一个要点：栈（一种数据结构）（关于栈和堆我会在下一篇发出）  
程序存储分布有三个区域：栈、静态和动态。能够从代码直接操作的对象，比如任何类型的变量、指针，都是在栈上的；  
动态和静态存储区是靠栈上的指针来间接操作的。sizeof 操作符，计算的是对象在栈上的投影体积；记住这个就很多东西都很清楚了。

堆与栈  
堆和栈都是一种数据排列的数据结构  
对于堆栈：  
只能在一端(称为栈顶(top))对数据项进行插入和删除。在单片机应用中，堆栈是个特殊的存储区，主要功能是暂时存放数据和地址，通常用来保护断点和现场。  
简而言之（结合内存想）：  
堆，队列优先,先进先出（FIFO—first in first out） 。栈，先进后出(FILO—First-In/Last-Out)。  
  
堆栈空间分配  
栈：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放。  
堆：一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。堆是存放在二级缓存中的，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度相对于栈要来得低一些。

c/c++：  
一个由C/C++编译的程序占用的内存分为以下几个部分  
1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数名，局部变量的名等。其操作方式类似于数据结构中的栈。  
2、堆区（heap）— 由程序员分配释放。若程序员不释放，程序结束时由OS（operating system）（操作系统）回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。  
3、静态区（static）—全局变量和局部静态变量的存储是放在一块的。程序结束后由系统释放。  
4、文字常量区—常量字符串就是放在这里的，程序结束后由系统释放 。  
5、程序代码区— 存放函数体的二进制代码。

printf中的修饰符：

-：从字符的左侧开始打印该项，如果不使用-，则默认为右对齐。如“%-6.2f”

+：有符号值若为正，则在值前面显示正号，若为负，则在值前面显示负号，如“%+6.2f”

空格：若符号值为正，则在值前面多显示一个空格，若值为负，则在值前面显示减号标记，减号标记覆盖一个空格

0：用处是使用数字0填充数字前面的空格。对于整数格式，如果出现-标记或者指定精度，则忽略该标记，如“%010d”“%0.83f”，即输出结果可能是00000007

#：把结果转换成另外一种形式，如果是%o形式，则以0开始，如果是%x或%X形式，则以0x或0X形式开始。对于后面的所有浮点格式，#保证了后面即使没有任何数字，也打印一个小数点字符。对于%g和%G格式，#防止结果后面的0被删除（我发现#加在scanf里面编译器也不会报错，为什么不报错暂时百度上也找不到原因）

我粗鄙的想了一下，无论是scanf，还是printf，他们的本质都是指针指向一个地址，把值放在那个地址上，然后再输入输出。从硬件方面讲，到现在为止我的认知是CPU通过地址引脚和数值引脚把这些数据传递内存里。关于硬的方面结合软的方面我最近的确因该整理一下了。 ​​​​

我发现了一个有点奇妙的东西，在用scanf给整个数组赋值的时候没有用&  
如果用scanf（）读取基本变量类型的值，在变量名前加上一个&  
如果用scanf（）把字符串读入字符数组中，不要使用&（就是char a【30】这种样子的字符数组不用&）  
那么，这是为什么呢？  
从scanf函数的函数原型来进行解释：  
scanf函数原型是：int scanf(char \*format, args, ...);//注：args表示的是字符串变量名。  
功能：从标准输入设备按format指定的输入格式将数据输入到args所指向的内存单元  
返回值：读入并赋给args的数据个数。遇文件结束返回EOF，出错返回0  
从scanf函数的函数原型可以看出，scanf接收到的数据是赋给args指向的内存单元的，也即args表示的是一个内存地址，而变量的内存地址用取地址符&来表示，数组的地址就是数组名。所以数组的前面不用加&  
举例如下：  
  
int a;     // 定义一个变量  
char b[3]; // 定义一个字符数组  
char \*str=(char \*)malloc(sizeof(char)\*4);  // 定义字符串并分配内存单元，malloc的用处是动态分配内存地址，sizeof（char）\*4表示的是一个char类型的内存大小乘以4倍。然后把这么一块的内存动态分配给malloc前面的那一个指针。

问题：为什么在VS编译器中使用scanf会报错？  
生生回答：这个其实涉及到了程序的一个安全性问题。首先声明一下，scanf\_s是微软公司在VS里面创造的，scanf是c语言标准库里面的。也就是说，scanf\_s在其他的编译器里面是不通用的，如果在程序源码里面使用了scanf\_s会大大降低程序的通用性。存在的东西一定有它存在的理由，scanf\_s的优点是增强了系统的安全性：它避免了堆栈的溢出，防止读入过多写入其他的内存区块导致程序运行出错。然而scanf()在读取数据时不检查边界，所以很可能造成内存访问越界  
举一个例子：  
//例如：编写了一个程序，程序给一个变量分配了3个字节的空间但是用户输入了10字节，就会导致scanf()读到10个字节  
char buf[3]={'\0'};  
scanf("%s", buf);  
//如果输入1234567890，那么从4开始以后的部分会被写到别的变量所在的空间上去，从而可能会导致程序运行异常。  
如果源代码使用scanf\_s就没有了溢出这个漏洞：  
char buf[3]={'\0'};  
scanf\_s("%s",buf,3); //最多读取2个字符，因为buf[2]要放'\0'   
//如果输入1234567890，则buf只会接受前2个字符  
  
所以在使用VS的时候，可以手动设置或者#define掉这个警告

补充在前面没有写到的：什么是修饰符？答：可以在百分号和转换字符之间使用修饰符。但如果使用多个修饰符，必须按照下面的书写顺序。  
Scanf的修饰符：  
1、\*：抑制赋值，如“%\*d”（这个表示在输入的部分中，到达了符合%d整形这一部分的时候抑制赋值）  
2、数字：表示最大字符宽度。输入达到最大字符宽度处，或第一次遇到空白字符时停止。如“%10s”  
3、hh：把整数作为signed char或者unsigned char类型读取。如“%hhd“  
4、ll：把整数作为long 或者unsigned long类型读取（C99）。如“%lld“  
%p记数法：这个是新增的，是十六进制指数记数法  
  
事实上关于scanf的修饰符还有几个，考虑到我还记不住，以后碰到的时候一定好好学一下。

# Goto语句

goto语句有两个部分：goto和标签名  
一般情况下goto语句的样式是这个样子的：  
goto  abc;      
abc：XXXXXX        
goto常常用于在程序发生错误的时候用于跳出语句和检查错误。//很多人建议少用goto语句标签名，他们给出的原因是这么用会导致源代码混乱。 ​​​​

运算符优先级（从高到低）  
（）：从左往右  
+-（一元）：从右往左//注：（x+y）++无效，因为递增和递减运算符只能影响一个变量  
\*/：从左往右  
+-（二元）：从左往右  
=：从右往左 ​​​​

**关于stdlib.h（下面的15个函数名称的具体解释来源于百度百科）**stdlib 头文件即standard library标准库头文件。stdlib.h里面定义了五种类型、一些宏和通用工具函数。 类型例如size\_t、wchar\_t、div\_t、ldiv\_t和lldiv\_t； 宏例如EXIT\_FAILURE、EXIT\_SUCCESS、RAND\_MAX和MB\_CUR\_MAX等等； 常用的函数如malloc()、calloc()、realloc()、free()、system()、atoi()、atol()、rand()、srand()、exit()等等。 具体的内容可以打开编译器的include目录里面的stdlib.h头文件查看。  
  
包含的函数  
输入样式：C语言模式：[#include <stdlib.h>  
C++样式：#](https://s.weibo.com/weibo/%23include+%3Cstdlib.h%3E%0AC%2B%2B%E6%A0%B7%E5%BC%8F%EF%BC%9A%23?from=default)include <cstdlib>  
1函数名称:calloc  
函数原型: void calloc(unsigned n,unsigned size);  
函数功能: 分配n个数据项的内存连续空间,每个数据项的大小为 size  
函数返回: 分配内存单元的起始地址,如果不成功,返回0  
2函数名称:free  
函数原型: void free(void\* p);  
函数功能: 释放 p 所指的内存区  
参数说明: p是被释放的指针  
3函数名称:malloc  
函数原型: void \* malloc(unsigned size);  
函数功能: 分配 size 字节的存储区  
函数返回: 所分配的内存区地址,如果内存不够,返回0  
4函数名称: realloc  
函数原型: void \* realloc(void \* p,unsigned size);  
函数功能: 将 p 所指出的已分配内存区的大小改为 size,size 可以比原来分配的空间大或小  
函数返回: 返回指向该内存区的指针.NULL-分配失败  
5函数名称: rand  
函数原型: int rand(void);  
函数功能: 产生0到32767间的随机整数(0到0x7fff之间)  
函数返回: 随机整数  
6函数名称: abort  
函数原型: void abort(void)  
函数功能: 异常终止一个进程.  
7函数名称: exit  
函数原型: void exit(int state)  
函数功能: 程序中止执行，返回调用过程  
参数说明: state:0- 正常中止，非 0- 非正常中止  
8函数名称: getenv  
函数原型: char\* getenv(const char \*name)  
函数功能: 返回一个指向环境变量的指针  
函数返回:环境变量的定义  
参数说明: name- 环境字符串  
9函数名称: putenv  
函数原型: int putenv(const char \*name)  
函数功能: 将字符串name增加到DOS环境变量中  
函数返回: 0:操作成功，-1:操作失败  
参数说明: name-环境字符串  
10函数名称: labs  
函数原型: long labs(long num)  
函数功能: 求长整型参数的绝对值  
函数返回:绝对值  
11函数名称: atof  
函数原型: double atof(char \*str)  
函数功能: 将字符串转换成一个双精度数值  
函数返回: 转换后的数值  
参数说明: str- 待转换浮点型数的字符串  
12函数名称: atoi  
函数原型: int atoi(char \*str)  
函数功能: 将字符串转换成一个整数值  
函数返回: 转换后的数值  
参数说明: str- 待转换为整型数的字符串  
13函数名称: atol  
函数原型: long atol(char \*str)  
函数功能: 将字符串转换成一个长整数  
函数返回: 转换后的数值  
参数说明: str- 待转换为长整型的字符串  
14函数名称:ecvt  
函数原型: char \*ecvt(double value,int ndigit,int \*dec,int \*sign)  
函数功能: 将浮点数转换为字符串  
函数返回: 转换后的字符串指针  
参数说明: value- 待转换底浮点数，ndigit- 转换后的字符串长度  
15函数名称:fcvt  
函数原型: char \*fcvt(double value,int ndigit,int \*dec,int \*sign)  
函数功能: 将浮点数变成一个字符串  
函数返回: 转换后字符串指针  
参数说明: value- 待转换底浮点数，ndigit- 转换后底字符串长度

**逻辑运算符：（c有三种逻辑运算符）**&&与  
||或  
！非  
特别要注意：&&的优先级比||高  
  
条件运算符：？是c语言里面唯一的三元运算符  
条件运算符的通用形式如下：a?b:c  
如果条件a为真，那么跳转到b；如果条件b为假，那么跳转到c

**计算机是怎么跑起来的  【关于计算机硬件方面的笔记】**计算机执行的是输入，计算，输出  
软件是指令和数据的集合  
IC的意思是集成电路（intergrated circuit）  
I/O是基本输入输出端口（input/output）  
CPU（Central Processing Unit）中央处理器是根据时钟频率工作的，有一对引脚是电源引脚。比如，一端5V，一端0V。不同的引脚根据时钟频率的不同变换不同引脚的电压输出高压（1），低压（0）。  
引脚分为地址引脚（Address）和数据引脚（Date）。  
DMA（direct memory access）直接存储器访问，不经过CPU直接从外部设备读取内存的行为。  
CPU以K结尾，表示不锁倍频，就是指可以超频。  
LGA：(land grid array)栅格阵列封装，这个是CPU的封装方式，就是一块板子上一个个小格子不是针脚是触脚，针脚在主板上。  
PCIE（peripheral component interconnect express）：既是总线也是接口协议，全名为高速窜行计算机拓展总线标准  
m.2接口：一般是给固态硬盘用的。使用PCIE（peripheral component interconnect express)是一种高速串行计算机扩展总线标准，传输速度要比SATA总线的盘快。在M.2里面2280表示宽22mm，高80mm这种；规格叫做全高盘  
NVME（Non-Volatile Memory express）：非易失性内存主机控制器接口规范，是M.2里面的协议  
OTG（On-The-Go）：协议，目的是让你的设备链接各种不同的其他USB设备

QLC（Quad-Level Cell）：四层单元存储，并且每个cell单元都可存储4bit的信息,电压有16种变化从0000到1111。这是固态里面的颗粒属性。分为单层，双层，三层，四层。优点是存放同样多少的数据陈本更加低，缺点是查找数据的时候会变慢，对一个单元的多次访问会降低使用耐久，所以QLC的耐久度是最低的。  
RGB（red-green-blue）：光学三原色：红色绿色蓝色  
强度为0~255  
SSD（solid state disk）：固态硬盘的内部采用闪存芯片作为存储介质  
HDD（Hard Disk Drive）：机械硬盘/机械硬盘驱动器。机械硬盘数据安全性相对于固态硬盘更好，但是速度要慢。  
USB（universal Serial Bus）：通用串行总线usb3.1Gen2就是现在的type-c  
WLAN（wireless local area networks）： 无线局域网，也就是我们常说的WiFi使用的是IEEE （电气和电子工程师协会。IEEE全称是Institute of Electrical and Electronics Engineers）802.11的通信标准 当一个设备进入网络时进行动态IP（Internet Protocol Address，又译为网际协议地址，是分配给用户上网使用的网际协议）分配，离路由器越远信号越差。  
PCB（printed circuit board）印刷电路板  
R:Resistor,电阻。C:Capacitor,电容。L:Inductor,电感。  
D：dideo二极管

# 算法

机器语言有时候也叫原生代码（native code）  
哪个寄存器对应哪个地址取决于CPU和I/O之间的布线方式

排序算法  
桶排序（简化版）：  
可以解决的问题：有一组数据，对他们进行排序。  
解决方法：设置一个一维数组，把这个一维数组想象成一个个桶。把输入的数组一个个丢进去，这个一维数组的单个数组值用来表示这个数字出现的次数。最后输出结果的时候再对这个一维数组进行遍历。  
弊端：如果输入数字很大，那么这个一维数组需要建造的很大，很浪费空间。无法排列小数。  
优点：在小数排列的时候，能够大大节省时间  
  
冒泡排序：  
可以解决的问题：有一组数据，对他们进行排序。  
解决方法：每次比较两个相邻的元素，如果它们的排序错误就把它们交换过来。  
弊端：时间复杂度偏高，时间复杂度为N的二次方。  
优点：搭配结构体可以进行带人名的分数排序；可以进行小数排序；无惧大数  
  
快速排序（冒泡排序的升级版）：  
可以解决的问题：有一组数据，对他们进行排序（冒泡排序plus版本）。  
解决方法：设置一个基准数（一般默认数据的第一个数），然后设置两个哨兵（称为哨兵A和哨兵B），一个哨兵（哨兵A）在数据首端，另一个哨兵（哨兵B）在数据末端。接下来开始比较基准数和哨兵B。如果哨兵B大于基准数，那么哨兵逐个向数据首端推进，直到发现比基准数小的，然后哨兵B停在这个比基准数小的数上面。接着哨兵A开始向数据末端推进，跳过比基准数小的数，停在第一个比基准数大的数上。然后哨兵A和哨兵B下面的数进行对换。如此往复下去。直到哨兵A和哨兵B相遇，再把基准数和相遇位置的数字对换，完成第一次快速排序。 接着，开始第二次快速排序：整段数据已经被基准数分为了两段，对前段和后段落同第一次快速排序一样进行操作。再进行第三次，第四次等等快速排序……等到整组数据只存在一个数的时候，整列数据就被排列好了。  
思想：用到了二分法。  
弊端：最坏的情况下和冒泡排序的时间复杂度一样。  
优点：是冒泡排序的PLUS版本，相对于冒泡排序大大降低了时间复杂度。虽然最坏的情况和冒泡排序的时间复杂度一样，但是这种情况极少。它的平均时间复杂度为NlogN  
  
写道这里的一种感觉：算法存在的理由是减少程序运行消耗的时间，减少代码的时间复杂度。

栈、队列、malloc函数  
队列：队列是一种特殊的线性结构，它只允许在队列的首端进行删除，在队列的末尾进行增添。称之为 先进先出，first in first out（FIFO）原则，队列是一种数据结构。  
对于队列，可以想象成一个买票的队伍。  
下面是将队列的三个基本元素（一个数组，两个变量）封装为一个结构体类型：  
struct queue  
{  
Int date [99];  
Int head;  
Int tail;  
};  
(温故一下申明一个新的结构体：struct queue num1)  
(温故一下引用结构体中的成员：num1.head)  
  
栈(stack)：栈是一种后进先出的数据结构，它只能在一端进行增添、删除。  
可以把栈想象成弹夹，数据想象成子弹。  
栈最初存在的目的是为了解决子程序的调用和返回。栈可以用于处理回文数类问题。  
  
链表：有时候使用数组不灵活，就可以考虑使用链表。链表可以在很多地方自由增添/删除。  
  
malloc是动态内存分配函数，如果要申请4个字节的空间：malloc(4);或者malloc(sizeof(int))；  
对这个空间进行操作，需要用指针指向这个空间（即存储这个空间的首地址），如：  
Int \*p; p=(int \*)malloc(sizeof(int));  
使用malloc函数有利于后期程序的维护

由于我暂时使用的是安卓系统的华为mate10Pro，这几天想备份数据，又因为文件夹洁癖强迫症犯了，在网上搜了一点安卓系统文件夹的情况，如下。  
安卓手机的文件比较不容易区分，一般都是英语命名，除非是自己设置的文件名，所以能不删的，尽量都别删，如果是空间不足的话，又想删除一些东西，那文件名前面有带一个点的 是一定不能删的 这类文件类似系统文件 没有带点的文件名删了不会影响手机 但是会影响你下载的某些软件或者游戏下面这些是网上的一个帖子：  
1、.android\_secure 是官方app2sd的产物，删了之后装到sd卡中的软件就无法使用了。  
2、.Bluetooth 顾名思义，用蓝牙之后就会有这个。  
3、.mobo 顾名思义，Moboplayer的缓存文件。  
4、.QQ 顾名思义，QQ的缓存文件。  
5、.quickoffice 顾名思义，quickoffice的缓存文件。  
6、.switchpro 顾名思义，switchprowidget（多键开关)的缓存文件。  
7、.ucdlres 顾名思义，UC迅雷的缓存文件。  
8、albumart 音乐专辑封面的缓存文件夹。  
9、albums 相册缩略图的缓存文件夹。  
10、Android 比较重要的文件夹，里面是一些程序数据，比如google map的地图缓存。  
11、backups 一些备份文件，比如联系人导出到SD卡时会导入到此文件夹。  
12、baidu 顾名思义，掌上百度、百度输入法之类程序的缓存文件夹。  
13、bugtogo 系统出现问题的时候会形成一些报告文件，存放于此文件夹。  
14、cmp 个人初步判断是音乐的缓存文件夹。  
15、data 同样是缓存数据的文件夹，与Android性质类似。  
16、DCIM 相机的缓存文件夹。  
17、documents Documents To Go 的相关文件夹。  
18、etouch 易行的缓存文件夹。  
19、extracted androzip等解压缩软件默认的解压目录。  
20、gameloft 顾名思义，gameloft游戏数据包存放的文件夹。  
21、handcent 顾名思义handcent（超级短信）数据文件夹。  
22、handyCurrency货币汇率相关的文件夹，装了handycalc（科学计算器）之后才会有。  
23、ireader 顾名思义，ireader的缓存文件夹。  
24、KingReader 顾名思义，开卷有益的缓存文件夹。  
25、LazyList Applanet（黑市场）的缓存目录，也许和其他程序也有关，暂时不太清楚。  
26、LOST.DIR 卡上丢失或出错的文件会跑这里，此目录无用，删了会自动生成。  
27、moji 顾名思义，墨迹天气的缓存目录。  
28、MusicFolders poweramp产生的缓存文件夹。  
29、openfeint 顾名思义，openfeint的缓存文件夹。  
30、Picstore 图片浏览软件建立的一个目录。  
31、Playlists 播放列表的缓存文件夹。  
32、renren 顾名思义，人人网客户端的缓存文件夹。  
33、screenshot 貌似是截屏图片保存的目录，不过我不记得自己装过screenshot这个软件。  
34、ShootMe 顾名思义，shootme截屏后图片文件保存的目录。  
35、SmartpixGames Smartpix Games出品游戏的缓存文件夹，比如Jewellust。  
36、sogou 顾名思义，搜狗拼音的缓存文件夹。  
37、SpeedSoftware RE文件管理器的缓存文件夹。  
38、SystemAppBackup SystemApp remove（深度卸载）备份系统文件后，备份文件保存的目录。  
39、TalkingFriends talking tom（会说话的tom猫）录制的视频文件所保存的目录。  
40、Tencent 顾名思义，腾讯软件的缓存目录，比如QQ。（与上面的.QQ文件夹并不相同）  
41、TitaniumBackup 顾名思义，钛备份备份的程序所保存的目录。  
42、TunnyBrowser 感觉是海豚浏览器的缓存目录，但不知道为什么叫这个名字，金枪鱼浏览器.....  
43、UCDLFiles UC迅雷下载文件的保存目录。  
44、UCDownloads UC浏览器下载文件的保存目录。  
45、VIE Vignette（晕影相机）的缓存目录。  
46、VPN 顾名思义，VPN数据的缓存目录。  
47、yd\_historys 有道词典搜索历史的缓存目录。  
48、yd\_speech 有道词典单词发音的缓存目录。  
49、youmicache 删掉后还会自动生成，悠米广告的缓存目录，广告程序内嵌在其他程序中。  
50、Glu Glu系列游戏的资料包存放地。如3D猎鹿人等。  
51、apadqq-images QQ for pad 的缓存目录。  
52、DunDef 地牢守护者的数据包。  
53、KuwoMusic 顾名思义，酷我音乐的相关文件夹。  
54、MxBrowser 遨游的缓存目录。  
55、Camera360 顾名思义，camera360的缓存目录。  
56、TTPod 顾名思义，天天动听的缓存目录。  
总的来说，那个文件夹是干什么用的凭借它本身的文件名是能够大致的推测出来的。

链表  
链表是由一个个结点构成的。  
讲一下结点和节点的区别：  
结：连结，终结  
节：关节  
节点是一个实体，有处理能力。例如网络上的一台计算机；  
结点是一个交叉点，像“结绳记事”，打个结，做个标记，一般算法中点都是结点；  
复杂网络理论中所谈到的点是“节点”。  
  
链表中的结点是这么存储的：  
链表的结点分为两个部分。一个部分用来存放具体的数据（比如int类型数据等），另一部分用于存储下一个结点的地址，存储下一个结点的地址往往可以使用指针来实现（称为后继指针）。那么一个简单的结点可以表示成：  
struct node  
{  
int data;  
struct node \*next;  
}  
因为下一个结点的数据结构是struct node，所以这个指针（next）的类型是struct node \*  
首先，需要一个头指针head指向链表的最开始。当链表还没有建立的时候头指针head为空，或者说头指针指向空结点。  
如：struct node \*head；  
head=NULL；//头指针初始为空  
接下来创造第一个节点，并用临时指针p指向这个结点：  
struct node \*p；  
p=（struct node \*p）malloc（sizeof（struct node））；  
//动态申请一个空间，用来存放一个节点，并用临时指针p指向这个结点  
  
接下来分别设置新创建的这个结点的两个部分：  
Scanf（“%d”，&a）；  
p->data=a；//将数据存储到当前结点的data变量中  
p->next=NULL；//设置当前结点的后继指针指向空，也就是当前结点的下一个结点为空。  
  
对于->符号的讲解：->叫做结构体指针运算符，它是用来访问结构体内部成员的。因为上文中的p是一个指针，所以不能使用 . 这个符号来访问内部成员，而而要使用->  
  
接下来设置头指针并设置新创建结点的\*next指向空。头指针的作用是方便以后从头遍历整个链表。  
If（head=NULL）  
Head=p;//如果这是第一个创建的结点，则将头指针指向这个结点  
Else   
q->next=p;//如果不是第一个创建的结点，则将上一个节点的后继指针指向当前的结点  
  
最后要将指针q也指向当前结点，因为过一会临时指针p将会指向新创建的结点。  
q=p;//指针q也指向当前结点  
  
注：上面的代码没有释放动态申请的空间，虽然没有错误，但是这样会很不安全，这时候要使用到free命令（下一篇将讲free函数）  
  
假设接下来要插入一个数（在按照顺序排序的链表中插入）：  
首先使用一个临时指针t从链表的头部开始遍历：  
t=head;//从链表的头部开始遍历  
等到指针的下一个结点的值比6大的时候，将6插入到中间。  
即  
t->next->data大于6的时候插入  
如下：  
scanf("%d",&a);//读入待插入的数  
while(t!=NULL)//当没有到达链表尾部的时候循环  
{  
if(t->next==NULL||t->next->data>a)//如果当前结点是最后一个结点或者下一个结点的值大于待插入数的时候插入  
{  
  p=(struct node \*)malloc(sizeof(struct node));//动态申请一个空间，用来存放新增结点  
  p->data=a;  
  p->next=t->next;//新增结点的后继指针指向当前结点的后继指针所指向的结点   
  t->next=p;//当前结点的后继指针指向新增结点  
  break；//插入完成后推出循环   
}  
t=t->next;//继续下一个结点    
}

free（）是一个C语言函数，它所需要的是头文件：malloc.h或stdlib.h  
作用：释放malloc (或calloc、realloc)函数给指针变量分配的内存空间的函数  
使用后该指针变量一定要重新指向NULL，防止野指针出现，有效规避误操作。  
free（）/\*一般这个括号里面是malloc申请空间对应的数据结构）free（）一般与malloc()函数配对使用，释放malloc函数申请的动态内存。（另：对于free(p)这句语句，如果p 是NULL指针，那么free 对p 无论操作多少次都不会出问题。如果p 不是NULL 指针，那么free 对p连续操作两次就会导致程序运行错误。）

链表的话，我感觉就像是一串钥匙扣一样，一个扣着另一个。如果要在一串钥匙扣里面增添或者删除一个结点（钥匙扣），就得把这个结点（钥匙扣）的前面一个和后面一个松开，再增添或者删除一个结点（钥匙扣）。之后，还得把前面和后面的钥匙扣再连接上去，再次成为完整的链表（一串钥匙串）。  
  
链表还有一种用数组实现的方法（有别于刚刚用结构体和指针实现的），叫做模拟链表。  
模拟链表的主要思路是建立两个数组。第一个数组data用于存储数据，另外一个数组用于存储相应data数组的相应位置的元素的下一个元素的下一个位置是哪里（所以第二个数组每一个元素的初始化值是数组序号加1）。事实上，模拟链表的第二个数组也是起到了指针的作用，用于指向第二个地址。  
  
总的来说，我认为模拟链表相对于链表就像是不增加其他变量对a，b变量进行值交换对于使用temp变量对a，b变量进行值交换。

广度优先搜索/算法（BFS）和深度优先搜索/算法（DFS）  
  
深度优先搜索算法（Depth-First-Search）是一种用于遍历或搜索树或图的算法。沿着树的深度来遍历树的节点，以尽可能深的搜索树的分支。当节点所在边都己被探寻过，搜索将回溯到发现节点的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。  
  
DFS的思想是从一个顶点V0开始，沿着一条路一直走到底，如果发现不能到达目标解，那就返回到上一个节点，然后从另一条路开始走到底。   
DFS适合此类题目：给定初始状态跟目标状态，要求判断从初始状态到目标状态是否有解。  
  
广度优先算法（Breadth-First- Search），同广度优先搜索，又称作宽度优先搜索，或横向优先搜索，是一种图形搜索演算法，广度，就是一层一层的，向下遍历，层层堵截。  
简单的说，BFS是从根节点开始，沿着树的宽度遍历树的节点，如果发现目标，则演算终止。  
  
BFS是一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。BFS并不使用经验法则算法。  
  
深度与广度的比较  
我们搜索一个图是按照树的层次来搜索的。  
我们假设一个节点衍生出来的相邻节点平均的个数是N个，那么当起点开始搜索的时候，队列有一个节点，当起点拿出来后，把它相邻的节点放进去，那么队列就有N个节点，当下一层的搜索中再加入元素到队列的时候，节点数达到了N的平方，你可以想想，一旦N是一个比较大的数的时候，这个树的层次又比较深，那这个队列就得需要很大的内存空间了。  
于是广度优先搜索的缺点出来了：在树的层次较深且子节点数较多的情况下，消耗内存十分严重。广度优先搜索适用于节点的子节点数量不多，并且树的层次不会太深的情况。  
那么深度优先就可以克服这个缺点，因为每次搜的过程，每一层只需维护一个节点。但回过头想想，广度优先能够找到最短路径，那深度优先能否找到呢？深度优先的方法是一条路走到黑，那显然无法知道这条路是不是最短的，所以你还得继续走别的路去判断是否是最短路？  
于是深度优先搜索的缺点也出来了：难以寻找最优解，仅仅只能寻找有解。其优点就是内存消耗小，克服了刚刚说的广度优先搜索的缺点。  
  
补充一个小知识：在这次编辑中，我发现在word文档里面选定想变为上标的东西，按住ctrl+shift+=就会把选中的东西变成上标。（可惜微博不支持上标，于是我又改了回来）

三个路径算法  
Floyd-warshall:  
Floyd-Warshall 算法用来找出每对点之间的最短距离。它需要用邻接矩阵来储存边，这个算  
法通过考虑最佳子路径来得到最佳路径。单独一条边的路径不一定是最佳路径。   
从任意一条单边路径开始，所有两点之间的距离是边的权的和(如果两点之间没有边相连, 则为无穷大）。 从第一个顶点开始，依次将每个顶点作为媒介 k，然后对于每一对顶点 u 和   
v ，查看其是否存在一条经过 k 的，距离比已知路径更短的路径，如果存在则更新它。  
  
Dijkstra:  
Dijkstra算法一般的表述通常有两种方式，一种用永久和临时标号方式，一种是用OPEN, CLOSE表的方式，这里均采用永久和临时标号的方式。注意该算法要求图中不存在负权边。     
  
Bellmanford:  
Dijkstra算法无法判断含负权边的图的最短路。如果遇到负权，在没有负权回路（回路的权值和为负，即便有负权的边）存在时，也可以采用Bellman - Ford算法正确求出最短路径。  
Bellman-Ford算法能在更普遍的情况下（存在负权边）解决单源点最短路径问题。对于给定的带权（有向或无向）图 G=(V,E), 其源点为s，加权函数 w是 边集 E 的映射。对图G运行Bellman - Ford算法的结果是一个布尔值，表明图中是否存在着一个从源点s可达的负权回路。若不存在这样的回路，算法将给出从源点s到 图G的任意顶点v的最短路径d[v]。

关于树：  
结点：结点分为子节点，父节点。第一个结点（没有父节点）叫做根节点，没有子节点的叫做叶节点。不是叶节点也不是根节点的叫做中间结点。  
二叉树：一个结点要么没有子节点，要么有两个子节点  
完全二叉树：在树的最右只有一个或多个叶节点缺失。完全二叉树的最典型应用就是——堆  
  
堆(heap)：是一类数据结构的统称。堆通常是一个可以被看做一棵树的数组对象。  
满足下列性质：  
堆中某个节点的值总是不大于或不小于其父节点的值，堆总是一棵完全二叉树。  
将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。  
  
并查集：  
在一些有N个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。并查集是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并及查询问题。常常在使用中以森林来表示。

红黑树(平衡的二叉搜索树)：  
红黑树（Red Black Tree）是一种自平衡二叉查找树，是一种数据结构，典型的用途是实现关联数组。  
红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。  
它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的：它可以在O(log N)时间内做查找，插入和删除，这里的N是树中元素的数目。  
  
区间树：  
区间树是在红黑树基础上进行扩展得到的支持以区间为元素的动态集合的操作，其中每个节点的关键值是区间的左端点。通过建立这种特定的结构，可是使区间的元素的查找和插入都可以在O(lgN)的时间内完成。  
  
线段树：  
线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。  
使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为O(logN)。而未优化的空间复杂度为2N，实际应用时一般还要开4N的数组以免越界，因此有时需要离散化让空间压缩。  
线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。  
对于线段树中的每一个非叶子节点[a,b]，它的左儿子表示的区间为[a,(a+b)/2]，右儿子表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的子节点数目为N，即整个线段区间的长度。  
使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为O(logN）。而未优化的空间复杂度为2N，因此有时需要离散化让空间压缩。

树状数组(Binary Indexed Tree)：  
是一个查询和修改复杂度都为log(N)的数据结构。主要用于查询任意两位之间的所有元素之和，但是每次只能修改一个元素的值；经过简单修改可以在log(N)的复杂度下进行范围修改，但是这时只能查询其中一个元素的值(如果加入多个辅助数组则可以实现区间修改与区间查询)。  
注：这种数据结构（算法）并没有C++和Java的库支持，需要自己手动实现。树状数组和线段树很像，但能用树状数组解决的问题，基本上都能用线段树解决，而线段树能解决的树状数组不一定能解决。相比较而言，树状数组效率要高很多。  
  
哈希树：  
在计算机科学中，哈希树（或哈希特里）是一种持久性数据结构，可用于实现集合和映射，旨在替换纯函数式编程中的哈希表。 在其基本形式中，哈希树在trie中存储其键的哈希值（被视为位串），其中实际键和（可选）值存储在trie的“最终”节点中  
  
字典树（Trie树）：  
又称单词查找树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

二叉查找树（Binary Search Tree）：  
又叫二叉搜索树，二叉排序树。它是一棵空树，或是具有下列性质的二叉树：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；它的左、右子树也分别为二叉排序树。  
二叉排序树的查找过程和次优二叉树类似，通常采取二叉链表作为二叉排序树的存储结构。中序遍历二叉排序树可得到一个关键字的有序序列，一个无序序列可以通过构造一棵二叉排序树变成一个有序序列，构造树的过程即为对无序序列进行排序的过程。每次插入的新的结点都是二叉排序树上新的叶子结点，在进行插入操作时，不必移动其它结点，只需改动某个结点的指针，由空变为非空即可。

无向图：  
边没有方向的图称为无向图。  
  
连通图：  
在图论中，连通图基于连通的概念。在一个无向图 G 中，若从顶点i到顶点j有路径相连，则称i和j是连通的。如果 G 是有向图，那么连接i和j的路径中所有的边都必须同向。如果图中任意两点都是连通的，那么图被称作连通图。如果此图是有向图，则称为强连通图（注意：需要双向都有路径）。图的连通性是图的基本性质。  
  
最小生成树：  
一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。最小生成树可以用kruskal（克鲁斯卡尔）算法或prim（普里姆）算法求出。  
生成树和最小生成树有许多重要的应用。  
  
二分图匹配：  
给定一个二分图G，在G的一个子图M中，M的边集{E}中的任意两条边都不依附于同一个顶点，则称M是一个匹配。  
  
极大匹配(Maximal Matching)：  
是指在当前已完成的匹配下,无法再通过增加未完成匹配的边的方式来增加匹配的边数。最大匹配(maximum matching)是所有极大匹配当中边数最大的一个匹配。选择这样的边数最大的子集称为图的最大匹配问题。如果一个匹配中，图中的每个顶点都和图中某条边相关联，则称此匹配为完全匹配，也称作完备匹配。

枚举法：

在暴力的使用枚举法的时候，千万要注意算法的时间复杂度。有时候稍微思考一下可以省下一个数量级。

贪心算法（贪婪算法）：

在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。