**C语言学习补充**

**（一）**

2019年9月29日

目录

[1. pow函数应用](#_Toc11864_WPSOffice_Level1) [1](#_Toc11864_WPSOffice_Level1)

[2. 不用中间变量换值](#_Toc32735_WPSOffice_Level1) [1](#_Toc32735_WPSOffice_Level1)

[3. C语言^符号应用](#_Toc25920_WPSOffice_Level1) [2](#_Toc25920_WPSOffice_Level1)

[4. 开方函数 sqrt 的用法](#_Toc30474_WPSOffice_Level1) [2](#_Toc30474_WPSOffice_Level1)

[5. C语言中间接引用运算符的应用（->）](#_Toc3223_WPSOffice_Level1) [2](#_Toc3223_WPSOffice_Level1)

[6. Malloc函数用法](#_Toc31008_WPSOffice_Level1) [3](#_Toc31008_WPSOffice_Level1)

[7. 关于typedef的用法总结](#_Toc16935_WPSOffice_Level1) [4](#_Toc16935_WPSOffice_Level1)

[8. Const用法 (可读不可改)](#_Toc14496_WPSOffice_Level1) [5](#_Toc14496_WPSOffice_Level1)

[9. Getchar()返回int型的原因](#_Toc10297_WPSOffice_Level1) [8](#_Toc10297_WPSOffice_Level1)

[10. C语言三个结束符 EOF ,‘\0’ , '\n'](#_Toc18767_WPSOffice_Level1) [8](#_Toc18767_WPSOffice_Level1)

[11. Makefile文件](#_Toc7070_WPSOffice_Level1) [9](#_Toc7070_WPSOffice_Level1)

[12 . c语言中为什么字面常量与宏定义不占内存空间](#_Toc13421_WPSOffice_Level1) [10](#_Toc13421_WPSOffice_Level1)

[13 . 简单的定义常量的方式：](#_Toc27005_WPSOffice_Level1) [11](#_Toc27005_WPSOffice_Level1)

[14 . 全局/静态变量初始化为0放在bss段还是data段](#_Toc32713_WPSOffice_Level1) [11](#_Toc32713_WPSOffice_Level1)

[15 . C语言局部变量和全局变量同名](#_Toc16546_WPSOffice_Level1) [12](#_Toc16546_WPSOffice_Level1)

[16 . 函数形参重复定义](#_Toc17996_WPSOffice_Level1) [12](#_Toc17996_WPSOffice_Level1)

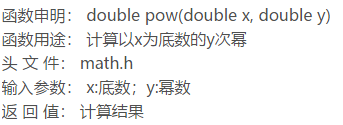
[17 . 宏定义](#_Toc8798_WPSOffice_Level1) [12](#_Toc8798_WPSOffice_Level1)

[18 . ANSI C标准中的几个标准预定义宏：](#_Toc13803_WPSOffice_Level1) [13](#_Toc13803_WPSOffice_Level1)

[19 . #操作符和##操作符](#_Toc30507_WPSOffice_Level1) [14](#_Toc30507_WPSOffice_Level1)

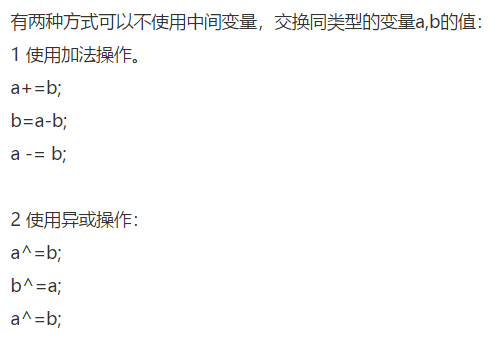
[20 .](#_Toc9764_WPSOffice_Level1) [14](#_Toc9764_WPSOffice_Level1)

1. pow函数应用



double pow(double x,double y)   
这是C语言书中的格式。   
但是x,y可以为别的类型，例如整型，[返回值](http://www.so.com/s?q=%E8%BF%94%E5%9B%9E%E5%80%BC&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)肯定为double举个例子   
double s;   
s=pow(3,2);   
那么s=9.000000

2. 不用中间变量换值



3. C语言^符号应用

（1）C语言^符号只能是异或运算符



（2）异或运算中的特殊运算

a ^ b ^ a = b

B ^ a ^ b = a 取中间量

4. 开方函数 sqrt 的用法

double sqrt(double);

（1）输入输出都是double型

（2）#include <maeh.h>

（3）linux运行需要加 -lm

5. C语言中间接引用运算符的应用（->）

->在C语言中称为间接引用运算符，是二目运算符，优先级同成员运算符“.”。   
用法：   
p->a，其中p是指向一个结构体的指针，a是这个结构体类型的一个成员。表达式p->a引用了指针p指向的结构体的成员a。   
例如：   
struct T   
{   
int a;   
char b;   
}s;   
  
struct T\* p=&s;   
那么，   
p->a相当于s.a。   
显然，有个等价写法：(\*p).a，和p->a完全等效。

6. Malloc函数用法

int \*p;

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

注意：

（1）因为malloc返回的是不确定类型的指针，所以返回之前必须经过类型强制转换，否则编译报错，如：“ 不能将void\*赋值给int\*变量 ”。

（2）malloc只管分配内存，并不会初始化，其内存空间中的值可能是随机的。如果分配的这块空间原来没有被使用过，那么其中每个值都可能是0。相反，空间里面可能遗留各种各样的值。

（3）实参为需要分配的字节大小，如果malloc(1)，那么系统只分配了1个字节的内存空间，这时注意，如果在这块空间中存放一个int值，由于int类型占4个字节，那么还有3个字节未分配空间，系统就会在已经分配的那1个字节的基础上，依次向后分配3个字节空间，而这就占有了“别人”的3个字节空间，“别人”原有的值就被清空了。

（4）分配的空间不再使用时，要用free函数释放这块内存空间。

7. 关于typedef的用法总结

**用途一：**

定义一种类型的别名，而不只是简单的宏替换。可以用作同时声明指针型的多个对象。比如：

char\* pa, pb; // 这多数不符合我们的意图，它只声明了一个指向字符变量的指针，

// 和一个字符变量；

以下则可行：

typedef char\* PCHAR;

PCHAR pa, pb;

这种用法很有用，特别是char\* pa, pb的定义，初学者往往认为是定义了两个字符型指针，其实不是，而用typedef char\* PCHAR就不会出现这样的问题，减少了错误的发生。

用途二:  
用在旧的C代码中，帮助struct。以前的代码中，声明struct新对象时，必须要带上struct，即形式为： struct 结构名对象名，如：

struct tagPOINT1

 {  
    int x;

    int y;   
};

struct tagPOINT1 p1;

而在C++中，则可以直接写：结构名对象名，即：tagPOINT1 p1;

typedef struct tagPOINT  
{  
    int x;

    int y;  
}POINT;

POINT p1; // 这样就比原来的方式少写了一个struct，比较省事，尤其在大量使用的时

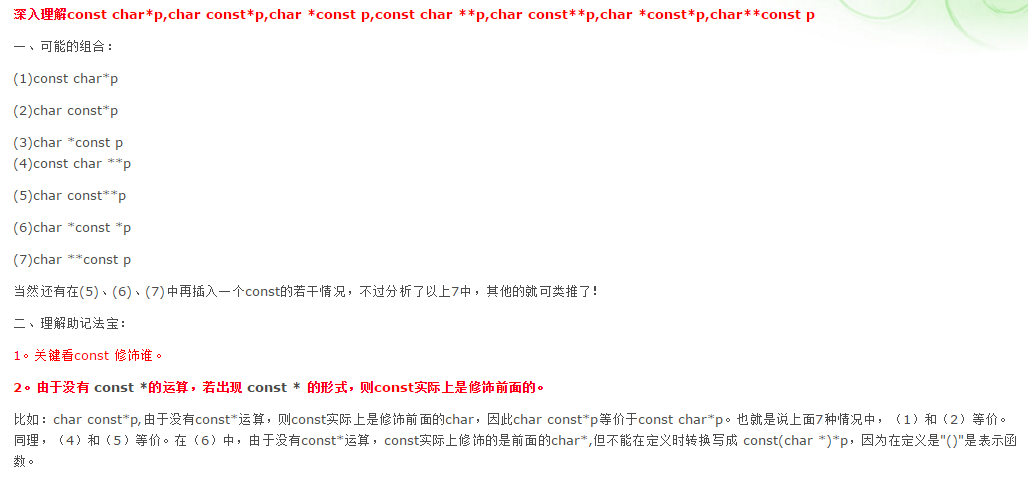
候,或许，在C++中，typedef的这种用途二不是很大，但是理解了它，对掌握以前的旧代

码还是有帮助的，毕竟我们在项目中有可能会遇到较早些年代遗留下来的代码。

8. Const用法 (可读不可改)

1. const int \*p——const用来修饰int \*, \*p的内容不可变，  
   比如你[定义](http://www.so.com/s?q=%E5%AE%9A%E4%B9%89&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)了const int \*p=5;  
   那么你再对p的内容赋值就是不合法的：\*p=6; // no!  
   2. int \*const p=&i;——const用来修饰 p, 指针p的地址不可变，  
   int i=0;  
   int j;  
   int \*const p=&i;  
   p=&j; // no!  
   i=1; // ok!  
   3.const int \*const p=&i——限制了指针指向的内容和指向的地址

<https://www.cnblogs.com/rushuizhijing/archive/2011/08/26/2154737.html>



(1)const char\*p

(2)char const\*p

(3)char \*const p

(4)const char \*\*p

(5)char const\*\*p

(6)char \*const \*p

(7)char \*\*const p

(1)char \*p：p是一个工作指针，可以用来对任意位置 （非系统敏感区域）进 行读操作和写操作 ，一次读写一个字节（char占一个字节）。

(2)const char\*p或者char const \*p（因为没有const\*p运算，因此const修饰的还是前面的char）：可以对任意位置（非系统敏感区域）进行“只读” 操作。（“只读”是相对于char \*p来说所限定的内容）

(3)char \*const p（const 修饰的是p）：只能对“某个固定的位置” 进 行读写操作，并且在定义p时就必须初始化（因为在后面不能执行“p=..”的操作，因此就不能在后面初始化，因此只能在定义时初始化）。（“某个固定的位 置”是相对于char \*p来说所限定的内容）

可以总结以上3点为：char \*p中的指针p通常是”万能”的工作指针 ，而（2）和（3）只是在（1）的基础上加了些特定的限制 ，这些限制在程序中并不是必须的，只是为了防止程序员的粗心大意而产生事与愿违的错 误。

另外，要明白“每块内存空间都可有名字；每块内存空间内容皆可变（除非有所限） ” 。比如函数里定义的char s[]="hello";事实上在进程的栈内存里开辟了6个变量共6个字节的空间，其中6个字符变量的名字分别为：s1[0]、s1[1]、 s1[2]、s1[3]、s1[4]、s1[5]（内容是'\0'）

{

待验证 ： 还有一个4字节的指针变量s 。不过s是“有所限制”的，属于char \*const类型，也就是前面说的 (3)这种情况,s一直指向s［0］, 即（\*s==s[0]=='h'）,可以通过\*s='k'来改变s所指向的 s[0]的值，但不能执行（char \*h＝“aaa”;s=h;）来对s另外赋值。

}

(4)上面的(2)和(3)只是对p进行限定，没有也不能对p所指向的空间进行限定，对于"char s[]="hello";const char\*p=s;" 虽然不能通过\*(p+i)='x'或者p[i]='x'来修改数组元素s[0]～s[4]的值，但可以通过\*(s+i)='x'或者s[i]='x'来修改原数组元素的值－－RAM里内存单元的可读可写属性不因对工作指针的限定而改变，而只会因对其本身的限定而改变。如const char c＝‘A’,c是RAM里一个内存单元（8字节）的名字，该内存单元的内容只可读，不可写。

(5)const char \*\*p或者char const\*\*p ：涉及两个指针p和 \*p。由于const修饰char ，对指针p没有任何限定，对指针\*p进行了上面情况(2)的限定。

(6)char \*const \*p：涉及两个指针p和 \*p。由于const修饰前面的char\*,也就是对p所指向的内容\*p进行了限定(也属于前面的情况(2))。而对\*p来说，由于不能通过"\*p＝..."来进行另外赋值，因此属于前面的情况(3)的限定。

1. char \*\*const p ： 涉及两个指针p和 \*p，const修饰p，对p进行上面情况(3)的限定，而对\*p,没有任何限定

9. Getchar()返回int型的原因

getchar()的返回值一般情况下是字符，但也可能是负值，即返回EOF

这里要强调的一点就是，getchar函数通常返回终端所输入的字符，这些字符系统中对应的ASCII值都是非负的。因此，很多时候，我们会写这样的两行代码：

|  |
| --- |
| char c; c =getchar(); |

这样就很有可能出现问题。因为getchar函数除了返回终端输入的字符外，在遇到Ctrl+D([linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "https://www.cnblogs.com/wxl845235800/p/_blank)下)即文件结束符EOF时，getchar()的返回EOF，这个EOF在函数库里一般定义为-1。因此，在这种情况下，getchar函数返回一个负值，把一个负值赋给一个char型的变量是不正确的。为了能够让所定义的变量能够包含getchar函数返回的所有可能的值，正确的定义方法如下(K&R C中特别提到了这个问题)：

|  |
| --- |
| int c; c =getchar(); |

# 10. C语言三个结束符 EOF ,‘\0’ , '\n'

EOF（End of file）是C/C++里面的宏定义，具体定义式是#define EOF -1，表示的是文件的结束标志，值等于-1，一般用在文件读取的函数里面，比如fscanf fgetc fgets等，一旦读取到文件最后就返回EOF标志并结束函数调用

'\0'是转义字符，值等于0，主要用在C风格字符串的末尾，表示字符串结束标志。通常用在和字符串相关的函数里面，如strcmp strcpy等会用到它

'\n'表示换行符，通常用作一些读取函数的读取结束标志，比如scanf,getchar(),gets()等，一旦遇到'\n'就结束读取并返回

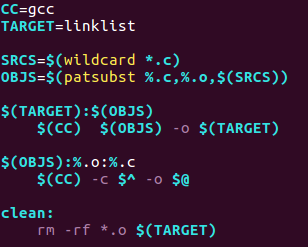
1. Makefile文件

## ****make是如何工作的****

在默认的方式下，也就是我们只输入**make命令**。那么，

1. make会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。
3. 如果edit文件不存在，或是edit所依赖的后面的 .o 文件的文件修改时间要比edit这个文件新，那么，他就会执行后面所定义的命令来生成edit这个文件。
4. 如果edit所依赖的.o文件也存在，那么make会在当前文件中找目标为.o文件的依赖性，如果找到则再根据那一个规则生成.o文件。（这有点像一个堆栈的过程）
5. 当然，你的C文件和H文件是存在的啦，于是make会生成 .o 文件，然后再用 .o 文件声明make的终极任务，也就是执行文件edit了。

**Makeflie文件：**



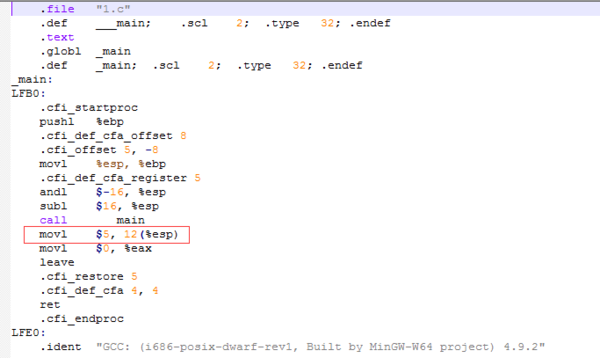
# 12 . c语言中为什么字面常量与宏定义不占内存空间

宏定义不占内存空间，因为宏在预处理阶段就会被替换掉，到了编译的阶段是没有宏存在的，它自然到不了可执行文件中，所以它不占内存空间。

字面常量，占不占空间要看情况，以gcc的处理方法来看，这个设计汇编的问题。



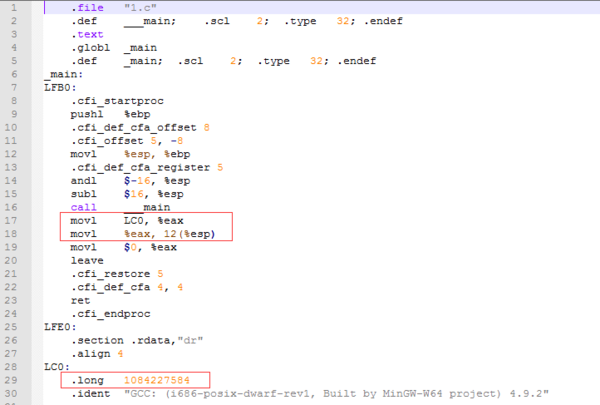
将这段反汇编，得到



可以看到，5是以立即数的形式出现的，所以它不占内存。

然后再将a定义为float：





可以看到，程序的最后定义了5.0的单精度浮点表示，然后将其传入eax寄存器，再赋值给a。

另外，字符常量不占内存空间，字符串常量占内存空间。总之，整形常量是不占空间的，其他占。

13 . 简单的定义常量的方式：

使用 #define 预处理器。

使用 const 关键字。

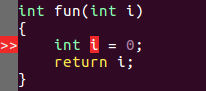
# 14 . 全局/静态变量初始化为0放在bss段还是data段

放在bss区。未初始化可以理解为幅值为0。

15 . C语言局部变量和全局变量同名

C 语言是允许局部变量和全局变量同名的，他们定义后在内存中占有不同的内存单元。如果在同一源文件中，全局变量和局部变量同名，在局部变量作用域范围内，只有局部变量有效，全局变量不起作用，也就是说局部变量具有更高优先级。但是从编程规范上讲，是要避免全局变量与局部变量重名的，从而避免不必要的误解和误操作。

16 . 函数形参重复定义





形参实际上是在函数体内声明的变量。如果在函数体内再次申明相同的变量，函数体内就有两个相同的变量，系统肯定会报错！

17 . 宏定义

1. 带参数

C语言允许宏带有参数。在宏定义中的参数称为“形式参数”，在宏调用中的参数称为“实际参数”，这点和函数有些类似。

对带参数的宏，在展开过程中不仅要进行字符串替换，还要用实参去替换形参。

下面定义一个宏D（a），作用是返回a的2倍数值：

#define D(a) **(**2 \* **(**a**)** **)**

**对于带参宏定义不仅要在参数两侧加括号，还应该在整个字符串外加括号。**

1. 不带参数

#define 宏名 字符串 如 #define DEBUG 10;

右边的字符串也可以省略，如

#define DEBUG

表示宏是空的，在预处理时会按照空来处理，所以也是可以的，不会报错。

1. 终止宏定义作用域

可以用#undef命令终止宏定义的作用域。

18 . ANSI C标准中的几个标准预定义宏：

\_\_LINE\_\_：在源代码中插入当前源代码行号；是个十进制数

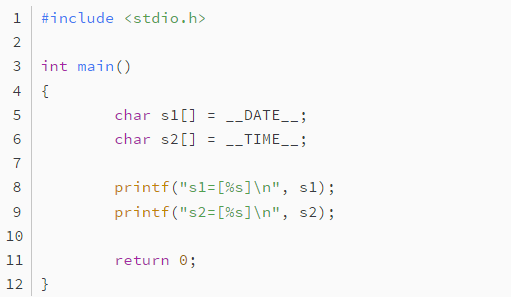
\_\_FILE\_\_：在源文件中插入当前源文件名；是一个字符串

\_\_DATE\_\_：在源文件中插入当前的编译日期；格式是Mmm:dd:yyyy 是字符串

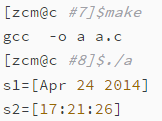
\_\_TIME\_\_：在源文件中插入当前编译时间；格式是hh:mm:ss 是字符串

\_\_STDC\_\_：当要求程序严格遵循ANSI C标准时该标识被赋值为1；

\_\_cplusplus：当编写C++程序时该标识符被定义。



运行结果：



19 . #操作符和##操作符

1、#操作符

#操作符用于预处理阶段，将宏参数转换为字符串，只有宏定义中使用(#define)

使用方法：

#define  STRING(x)   #x

printf("%s\n",STRING(Hello World!));

2、##操作符

##操作符用于预处理阶段，将粘连两个标识符，只有宏定义中使用(#define)

使用方法：

#define  CONNECT(a,b)   a##b

int  CONNECT(a,1);    //int a1

a1 = 2;

20 . 负数的移位运算

* 负数的 >> 运算

（1）逻辑推理

0 >> 1 = 0

-1 >> 1 = -1

-2 >> 1 = -1

-3 >> 1 = -2

-4 >> 1 = -2

可以推断出：

**- 奇数 >> 1 = -奇数 / 2 - 1**

**- 偶数 >> 1 = -偶数 / 2**

（2）实际上的原理：

负数的移位运算，本质上是负数补码的移位运算

char num = -3;

原码：1 000,0011

补码：1 111,1101

移位运算后，补码：1 111，1110

取反码得：1 000，0001

取原码得：1 000，0010

即：-3 >> 1 = -2

**补码 --> 原码：取反再-1 或者 -1再取反**

**原码 --> 补码：取反再+1**

* 负数的 << 运算

char num = -1;

num << 1 = -2;

直接 \*2 即可；