1.操作系统底层就是用C语言写的，MySQL数据库也是用c/c++写的。

为什么要学习Linux系统编程？

操作系统资源指: CPU 内存 磁盘 网络

因为c/c++不提供操作这些的接口，故在什么操作系统下，就要学习对应操作系统提供的接口：windows下就是windows系统编程，提供的是c/c++接口； Linux系统提供的是纯C的接口，因为Linux系统是用纯C写的。

大型程序：必须用面向对象的思想来写大型软件，因为方便维护升级。

Linux内核就是用纯C写的面向对象的程序。 C也能写面向对象(比如C中的结构体)，只不过水平要求很高。

c/c++难度在于：精准的内存管理

英特尔的CPU有一级缓存和二级缓存，AMD的CPU没有，所以英特尔的CPU更好。

2.

(1)微软的IDE即VS用的编译器是masm编译器，不用GCC

谷歌、Linux的编译器用的是GCC

因为**微软的masm编译器是C与C++混合的编译器**，所以我们选择C++文件，但是我们新建关于C的代码文件时，后缀命名为.c，比如这次我们以main.c命名，那么masm编译时，就会按照C语言的规则进行编译。

(2)VS中编辑、编译、调试均在一起的(IDE)；Linux都是分开的，编辑(vim)、编译(gcc)、调试(gdb)。

VS中一个解决方案文件夹中可以有多个项目文件夹，而一个项目文件夹中可放多个.c文件，1个项目最终会编译成1个可执行程序(windows下即exe文件)。

编好exe可执行程序后，会放在解决方案文件夹下的Debug文件夹中。

(3)VS的win32控制台程序的地址空间是2的32次方，即4G(VS看内存就是在这个范围内)

一般设成内存只用看前8列(只是显示问题，方便看)，1列代表4位，1个字节有2列即8位，所以共有32位即4个字节。

VS内存以16进制显示：因为这样每个字节都是对齐的，看内存会比较方便。

VS内存的4个字节之后： 显示每个字节的二进值转换为10进制的ASCII值后所对应的字符，**若VS中内存某个字节为cc，而cc对应的十进制ASCII码值为204，超过了127，故显示为问号？** （ASCII码表共有256个字符）：**不在0~127之间，便会显示 问号？**

**2个cc对应一个烫(即cc cc)： 因为1个汉字(烫)占2个字节**

(4)malloc申请堆空间的处理方式(微软、Linux)

1)微软VS对于malloc申请的堆空间会初始化为**cdcd，即屯**(cd cd).

2)Linux下malloc申请的空间是**脏数据**，里面存的是上一个用过的数据，即之前占用此空间的数据。

(5)**Gnu/Linux操作系统用的是Linux内核；Windows操作系统用的是Windows NT内核**；

苹果(手机、mac的操作系统) 用的是Unix内核

Windows下函数的栈空间大小是1M

(6) **sln文件**:在开发环境中使用的解决方案文件。它将一个或多个项目的所有元素组织到单个的解决方案中：此文件存储在父项目目录中，即解决方案文件，是一个或多个.proj（项目）的集合。(用sln文件来打开VS，包含了解决方案下的所有的项目)

(7)在.h头文件中，写好所有函数的声明。 (好处是：新手去了解项目时，可先去头文件中看所有的函数)

3.(1) go语言主要优势：在于协程编程（内核中只有一个线程，不断地跳转执行）

(2) **先定义的变量在高地址，后定义的变量在低地址。**

(3) arr[10]={0}; 注意：一个数组中： **&arr=arr** (**值相等**，均为数组首元素的地址)

(4) 调用函数时，若**实参是一维数组名，传递的只是地址，并不会传递数组的长度**；**形参是个指针a，大小为4个字节**，故在形参中定义时不用写数组长度；若要传数组长度，需要再单独传递过去。

注：形参的 &a并不等于传递来的地址值，指针a的内容才是传递过来的地址值。

即**形参中的 &a != a**

4.C语言不可跨平台，可移植性强(相对于汇编)：C是运行速度最快的高级语言

Java可以跨平台，Java及其他脚本语言中没有指针，无法访问物理地址，所以系统中的驱动都需要用C或C++进行编写。

程序执行的过程就是内存的变化过程。(学java就不用关注内存变化，有自动回收)

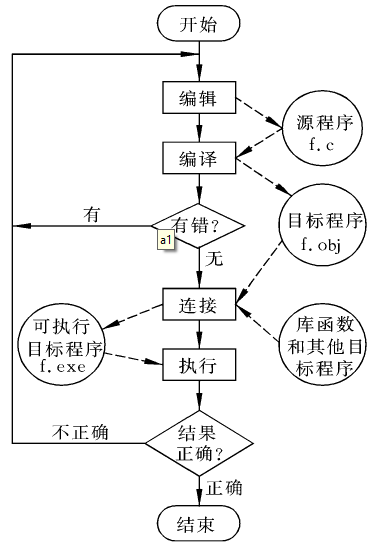
5．system(“pause”);

pause:windows下的批处理命令，即Windows的脚本命令。

**system(“pause”);** system就相当于用C语言嵌套了Windows的脚本：即C语言system()嵌套其他语言pause； C语言还可以嵌套其他语言，如python

6. VS中的目标程序.o文件(如main.obj)，里面都是二进制： **why目标程序已经变成了机器码即0101...但还是不能执行那？**

因为里面的函数(如printf函数)还只是一个符号，还不是一个地址，还要**通过链接把符号替换成1个地址，再根据地址跳转到printf函数中才能执行**：**没链接之前函数仅仅是一个符号。**



7.

我们自己写应用程序时，命名变量第一个字符用字母

写中间键时，命名变量第一个字符用的是下划线 \_

**计算机只能存储二进制，即0和1，对应着高低电频。**

十进制：10个数，0到9； 八进制：8个数，0到7； 十六进制：16个数，0到15

设计补码是为了来表示负数的，用于减法(计算机只有加法器)，正数只有原码。

**加法器，就是一个三极管：高电平1+低电平0=高电平1； 高电平+高电平：进位**

**负数：在内存中用二进制补码存储，最高位是符号位。**

**EOF即为-1，** #define EOF -1

8.

char 1字节 -2^7 ~ 2^7-1

int 4字节 -2^31 ~ 2^31-1 //**signed:还有1位是符号位**，故不是32，为31

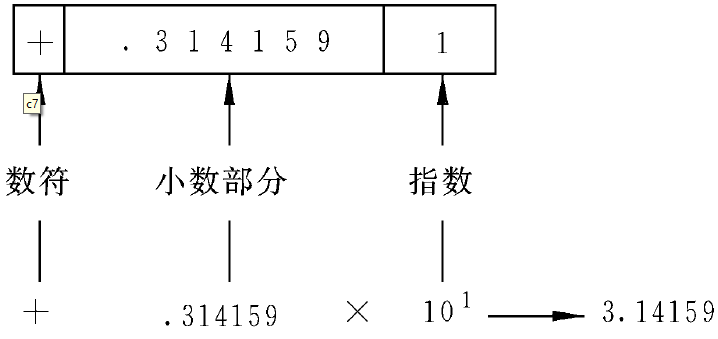
unsigned int 4字节 0 ~ 2^32-1 //**还要减1**后才为最大值(取不到31/32次方)

short 2字节 -2^15 ~ 2^15-1

unsigned short 2字节 0 ~ 2^16-1 //无符号位

long 4字节 -2^31 ~ 2^31-1 //Win32控制台是4字节，在64位是8字节

unsigned long 4字节 0 ~ 2^32-1



在内存中存储浮点型时：

(1)数符占1位：0表示正数，1表示负数

(2)小数部分存的是314159对应的二进制，并不会存小数点.(小数点是便于人们理解)

(3)指数部分并不会存幂次1,因为电脑不能直接算10的幂次，故为了表示10的1次方，**计算机存的是2的幂次，其大小无限接近于10的幂次：所以说浮点数是一个近似值。**

(4)float浮点数有效数字(即精度)：从第1个不为0的数一直到最后:最多能表示6 ~ 7位

如float f=12345678900；可表示为f=1.2345678900e10；故第7位之后,即7之后的数字8、9、0、0则会发生精度丢失(**float精度：最多能表示6 ~ 7位有效数字**)。为防止精度丢失，变量f要设成double类型。(**double精度：最多能表示15 ~ 16位有效数字)。**

(5)浮点计算用协处理器完成：GPU中都是浮点处理器。

**浮点数要注意：精度丢失、浮点数判断是否相等问题。 整型数要注意：溢出。**

**浮点数判断相等不能用 = =判断**(浮点数用的是2的幂次，结果是近似值，故不能真正相等

故**用减法来判断精度，即if ( f1-f2 >-1e-6 && f1-f2 <1e-6 )； （float精度有6位）**

9.

\n 即换行(区别于 ’\r’)

\t 即tab键(一般为4个空格)

**\r 回车，即回到了行首**

**\b 退格，光标向前移1个，用于删除字符操作：\b \b** (中间有1个空格)

\\ 打印反斜杠

**\\n 打印\n**

**\\\\ 打印两个反斜杠**

10. 重点！！！！！scanf原理、缓冲区原理：行缓冲

(1) int i;

char c;

scanf(“%d”,&i);

scanf(“%c”,&c);

因为第一个scanf是%d格式，故只读取整型，不会匹配\n，故\n仍留在缓冲区内。第二个**scanf是%c格式，故程序运行到第二个scanf时，并不会卡住，会直接读取缓冲区中的\n。**

**%c时：输入的数字都会按字符处理，多个数字只能1个1个读取(即每次只能读取一个字符)。**

(2) int i,j;

scanf(“%d”,&i);

scanf(“%d”,&j);

程序运行到第二个scanf时，**会卡住等待用户输入:因为其是%d格式，并不会读取缓冲区中的\n，且会忽略并删除当前缓冲区的\n** (当%d时，scanf会忽略先导的空白符。不管是有几百个回车也好，几万个空格也罢，只要它们连续地出现在缓冲区的开头，就统统忽略他们)

**在缓冲区中：**

**%d、%f：会忽略空格、换行\n**

**%c：什么都不忽略**

(3)**在键盘上输入的全部是字符，之所以会变成浮点数或整型数，是因为scanf处理的(其根据格式%f、%d把字符进行转换为浮点数或整型数)**

(4) **每次printf时要加上\n的好习惯： 因为加\n的话，标准输出会立刻强制刷新，输出结果能马上推到屏幕上。否则不加\n,要等缓冲区快满时才会刷新，故有可能输出结果仍在缓冲区内，不能马上推到屏幕上去。 (因为printf、scanf都是行缓冲，即\n)**

(5)int i,ret;

**while((ret=scanf(“%d”,&i))!=EOF)**

{

printf(“%d\n”,i);

}

如scanf第一次输入为2，第二次输入字符a，之后便会一直死循环打印2。

因为：第二次输入字符a，此时缓冲区中为 a\n ,因为并不是整型，所以无法读取a，故scanf返回值是0，但0也不等于-1(EOF)，仍会进去while循环体，故仍打印上一次i的值即2。接着，下一次循环scanf去匹配缓冲区时，缓冲区中依旧为 a\n，仍无法读取a，便又进入while循环体，所以会陷入了死循环，一直打印2。(%d时会忽略空格、换行\n,故不会刷新)

解决办法：scanf在下一次匹配时:先清空标准输入缓冲区 fflush(stdin)、rewind(stdin)可修改为： **while(fflush(stdin), (ret=scanf(“%d”,&i))!=EOF)**

故在下一次循环时，会先执行fflush(stdin) 刷新缓冲区(**，逗号表达式**)，然后再去执行操作 ret=scanf(“%d”,&i))!=EOF

(7) C的缓冲三种特性：

**1) 全缓冲**：在这种情况下，当填满标准I/O缓存后才进行实际I/O操作。全缓冲的典型代表是对磁盘文件的读写。

**2) 行缓冲**：在这种情况下，当在输入和输出中遇到换行符时，才执行真正的I/O操作。这时，我们输入的字符先存放在缓冲区，等按下回车键换行时才进行实际的I/O操作。典型代表是标准输入(stdin)和标准输出(stdout)。

**3) 不带缓冲**：也就是不进行缓冲，标准出错情况stderr是典型代表，这使得出错信息可以直接尽快地显示出来。(只要往缓冲区放一个数据，便会立马推到屏幕上去)

11. **短路运算(程序员比较懒，不想多写if判断)：**



**逻辑运算时：**

(1) **逻辑与&&**： 当前面表达式假的，后面表达式不会打印执行。

i=1;

i==0&&printf(“sysytem is error\n”); //后面的不会打印

反之若i=0时，故前面为真，后面会打印

(2) **逻辑或||**： （和逻辑与相反）前面表达式为真的，后面表达式不会打印执行。

i=1;

i==1 || printf(“system is error\n”); //前面为真，故后面不会打印

12.

当程序有 乘2、除2操作时：用左移、右移来完成（更高效）。

申请1G内存时，用**malloc(1<<30)**相比**malloc(1024\*1024\*1024)** 更高效，因为计算机执行乘法操作时，相比移位操作是很慢的。

**左移：为了乘2，高位丢弃，低位补0**

**右移：为了除2，正数：低位丢弃，高位补0**

**负数：低位丢弃，高位补1**

13.

C语言提供了13种类型的运算符，如下：

(1)算术运算符 （+ - \* / %）

(2)关系运算符 （＞ ＜ ＝＝ ＞＝ ＜＝ ！ ＝）

(3)逻辑运算符 （！ ＆＆ ||）

(4)位运算符 （<< >> ~ |∧＆）

(5)赋值运算符 （＝及其扩展赋值运算符）

(6)条件运算符 （？：）

(7)逗号运算符 （，）

(8)指针运算符 （\*和＆）

(9)求字节数运算符（ｓｉｚｅｏｆ）

(10)强制类型转换运算符（(类型)）

(11)分量运算符（．->）

(12)下标运算符（［ ］）

(13)其他 （如函数调用运算符（））

**！(单目) > 算术运算符 > 关系运算符 > && > || > 赋值运算符**

注：关系表达式、逻辑表达式：结果只为0或1(真或假)

14.重点算法！！！按位异或^

**一个数和0异或：为自身**

**自己和自己异或：为0**

**异或满足交换律！！可用于：在一堆数中，找到只出现1次的数。**

如：int a[5]={13,6,7,13,6},result=0;

for(int i=0;i<5;i++)

{

result=result^a[i];//交换后，再各个异或，最后0和7异或为7

}

printf(“%d\n”,result);//最终打印的为7(只出现了一次)

15.

(1)用反斜杠\，可将2行代码拼接成1行。

**(2)在内存中，数据均以补码的二进制形式进行存储**

(3)求三个数大小：用三目运算符

(1)max=(i>j?i:j)>k?(i>j?i:j):k;

(2)max=i>j?(i>k?i:k):(j>k?j:k);

16. **逗号运算符**：**用于做多次操作**。**其优先级最低，逗号表达式的值为最右边表达式的值**。

(1) i=5；j=i,i+5; 最终j等于5 优先级最低，因为赋值运算符优先级高，先执行j=i

(2) i=5; j=(i,i+5);最终j等于10 执行逗号运算符：故括号内逗号最右边为最终值真

17. int i=-1,j;

j=i++>-1;

打印结果为j=0,i=0，因为i++>-1结果为假即0，故j=0。

18.

**switch简写形式：相同的行为，只用写1次**。如下：

int main()

{

int year,mon;

while(scanf("%d%d",&year,&mon)!=EOF)

{

switch (**mon**)

{

case 2:printf("%d is %d days\n",mon,28+(year%4==0&&year%100||year%400==0));break;

case 4:

case 6:

case 9:

case 11:**printf("%d is 30 days\n",mon);break; //4、6、9月仅一次**

case 1:

case 3:

case 5:

case 7:

case 8:

case 10:z

case 12:**printf("%d is 31 days\n",mon);break; //1、3、5、7、8、10、12月仅一次**

default:printf("error mon\n");

}

}

system("pause");

}

19.

只有代码段：数据作为一条指令，CPU(通过译码器)是可执行的

其余段(如堆、栈…)：都是数据，只能读写，不能执行

函数调用的本质：值传递(即实参的值赋值给形参)

20.

**goto语句——无条件转向语句**，使用方法为goto 语句标号；语句标号的命名规则与C语言变量的命名规则一致。

**goto才是循环的本质**，对应于汇编中的jmp跳转，C中的while，do while和for等循环在程序编译时，都要拆解为汇编的jmp。

21.

**do-while语句的特点:先执行循环体**，然后判断循环条件是否成立。

一般形式: do {

循环体语句

}

while (表达式)；

执行过程：先执行一次指定的循环体语句，然后判别表达式，当表达式的值为非零(“真”) 时，返回重新执行循环体语句，如此反复，直到表达式的值等于0为止，此时循环结束。

22. 重点continue！！！

在**for循环中，continue后面的语句不会得到执行，但表达式3依然会执行**。

continue跳过的是语句2中对应的内容！！！

23.

字符串常量一定是以‘\0’作为终止符，‘\0’之前的都是字符串的内容。

strcpy(d,c); //c中字符串**复制**给d，包括**’\0’也会复制过去**。

24.

指针就2个实用场景，一定要用指针：

1. **传递**：如要改变实参的值，必须地址(指针)传递：即保证实参i和形参i的地址相同才能改变；若实参传递的是i的值，即**值传递时，形参i和实参i并不分配在同一个地址**，故形参i改变，并不会改变原来的值i。

(2)**偏移**：我们把对指针的加减，称之为指针的偏移，加就是向后偏移，减就是向前偏移。

注：加一是指:加1个基类型的长度，如**double \*p，p+1就是偏移sizeof(double)的长度。**

void类型的指针不能偏移。

25.**指针与自增自减运算符：重点易错**

(1)int a[3]={2,7,8};

int \*p;

int j;

p=a;

**j=\*p++;**

printf("a[0]=%d,j=%d,\*p=%d\n",a[0],j,**\*p**); // 2 2 7

**j=p[0]++;**

printf("a[0]=%d,j=%d,\*p=%d,a[1]=%d\n",a[0],j,**\*p** ,**a[1]**); // 2 7 8 8

(2) int a[3]={2,7,8};

int \*p;

int j;

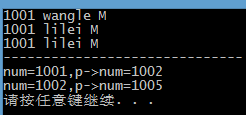
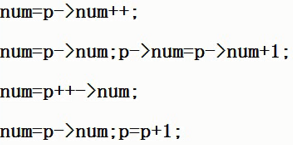
p=a;

**j=(\*p)++;**

printf("a[0]=%d,j=%d,\*p=%d\n",a[0],j,\*p); // 3 2 3

j=p[0]++;

printf("a[0]=%d,j=%d,\*p=%d,a[1]=%d\n",a[0],j,\*p,a[1]);// 4 3 4 7

(3)   

**26.内存泄露、野指针、free(p)、p=NULL**

(1)**malloc堆空间后不free：内存泄漏(不释放所申请的堆空间，则其他进程便不能再使用此空间了)**。 平时自己不free，是因为进程结束后操作系统会自己free回收的。但是工作以后的服务器是长时间不退出一直在后台运行的，故一定要自己free。

(2) char \*p;

scanf("%d",&i);//要申请的空间大小

p=(char\*)malloc(i);//申请堆空间

strcpy(p,"malloc success");

puts(p);

free(p); //**此时free的是p对应的堆空间，而不是free p所在的栈空间,故p中仍存储着之前申请堆空间的地址值。要改变p，则要free(&p)**

(3)free(p);p=NULL; 由(2)知**free(p)后，p中仍存着堆空间的地址值，为了避免其他人再用此野指针，故要赋值为NULL。**

free以后p不赋NULL，若仍拿着指针p去访问对应空间(仍存着之前的脏数据)：**野指针**

(4) free(p)后，不赋为NULL的后果

int \*p,\*p1,p2;

p=(int\*)malloc(4);

\*p=1;

p1=(int\*)malloc(4);

\*p1=2;

free(p);

//**free的是堆空间，指针p所在的栈空间还在，故p此时仍指向之前的堆空间**

//p=NULL; //free后，若此时p不赋为NULL（赋NULL是为了之后能抛出异常）

p2=(int\*)malloc(4); //**p2此时申请的空间为原指针p所指的空间**

\*p2=3;

\*p=100; //**p和p2所指的是同一空间，故\*p改变后，\*p2也改变了**

printf("\*p1=%d,\*p2=%d\n",\*p1,\*p2);//因为原空间未赋为NULL，故此时 **\*p2=\*p=100**

(5)内存泄露

**char \*p=(char\*)malloc(20);**

**p=”hello”; //内存泄漏**

因为刚开始p指向malloc申请的空间，之后p又指向了文字常量区，而之前malloc申请的空间也没有free(p)掉，故造成了内存泄漏：**故赋值要通过strcpy(p,”hello”)，而不能改变其指向;**

27.

(1)我们定义的整型，浮点型，字符型变量，数组变量，都在栈空间，栈空间的使用是在编译时确定大小。如果使用的空间大小不确定，那么我们就需要使用堆空间。

1. **栈是操作系统原生态提供的数据结构，计算机会在底层对栈提供支持**：分配专门的寄存器存放栈的地址(即栈顶指针)，压栈出栈都有专门的指令执行，决定了**栈的效率比较高**。

**Windows中的函数栈是1M： 即1M=1024k=1048576字节；1048576/4 (如int)=262144：即栈中只能存储262144个整型数，超出此大小就是爆栈Stack overflow**

(3) **堆则是C/C++函数库提供的**，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法，在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

**堆空间不会因为函数执行结束就释放（只能通过free），而栈空间在函数结束时就会释放**

28.

int b[4]; //b是指针常量

int \*p;

p=b; //正确

p=&b; //错误：&b为数组指针：int \* 和 int(\*)[4] 间接级别不同。

29. 二维数组名并不是二级指针（而是数组指针），

**要区分：数组指针(一级指针) 和 二级指针**

记 死: 1)数组指针和二维数组是对应的。

2)二级指针只服务于一级指针的传递与偏移。

即二级指针的偏移服务于指针数组, 二级指针和指针数组是对应的

**(1)一维数组名：存的是一级指针(指针常量)。**

**(2)二维数组名：存的是数组指针(一级指针)。**

**(3)指针数组名：存的是二级指针。**

**(4)一维、二维数组的地址值:**

int a[4]; //a=&a;

int b[3][4]; //b=&b=b[0]=\*b;

30. **偏移：是根据 要偏移的变量的类型 所对应的大小 来偏移的**

int a[3][4]={{1,2,3,4},{2,4,6,8},{1,2,3,4}};

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

1. 若a的地址： 0x2000

a+1的地址： 0x2010

**&a+1的地址：0x2030 (若是&偏移：移到a数组本身的末尾了)**

**&a：二维数组指针：偏移是根据类型对应的大小来偏移**

1. 若b的地址： 0x2000

b+1的地址： 0x2004

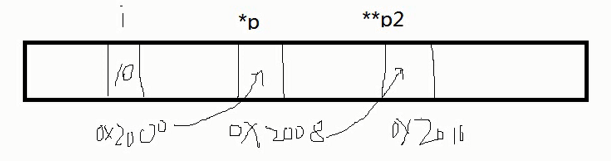
**&b+1的地址：0x2010 (若是&偏移：移到数组b本身的末尾了)**

**&b：数组指针 (一维数组指针)：偏移是根据类型对应的大小来偏移**

31. **重点二级指针 \*\*p2**

**1、二级指针的传递**

(1)



int i=10;

int \*p=&i;

int \*\*p2=&p;

printf("\*\*p2=%d,\*p=%d",\*\*p2,\*p); //结果均为10.

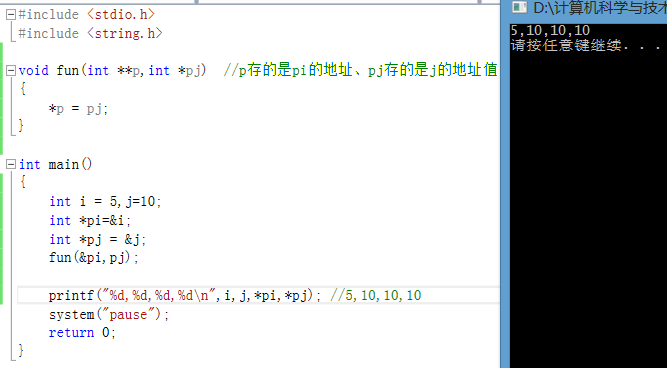
(2)

要想在子函数改变一个变量的值，必须把该变量的地址传进去。

**1)要想在子函数 改变一个指针变量的值，必须把该指针变量的地址传进去。**

**2)二级指针常用于:二级指针的传递:即在子函数里改变主函数中某个一级指针变量的值（即指向）**

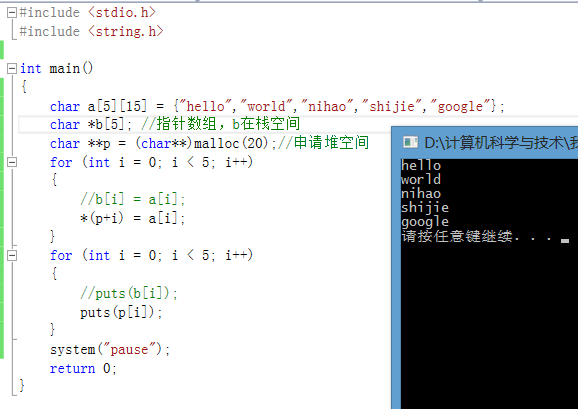
例：要仅改变 \*pi的值，而不改变i的值：通过二级指针实现。



1. **二级指针往往写在形参位置，很少定义在主函数。**
2. **一级指针、二级指针初始化：均只能赋值为某个变量的取地址，而不能直接赋某个确定的地址值**。
3. **二级指针的偏移服务于指针数组,即二级指针和指针数组完美对应。**
4. **二级指针每次偏移+1: 必只能偏移4个字节：因为是指针，如sizeof（char\*）;**

**2、二级指针的偏移：**

**(1)应用于在链表上的索引式排序、以及在堆上来模拟实现指针数组(栈上)**



**(2)二级指针p每次偏移+1: 只能偏移4个字节：因为是指针，sizeof（char\*）;**

1. **char\*\*p; //char\*(\*p); p是指针，指向char\*型数据。**

**char \*p[5]; 把[]解开，等价于 char \*(\*(p+5)),即指针数组对应于二级指针**

**(4)索引式排序**：实际在淘宝购物过程中，大家搜索的商品信息存在内存中，如果以某个查询条件搜索商品，淘宝需要把商品按你的要求进行排序，比如价格从低到高，这时交换内存中商品的信息会极大的降低效率，因为不同用户会有不同的查询需求，每件商品本身的信息存储又较大，假如我们让每个指针指向商品信息，排序比较时，我们比较实际的商品信息，但其**交换的是指针，这样交换成本将会极大降低**，这种思想称为索引式排序

注意：链表不能像数组一样很方便地进行交换，故要通过索引式排序(即通过改变指针指向)来实现数据次序的交换。

32.函数指针:传递一种行为，可实现类似C++的多态、重载

定义函数指针p，将函数b赋值给p，为什么可以赋值呢，其实是因为**函数名b本身存储的就是函数的入口地址** （**而函数指针就指向函数的入口地址**）

再将p传递给函数a，相当于把一个行为传递给函数a，之前我们传递给函数的都是数据，**通过函数指针可以将行为传递给一个函数，这样我们调用函数a可以就可以执行函数b的行为**，当然也可以实现执行其他函数的行为。

**面向接口编程**：a函数本身不用动，直接去执行b函数。 如qsort()。

函数指针声明如下：

**返回值类型 ( \***[**指针变量**](https://baike.baidu.com/item/%E6%8C%87%E9%92%88%E5%8F%98%E9%87%8F)**名) ([**[**形参**](https://baike.baidu.com/item/%E5%BD%A2%E5%8F%82)**列表]);**

(1)搞嵌入式则要深入学习函数指针、回调函数； 搞Linux系统编程和C++，走C++后台开发工程师路线，则函数指针并不重要，因为C++的重载使用起来更加方便。

(2)

void b()

{

printf("I am func b\n");

}

void a(void (\*p)())

{

p();

}

//定义**函数指针p，初始化只能赋函数名**

int main() {

**void (\*p)();** //**定义了一个函数指针变量p**

p=b; //**函数指针的返回值及入参要与函数保持一致**

a(p);

system("pause");

return 0;

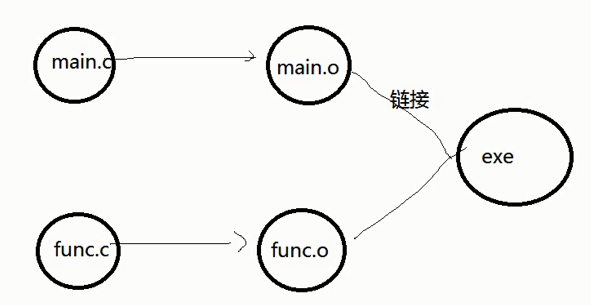
}

(3)

**int (\*p)()**, p和\*连在一起，故p是指针，即函数指针，后面为()。

**int (\*p)[N]** ，p和\*连在一起，故p是指针，即数组指针：指向包含4个元素的一维数组。

33.函数



(1)

VS中的目标程序.o文件(如main.obj)，里面都是机器指令二进制，此时函数还只是一个符号，还不是一个地址，还要通过链接把符号替换成1个地址，再根据地址跳转到函数中才能执行(即通过链接生成exe文件)：没链接之前函数仅仅是一个符号。

(2)

**生成：增量编译，检查所有.c文件，但只生成改变了的.c文件使之成为obj文件即.o文件**（不会去生成其他 .c文件，即其他的 .o文件并没有改变）,再和其他 .o文件进行链接。

重新生成：所有的 .c文件都重新生成 .o文件

(3)点击本地Windows调试器：默认是生成。

(4) 一个Ｃ程序由一个或多个程序模块组成，每一个程序模块作为一个.c源程序文件。对于较大的程序，通常**将程序内容分别放在若干个.c源文件中(如main.c、func.c)，一个人写1个.c文件分工明确，再由若干.c源程序文件组成一个C程序**。这样便于分别编写、分别编译，提高调试效率。一个.c源程序文件可以为多个C程序公用。

一个源程序文件由一个或多个函数以及其他有关内容（如命令行、数据定义等）组成。一个源程序文件是一个编译单位，**在程序编译时是以源程序文件为单位进行编译的，而不是以函数为单位进行编译的。main.c和func.c分别单独编译**，在链接成为可执行文件exe时，main中调用的函数printstar和print\_message才会通过链接去找到函数定义的位置。

所有函数都是平行的，即在定义函数时是分别进行的，是互相独立的。**一个函数并不从属于另一函数，即函数不能嵌套定义(苹果的swift可以嵌套定义)**，即不能在一个函数中定义另一个函数，只能声明或调用。函数间可以互相调用，但不能调用main函数。main函数是系统调用的。

34. 尽量不要用全局变量！！！

int i=10; //全局变量

void print()

{

printf("print i=%d\n",i);//**i=10，当i重名时用就近原则，故打印全局区i的值(离得近)**

}

int main()

{

int i=15; //**此时在栈区又定义了一个i，与全局区的重名了**

printf("main i=%d\n",i); //i=15

print();

system("pause");

return 0;

}

C语言很灵活：全局变量i在数据段，局部变量i在栈区，**因为在内存的不同区域，故可以重名，当打印时，如果局部变量与全局变量重名，则会系统采用 就近原则。**

35. **协程编程：setjmp、longjmp：用于直接返回，不用再返回中间的函数。**

#include <setjmp.h>

jmp\_buf envbuf; // **jmp\_buf：结构体类型名**、**全局变量envbuf：保存程序执行的上下文**

void b()

{

printf("I am b func\n");

**longjmp(envbuf,5); //跳转回到setjmp位置(保存现场的位置)**

}

void a()

{

printf("before b(),I am a func\n");

b(); //**b函数调用完，不会再返回此处。**

printf("after b(),I am a func\n"); //不会打印此句

}

void main()

{

int i;

**i=setjmp(envbuf); //保存进程执行的上下文**

**if(i==0) //防止死循环**

{

a(); //第二次返回值为5，不会进入循环了

}

system("pause");

return;

}

(1)

setjmp函数：保存程序执行的上下文，当第一次调用setjmp(),它的返回值为零。之后调用longjmp() , longjmp()的第二个参数即为setjmp()的返回值。

(2)

longjmp函数 : 使程序回到setjmp()的调用处继续执行。longjmp函数没有返回值

(3)

协程和go语言：就是用setjmp和longjmp实现的（可以实现从下层函数跳转）

Why使用go协程？ 因为小白对线程的使用没有炉火纯青，切换的调度策略不好，最终效果就不好； 用协程的话，调度都已设计好了。

36.

(1)爆栈stackOverflow (1)数组超过了定义的长度 (2)递归次数太多(不断地压栈)

(2)scanf为什么要取变量的地址传进去？ 因为值传递：在scanf被调函数中要改变主函数中某变量的值，则必须把此变量的地址传过去。

(3)typedef目的：代码即注释。见代码知道含义。

37.

**结构体：有时需要将不同类型的数据组合成一个有机的整体，以便于引用**。如：一个学生有学号/姓名/性别/年龄/地址等属性，如果针对学生的学号，姓名，年龄等都单独定义一个变量，那么当我们有很多个学生时，变量就难以分清，因此C语言提供结构体来管理不同类型的数据组合。（学习结构体也是为了灵活地学习各种数据结构）

1. **结构体对齐**：CPU通过数据总线(Win32应用台：32位 / 64位)从内存中获取数据,**为了提高CPU取每一个数据的效率（即空间换时间）**，避免小于CPU数据总线位数的数据还要分2次才能取到CPU里

对齐：32位数据总线：一次拿4个字节(每次均从4的倍数位置往下取4个字节)；

64位数据总线：一次拿8个字节(每次均从8的倍数位置开始取8个字节)。

故**每次算结构体大小时少用手算(因为有对齐机制可能算不准)，要用sizeof(结构体类型)。**

(2)例如：

struct student

{

int num;

char name[20];

char sex;

int age;

float score;

char addr[30];

};

struct student s1={1001,”Lilei”,’M’,18,98.5,”BeiJing”};

则：

sizeof(s1)=68; (结构体对齐)

4+20+4+4+4+32(均要为4的整数倍)=68.

38.

malloc()仅申请空间不会初始化(微软会处理成 cc)、 calloc()申请空间后会初始化为0

链表的每个结点有两部分组成：数据、指针(即为结点类型的指针next)

链表的：头插法、尾插法、有序插入。

39.

**typedef** struct student{

int num;

float score;

struct student \*pNext;

}stu,\*pstu;

**stu（结构体变量类型）**

**pstu（结构体指针类型） 等价于 struct student\***

40.共用体（联合体）：主要用在C语言的可变参数

使几个**不同的变量共占同一段内存空间**的结构称为“共用体”类型的结构。且**共用体变量所占的内存长度等于最长的成员的长度**（区别结构体）

注意：

(1)定义共用体类型变量的一般形式为：

union 共用体名

｛

成员表列

｝变量表列；

1. 同一个内存段可以用来存放几种不同类型的成员，但在每一瞬时只能存放其中一种，而不是同时存放几种。 只有先定义了共用体变量才能在后续程序中引用它，有一点需要注意：不能引用共用体变量，而**只能引用共用体变量中的成员**。
2. 共用体变量中**起作用的成员是最后一次存放的成员**，在存入一个新的成员后原有的成员就失去作用。
3. 共用体变量的地址和它的各成员的地址都是同一地址。
4. 不能对共用体变量名赋值，也不能企图引用变量名来得到一个值， 而且还不能在定义共用体变量时对它进行初始化。
5. 不能把共用体变量作为函数参数，也不能使函数return带回共用体变量，但可以用指向共用体变量的指针作为函数参数和return。

(7) 共用体类型可以出现在结构体类型定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型定义中，数组也可以作为共用体的成员。

枚举：将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。

(1)声明枚举类型用enum :

如：enum weekday{sun，mon，tue，wed，thu，fri，sat};

// enum是关键字； weekday是枚举类型; 括号内的枚举元素均为：枚举常量

(2)定义变量：

enum weekday workday，week-day; //weekday枚举类型， workday、week-day:枚举变量

enum{sun，mon，tue，wed，thu，fri，sat｝workday;

变量值只能是sun到sat之一：即枚举变量只能给其赋值枚举常量。

(3)在Ｃ编译中，对枚举元素按常量处理，故称枚举常量。它们不是变量，不能对它们赋值。

(4) 枚举元素作为常量,其是有值的，Ｃ语言编译按定义时的顺序使它们的值为０,１,２…

(5) 枚举值可以用来作判断比较。

(6) 一个整数不能直接赋给一个枚举变量。(一般不这样做)

41.

(1)先偏移再free释放指针所指空间,否则先free的话就不能再偏移到原来的下一个位置。

(2) Hash哈希，也即散列表

(3)基于赫夫曼编码的压缩和解压技术

(4)二叉树常被用于实现二叉查找树和二叉堆(堆排)。

**(5)堆排序**（heap sort），是以二叉堆树作为此算法的数据结构。

(6)堆(指堆积) 通常是一个可以被看作一棵树的数组对象。

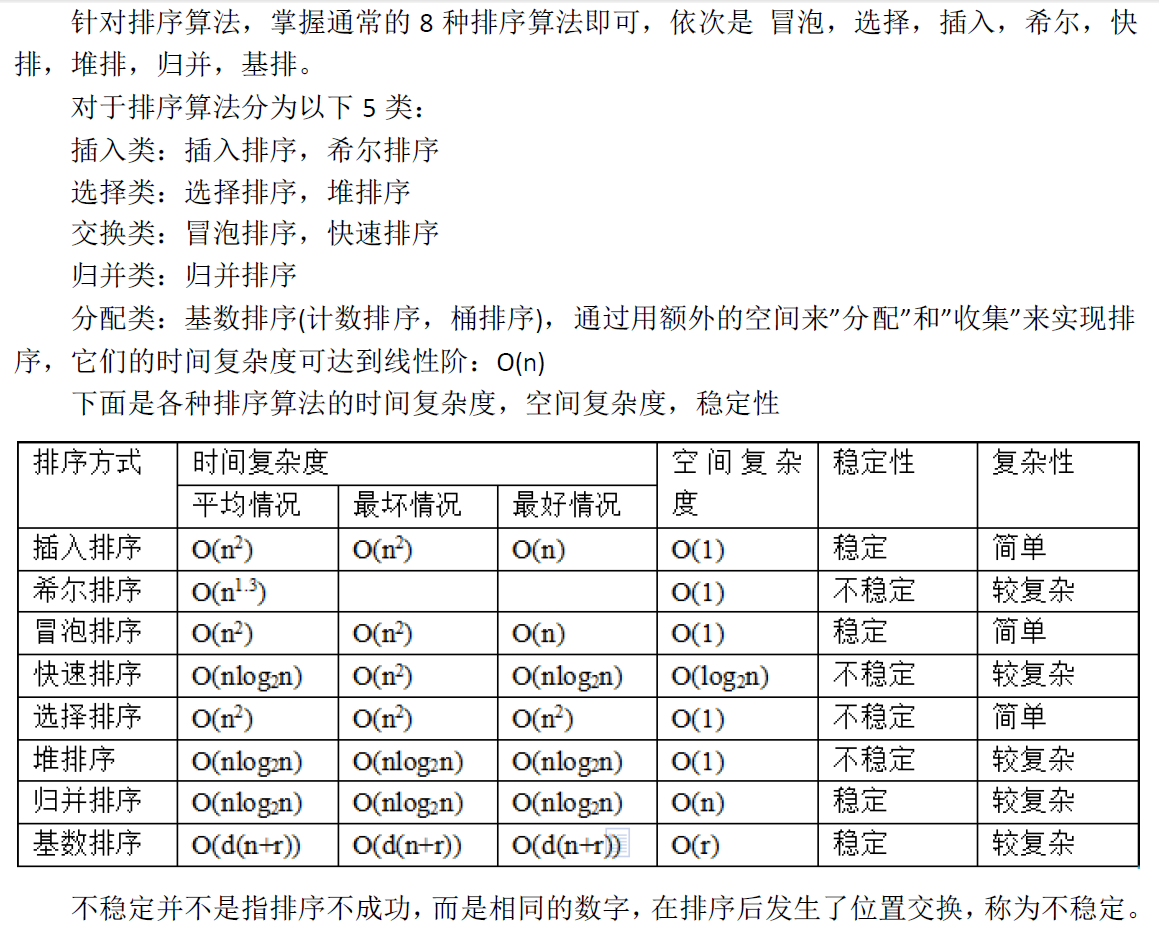
堆：是完全二叉树，分为 大顶堆和小顶堆。

(7)常见数据结构：

**数组Array(顺序存储结构)、链表Linked List(链式存储结构)、 堆栈Stack、队列Queue、 树Tree、 图Graph、 堆积Heap(特殊的完全二叉树)、 散列表Hash**



42.八大排序



1. 选择排序是对冒泡排序的优化：和冒泡一样要比较多次，但不用每一趟都交换。

每趟仅需要交换一次，但要记录每次比较时的最大值的位置，在其和每一个比较完后，将最大或最小的数放到最后。再开始下一趟交换。**选择排序相比冒泡排序优化之处：交换移动数据的次数少了**。

(3)**为什么快排相比于冒泡速度快了许多？** (**二分思想**)

因为快排每次交换是跳跃式的。 每次排序时设置一个基准点：将小于等于基准点的数全放到基准点的左边，将大于等于基准点的数全放到基准点的右边。这样每次交换时就不会像冒泡一样只能在相邻数之间交换，交换的距离就大得多了(跳跃式)，因此总的比较和交换次数就少了，速度自然就提高了。

在最坏的情况下(即本身就有序)，快排也可能是相邻两个数进行交换，故最坏时，快排的时间复杂度和冒泡排序一样，都是O(N^2)，平均时间复杂度是O(nlogn)。

1. **堆：特殊的完全二叉树**，分为大根堆和小根堆。树在优先队列中的应用：即堆的实现；

堆的应用：堆排序、求数列中第K大的数或第K小的数。

优先队列：支持插入元素和寻找最大最小值元素的数据结构；堆：即是一种优先队列的实现。

**建堆(特殊完全二叉树)：不需要去建立二叉树，通过数组即可完成。**

堆排步骤：如从小到大排序

1、把数组调整为大根堆(从最后一个父结点开始)

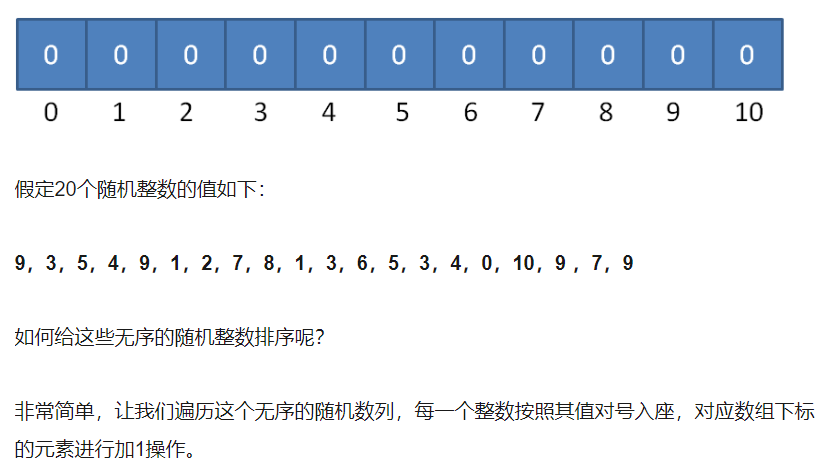
2、把树根和最后一个叶子结点交换

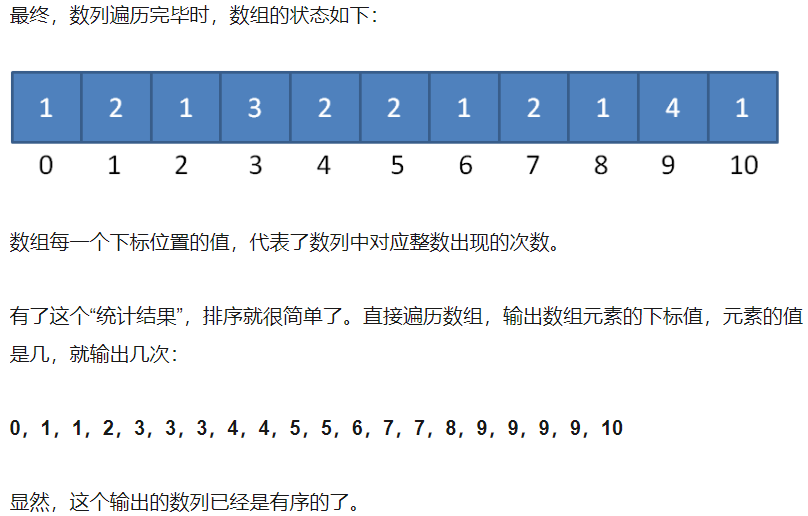
3、把剩余元素继续调整为大根堆

4、重复2,3步

(5)哈希、计数排序：均为空间换时间。哈希时间复杂度为O(1)，一般对于并发量很高的应用，适合用哈希，如12306买车票。

计数排序：<https://www.sohu.com/a/258882297_478315>。其常用于已知一定范围的整数排序，且在取值范围不是很大的情况下，性能甚至快于O(nlog n)的排序。





(6)二分查找(必须要有序)的时间复杂度是：**log n，因为n每次以除以2的速度**，来减少查找的次数。故当知道排序数的范围时，用基数排序更好。

(7)红黑树：查找、插入、删除的时间复杂度都为O(log n)

43.**完全二叉树特性:注意若针对于数组，则是从1开始对应于以下：**

(1)父节点编号为k,则其左儿子编号为2\*k，右儿子编号为2\*k+1

(2)儿子(左儿子或右儿子)编号为x，则其父结点编号为x/2

(3)有N个结点时，则：完全二叉树高度为向下取整[logN]+1，即最多有[logN]+1层结点。最后一个非叶结点(即最后一个父结点)：是第N/2个结点。

44. 重点面试题：

找n个数里边第K大的数(k非常大):

1、快速去重 计数排序:也叫位图 32

2、k个元素建成一个堆，小顶堆，发现一个元素比堆顶要大，放进来

3、k个元素无法放入内存，怎么办，n-k个元素是否能够放入 (若均不能放入时，要用多台机器)

4、要用两台机器来找第k大的数 n个数放到了2台机器：二分思想 log n

a k/2 k 有序k个数

b k/2 k 有序k个数

b[k/2] 删除法

45. 位图

位图的基本概念是用一个位（bit）来标记某个数据的存放状态，由于采用了位为单位来存放数据，所以节省了大量的空间。一般一个int数字要占用32位，如果能用一位就表示这个数，就可以缩减大量的存储空间。一般把这种方法称为位图法，即Bitmap。

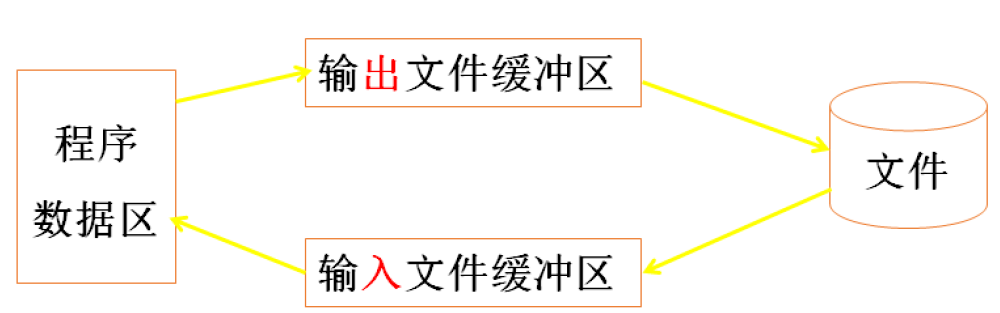
46.

操作系统的开机过程就是：进程的启动过程。

360开机加速：延迟非关键进程的启动速度，故加快了开机时间。即等开机以后再启动那些非关键进程。

47.

文件：文件指存储在外部介质(如磁盘磁带)上数据的集合。操作系统（windows，Linux，Mac均是）是以文件为单位对数据进行管理的。



48.

main函数固定传参格式及作用：

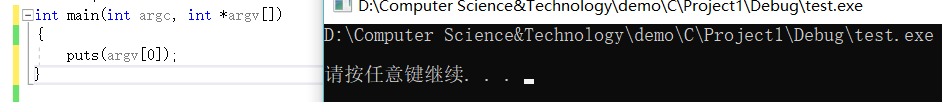
int main(int argc,char \*argv[]) //第二个参数是指针数组

{

puts(argv[0]); //VS通过绝对路径来启动程序。

}

1. 第二个参数是指针数组，数组中的每一个指针元素都指向传参。若没有传参，则只有1个参数是argv[0]为自身启动路径，此时argc为1。如下**没有传参时：打印argv[0]是绝对路径。**



(2)

1) 若传相对路径时:

fopen不用写死了，fp=fopen(“file.txt”,”r+”); 可以写成：

fp=fopen(**argv[1]**,”r+”);

VS找文件的路径是和main.c在同一个路径的

VS中向main函数传参，对项目右键点击-属性-调试-命令参数

2)若传绝对路径时:如：D:\Computer Science&Technology\demo\C\Project1\test\ The\_Holy\_Bible.txt,则fopen要写成：

fopen("D:**\\**Computer Science&Technology**\\**demo**\\**C**\\**Project1**\\**test**\\**The\_Holy\_Bible.txt", "r+"); 注意：**要写2个斜杠，若写1个的话会当成转义字符。**

若在VS中传参时，则写成1个反斜杠。

49.

文件操作如fputc、fputs、fwrite等不是直接写到磁盘上，而是先写到内核的缓冲区中，只有当全缓冲刷新时，才会写回磁盘。

缓冲区什么时候会把文件写到磁盘里？

(1)**用户主动刷新 fflush(stdin); rewind(stdin); ‘\n’;** 主动刷新输入流时

(2)**关闭文件**（关闭前会写进磁盘，否则文件会丢失）

(3)**缓冲区写满**时，会一起送到磁盘去

50.微软的文件操作历史遗留问题：

(1)每次打开文件时，光标都在开头。故每次写入文件时若不偏移位置指针，则都会覆盖掉之前写入到文件中的相应位置的内容，而不会把原有内容向后推。

(2)微软txt文本解析换行时不是\n, 而是\r\n，为了适配HTTP协议的规定，方便开发网页。

(3)**在Windows中：**

**r+(文本)模式写入\n,实际存储的是\r\n；读取时\r\n会变为\n。 故若是文本模式写入的，必须也要是文本模式才能读出来。**

**rb+(二进制)模式写入\n,实际存储的也是\n；读取时\n读取出仍是\n。 即内存上是什么，磁盘上存储的就是什么。**

在Linux这两个都是等价的，都可以，因为Linux换行时\n。

(4)**Windows下读和写之间、写和读之间：都必须用fseek()刷新位置指针**。而在Linux下读写之间不用刷新。

(5) FILE结构体中有位置指针，会记录文件中的位置。fgetc()获取字符并将其改为大写后，此时光标已经在下一个位置了，而要把前一个大写字符写到文件中，则要先fseek()偏移到上一个位置，再去fputc()写入。

**51. fgetc()、fgets()、fread()、gets()对比**

(1) fread()**不会赋结束符**。因为其即能读字符，也能读整型数，故不会赋结束符，否则读整型数时可能造成数组越界。

(2) fgetc()只能读字符，读入一个字符，并把位置指针标识符往前移动。成功，返回值为所读的字符；失败：返回EOF

(3) fgets()用于读字符串，故**会赋结束符**。一次读一行，且每行读到换行\n时停止读取，并在行尾赋结束符’\0’，注意： **fgets读取到buf中的内容中会包括行末尾的'\n'字符**,且**fgets会在所读取的内容最后面添加'\0'**。**如果读入成功，则返回缓冲区的地址；如果读入错误或遇到文件结尾(EOF)，则返回NULL。**

(4)gets()会将行末尾的'\n'或EOF替换成字符串结束符'\0',**gets()读取到buf中的内容不包括行末尾的’\n’字符**。

52.

清屏：system(“cls”);

键盘是字符设备，输入的每一个都是按字符处理的。

scanf()、getchar()、gets()的具体实现就是通过调用fscanf()、fgetc()、fgets()。

**fseek()：是按照磁盘字节大小进行偏移的。其在Windows下的偏移会算上 ’\r\n’，即不会忽略\r。**

getch()函数：当**按回车时，getch()拿到的是 ’\r’,**而不是’\n’。

53.重点！！！！

**fgetc()、fgets()、fscanf()、fprintf()、fread()、fwrite()等均是在输入输出缓冲区中(位于内存)进行操作的，而不是在文件中(位于磁盘)，故读写换行时仍然按 ’\n’处理；**

**而在磁盘上存储换行时，即在文件中存储的是 ”\r\n”，会多存储一个字节。**

**缓冲区与磁盘之间有转换过程：从缓冲区到磁盘：‘\n‘转换为”\r\n”**

**从磁盘到缓冲区：”\r\n”又转换为’\n’**