**1.** 递归的基本思想是把规模大的问题转化为规模小的相似的子问题来解决。在函数实现时，因为解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数调用它自身的情况。另外这个解决问题的函数必须有明显的结束条件，这样就不会产生无限递归的情况了。

**2.** 类似这样的语句，不是初始化，而是重新赋值！

C\C++语法是禁止结构体这样赋值的！！

cities[0]={"A",10000,11};

cities[1]={"B",20000,11};

**3.** strlen()用来计算指定的字符串s 的长度，不包括结束字符"\0"。

**4.** \*p++ 和 （\*p）++的之间的差别：

\*p++是 地址会变化。 口诀：取当前值，然后再移动地址！

（\*p）++ 是数值会要变化。 口诀：取当前值，然后再使数值增加1。

**5. 二级指针：**

\*p：一级指针：存放变量的地址。

\*\*q：二级指针：存放一级指针的地址。

常考题目： int x=7；

int\*p=&x，\*\*q=p；

问你：\*p为多少？\*q为多少？\*\*q为多少？

7 p 7

再问你：\*\*q=&x的写法可以吗？

不可以，因为二级指针只能存放一级指针的地址。

**6.** sizeof 计算字符串长度时候会把最后的 '\0' 算进去，而strlen 不会。

strlen，是判断字符串的长度，即：string length，在这里是good的长度，长 度是4.  
 sizeof，是判断类型的长度，在这里是数组的长度，在这里是str数据的长度， 长度是10。

**7.** 127.0.0.1被称为[本地回环地址](http://baike.baidu.com/item/%E6%9C%AC%E5%9C%B0%E5%9B%9E%E7%8E%AF%E5%9C%B0%E5%9D%80" \t "_blank)，主要作用有两个：

一是测试本机的网络配置，能PING通127.0.0.1说明本机的网卡和IP协议安装都没有问题；

另一个作用是某些SERVER/CLIENT的应用程序在运行时需调用服务器上的资源，一般要指定SERVER的IP地址，但当该程序要在同一台机器上运行而没有别的SERVER时就可以把SERVER的资源装在本机，SERVER的IP地址设为127.0.0.1也同样可以运行。

**8.** #pragma pack（n）按n字节对齐的方式补齐，嵌入式用的多，节省空间

\*对齐：就是每个成员必须放在自己大小整数倍的起始位置，大于4字节的按4 字节算

\*补齐：就是每个结构体的大小应该是最大的成员的整数倍，大于4字节的按4 字节算

**9.每一层做哪些规定是都有协议详细规定的**

**应用层常见协议：HTTP FTP POP3 SMTP**

**传输层常见协议：TCP：基于连接的传输控制协议**

**UDP：数据报协议**

**网络层常见协议：IP ARP（地址解析协议） RARP（方向地址解析协议）**

**数据链路层常见协议：PPP（点对点协议）**

**物理层常见协议：RJ-45**

**10.**

**代码段：**代码 const全局变量、字符串字面常量

**数据段：**初始化不为0的全局变量和静态局部变量(static)

**BSS：**未初始化的全局变量、初始化为0的全局变量

**堆区：**malloc

**栈区：**局部变量、参数、返回值

**11.** 在C语言中：  
 十进制直接表示,例如：1,2,3,4,5,6  
 八进制要加零表示,例如：00,01,02,03,04,05  
 十六进制要加0x表示,例如：0xA1,0x13,0xFF

**12.**sizeof既是关键字，又是操作符，但不是函数

**13.** C语言跟内存申请相关的函数主要有 alloca、calloc、malloc、free、realloc

<1>alloca是向栈申请内存,因此无需释放.

<2>malloc分配的内存是位于堆中的,并且没有初始化内存的内容,因此基 本上malloc之后,调用函数memset来初始化这部分的内存空间.

<3>calloc则将初始化这部分的内存,设置为0.

<4>realloc则对malloc申请的内存进行大小的调整.

<5>申请的内存最终需要通过函数free来释放.

当程序运行过程中malloc了,但是没有free的话,会造成内存泄漏.一部分的内存没有被使用,但是由于没有free,因此系统认为这部分内存还在使用,造成不断的向系统申请内存,使得系统可用内存不断减少.但是内存泄漏仅仅指程序在运行时,程序退出时,OS将回收所有的资源.因此,适当的重起一下程序,有时候还是有点作用.

**14. memset()的作用**

1.void \*memset(void \*s,int c,size\_t n)

总的作用：将已开辟内存空间 s 的首 n 个字节的值设为值 c。

2.memset() 函数常用于内存空间初始化。如：

char str[100];

memset(str,0,100);

3.memset可以方便的清空一个结构类型的变量或数组。

如：  
struct sample\_struct  
{  
char csName[16];  
int iSeq;  
int iType;  
};  
  
对于变量  
struct sample\_strcut stTest;  
  
一般情况下，清空stTest的方法：  
stTest.csName[0]='/0';  
stTest.iSeq=0;  
stTest.iType=0;  
  
用memset就非常方便：  
memset(&stTest,0,sizeof(struct sample\_struct));  
  
如果是结构体数组：  
struct sample\_struct TEST[10];  
则  
memset(TEST,0,sizeof(struct sample\_struct)\*10);

**15.static的作用**

**修饰局部变量：**该局部变量的生命周期为整个程序执行期间

作用域为函数内部

**修饰全局变量：**该全局变量的生命周期为整个程序执行期间（这个和全局 变量一样），但是它的作用域为模块（xxx.c）内部

比如说一个工程里，A写了一个全局变量int test=111；

B又写了一个全局变量，int test=123；把两个文 件一起编译时就会出错，因为重复定义了。那么这 个时候就应该在定义全局变量前加上static，这样 它们的作用域就只是包含它们的模块了

**修饰函数：**作用等同于修饰全局变量

比如说两个.c文件里都有一个名字一样的函数，只要有 一个加了static就可以运行。那么那个加上static 的作用域就是包含它的那个模块，另外那个没加的 作用域还是原来的。

**16. 关于字符串操作**

（1）字面值：”abc”

（2）字符指针：char \*str

（3）字符数组： char str[]

char \*str1=”abc”;//”abc”在代码段里，str1在栈里

char str2[]=”abc”;//”abc”有两份，一份在代码段，一份是放在字符数组里 的备份（栈里），str2占据的4个字节也在栈里

**17.sizeof与strlen**

sizeof是取字节运算符（关键字），strlen是函数。

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'，如果你只定义没有给它赋初值，这个结果是不定的，它会从aa首地址一直找下去，直到遇到'\0'停止。

请看第一个ASCII码，对是0，对应的字符是（Null），其实就是 ‘\0’，即空字符。判断一个字符串是否结束的标志就是看是否遇到‘\0’，如果遇到‘\0’，则表示字符串结束。而字符‘0’对应的ASCII码是 48

Sizeof与Strlen的区别与联系(转）

1.sizeof操作符的结果类型是size\_t，它在头文件中typedef为unsigned int类型。

该类型保证能容纳实现所建立的最大对象的字节大小。

2.sizeof是算符，strlen是函数。

3.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，比如：

short f();

printf("%d\n", sizeof(f()));

输出的结果是sizeof(short)，即2。

4.数组做sizeof的参数不退化，传递给strlen就退化为指针了。

5.大部分编译程序 在编译的时候就把sizeof计算过了 是类型或是变量的长度这就是sizeof(x)可以用来定义数组维数的原因

char str[20]="0123456789";

int a=strlen(str); //a=10;

int b=sizeof(str); //而b=20;

6.strlen的结果要在运行的时候才能计算出来，时用来计算字符串的长度，不是类型占内存的大小。

7.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。

8.当适用了于一个结构类型时或变量， sizeof 返回实际的大小，

当适用一静态地空间数组， sizeof 归还全部数组的尺寸。

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

9.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，

如：

fun(char [8])

fun(char [])

都等价于 fun(char \*)

在C++里参数传递数组永远都是传递指向数组首元素的指针，编译器不知道数组的大小

如果想在函数内知道数组的大小， 需要这样做：

进入函数后用memcpy拷贝出来，长度由另一个形参传进去

fun(unsiged char \*p1, int len)

{

unsigned char\* buf = new unsigned char[len+1]

memcpy(buf, p1, len);

}

我们能常在用到 sizeof 和 strlen 的时候，通常是计算字符串数组的长度

看了上面的详细解释，发现两者的使用还是有区别的，从这个例子可以看得很清楚：

char str[20]="0123456789";

int a=strlen(str); //a=10; >>>> strlen 计算字符串的长度，以结束符 0x00 为字符串结束。

int b=sizeof(str); //而b=20; >>>> sizeof 计算的则是分配的数组 str[20] 所占的内存空间的大小，不受里面存储的内容改变。

上面是对静态数组处理的结果，如果是对指针，结果就不一样了

char\* ss = "0123456789";

sizeof(ss) 结果 4 ＝＝＝》ss是指向字符串常量的字符指针，sizeof 获得的是一个指针的之所占的空间,应该是

长整型的，所以是4

sizeof(\*ss) 结果 1 ＝＝＝》\*ss是第一个字符 其实就是获得了字符串的第一位'0' 所占的内存空间，是char类

型的，占了 1 位

strlen(ss)= 10 >>>> 如果要获得这个字符串的长度，则一定要使用 strlen

**18.**常见基本类型的字节大小

32位操作系统

char ：1个字节(固定)

\*(即指针变量): 4个字节(32位机的寻址空间是4个字节。同理64位编译器)(变化\*)

short int : 2个字节(固定)

int： 4个字节(固定)

unsigned int : 4个字节(固定)

float: 4个字节(固定)

double: 8个字节(固定)

long: 4个字节

unsigned long: 4个字节(变化\*,其实就是寻址控件的地址长度数值)

long long: 8个字节(固定)

64位操作系统

char ：1个字节(固定)

\*(即指针变量): 8个字节

short int : 2个字节(固定)

int： 4个字节(固定)

unsigned int : 4个字节(固定)

float: 4个字节(固定)

double: 8个字节(固定)

long: 8个字节

unsigned long: 8个字节(变化\*其实就是寻址控件的地址长度数值)

long long: 8个字节(固定)

除了\*与long随操作系统子长变化而变化外，其他的都固定不变(32位和64相比)

bool 1个字节 char 1个字节 int 4个字节 float 4个字节 doubl 8个字节 long long 8个字节

**19.**

char \*str="abc";

char \*p=str; //左值是地址，右值是内容，一级指针存的是地址，但不能是指针变量的地址；

char \*\*p=&str; //二级指针存的是一级指针的地址

**20.%d相关**

**%3d** **输出3位宽度整数，不足前面补空格**

**%-3d 输出3位宽度整数，不足后面补空格**

**%03d 输出3位宽度整数，不足前面补0**

**%08x**为整型以16进制方式输出的格式字符串，会把后续对应参数的整型数字，以16进制输出。08的含义为，输出的16进制值占8位，不足部分左侧补0。

于是，如果执行

printf("0x%08x", 0x1234);

会输出0x00001234。

**21.逗号表达式（**表达式1，表达式2**）**

逗号表达式返回的是表达式2的值，如果表达式2中有表达式1中的变量，则先计算表达式1中变量的值，再带进表达式2中求出表达式2的值并返回。

**22.sprintf()函数**

函数功能：把格式化的数据写入某个字符串  
函数原型：int sprintf( char \*buffer, const char \*format [, argument] … );  
返回值：字符串长度（strlen）  
例子：  
char\* who = "I";  
char\* whom = "CSDN";  
sprintf(s, "%s love %s.", who, whom); //产生："I love CSDN. " 这字符串写到s中  
sprintf(s, "%10.3f", 3.1415626); //产生：" 3.142"

**23.关于数组首地址赋值给指针和数组指针的问题**  
int a[5] = {1,2,3,4,5};  
int \*p = a;  
int (\*pa)[5] = &a;  
由于a是一个有5个元素的数组，a表示数组起始地址，&a也表示取数组起始地址，我打印a和&a的值是一样的。  
p是一个指针，指向数组a;而Pa是一个指针，指向有5个int型元素的数组，那么P和Pa同样是一个指针，而a和&a同样是取地址值，为什么两者初始化时候不能交叉呢，也就是说写成： int \*p = &a;  
int (\*pa)[5] = a;  
这两种写法都有误的，请问a和&a的区别在哪里呢？和\*p、(\*pa)[5]在一起这样用怎么解释呢？谢谢！

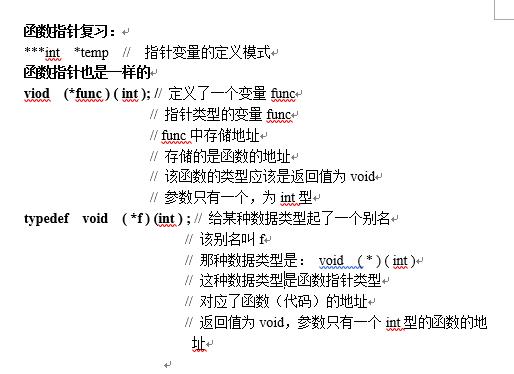
**int (\*pa)[5] ， 定义的二维数组指针，相当于 int pa[ ][5];   
 而a是一维数组，所以，类型不一会报告出错；**

**从编译器的角度来讲，int \*p定义了一个指向int型的指针p。而a本身已经是指向int型的指针了，对a做取值操作，返回的结果是int \*\*类型的，造成类型不匹配，所以不能这样写：int \*p = &a;  
而int (\*pa)[5]定义了一个指向5个元素的int数组的指针，而a只是数组的首地址，&a才是数组的指针，同样也是因为类型不匹配。**

**24. c语言往.txt文件中写入的内容是乱码**

fwrite用于二进制写入，怎样写对于数据来说都是乱码，而对于字符则能正常输出，而对于结构体，最好使用格式输出函数fprintf（）与fscanf（），这样可以保证输出的正确性，而fwrite函数时将内存中的二进制数据直接写入文本文件，像11001010这样的，所以文本文档当然会显示乱码了 。

**25.函数指针**



**25.typedef常见用法**

1.常规变量类型定义

例如：typedef unsigned char uchar

描述：uchar等价于unsigned char类型定义 uchar c声明等于unsigned char c声明

2.数组类型定义

例如： typedef int array[2];

描述： array等价于 int [2]定义; array a声明等价于int a[2]声明

扩展： typedef int array[M][N];

描述： array等价于 int [M][N]定义; array a声明等价于int a[M][N]声明

3.指针类型定义

例如： typedef int \*pointer;

描述： pointer等价于 int \*定义;pointer p声明等价于int \*a声明

例如： typedef int \*pointer[M];

描述： pointer等价于 int \*[M]定义 pointer p声明等价于int \*a[M]声明明

4.函数地址说明

描述：C把函数名字当做函数的首地址来对待，我们可以使用最简单的方法得到函数地址

例如： 函数:int func(void); unsigned long funcAddr=(unsigned long)func， funcAddr的值是func函数的首地址

5.函数声明

例如： typedef int func(void); func等价于 int (void)类型函数

描述1： func f声明等价于 int f(void)声明，用于文件的函数声明

描述2： func \*pf声明等价于 int (\*pf)(void)声明，用于函数指针的生命，见下一条

6.函数指针

例如： typedef int (\*func)(void)

描述： func等价于int (\*)(void)类型

func pf等价于int (\*pf)(void)声明，pf是一个函数指针变量

7.识别typedef的方法：

a).第一步。使用已知的类型定义替代typdef后面的名称,直到只剩下一个名字不识别为正确

如typedef u32 (\*func)(u8);

从上面的定义中找到 typedef \_\_u32 u32;typedef \_\_u8 u8

继续找到 typedef unsigned int \_\_u32;typedef unsigned char \_\_u8;

替代位置名称 typedef unsigned int (\*func)(void);

现在只有func属于未知。

b).第二步.未知名字为定义类型，类型为取出名称和typedef的所有部分，如上为

func等价于unsigned unsigned int (\*)(unsigned char);

c).第三部.定义一个变量时，变量类型等价于把变量替代未知名字的位置所得到的类型

func f等价于unsigned unsigned int (\*f)(unsigned char)

# 26.const修饰的变量如果在编译阶段被修改就会报错，如果在程序执行阶段被修改是可以的

**（如数组的越界访问对const变量的修改）**

int main

{

char buf[4];

const int a = 0;

buf[4] = 97;

printf(“the a is %d\n”,a);

}

其中最后一句printf的目的是看下变量a的值是否改变，根据const的理解，如果const修饰的是变量是不能被修改的话，那么a的值一定不会改变，肯定还是0。但是在实际运行的结果中，我们发现a的值已经变为97了。这说明const修饰的变量a，已经被我们程序修改了。

那综合这两个例子，我们来分析下，对于第二例子，修改的原因是buf[4]的赋值操作，我们知道buf[4]这个变量已经造成了buf这个数组变量的越界访问。buf数组的成员本身只有0,1,2,3，那么buf[4]访问的是谁那，根据局部变量的地址分配，可以知道buf[4]的地址和int a的地址是一样，那么buf[4]实际上就是访问了const int a；那么对buf[4]的修改，自然也修改了const int a的空间，这也是为什么我们在最后打印a的值的时候看到了97这个结果。

那么我们现在可以知道了，const修饰的变量是不具备不允许修改的特性的，那么对于第一个例子的现象我们又如何解释那。

第一个例子，错误是在程序编译的时候给出的，注意这里，这个时候并没有生成可执行文件，说明const修饰的变量可否修改是由编译器来帮我们保护了。而第二个例子里，变量的修改是在可执行程序执行的时候修改的，说明a还是一个变量。

综上所述，我们可以得出一个结论，那就是const修饰的变量，其实质是告诉程序员或编译器该变量为只读，如果程序员在程序中显示的修改一个只读变量，编译器会毫不留情的给出一个error。而对于由于像数组溢出，隐式修改等程序不规范书写造成的运行过程中的修改，编译器是无能为力的，也说明const修饰的变量仍然是具备变量属性的。

# 27. 函数原型char\* asctime (const struct tm \* timeptr).

功能：

把timeptr指向的tm结构体中储存的时间转换为字符串，返回的字符串格式为：Www Mmm dd hh:mm:ss yyyy。其中Www为星期；Mmm为月份；dd为日；hh为时；mm为分；ss为秒；yyyy为年份。

