## 基础篇

### 1.程序的局部变量存在于哪里，全局变量存在于哪里，动态申请数据存在于哪里。

答：程序的局部变量存在于栈区；全局变量存在于静态区；动态申请数据存在于堆区。

### 2.关键字const有什么含义？

答：1)只读。

2）使用关键字const也许能产生更紧凑的代码。

3）使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。

4)作用

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针（pointer to const）和自身是常量的指针（常量指针，const pointer）；
3. 修饰引用，指向常量的引用（reference to const），用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

5) [const 的指针与引用](https://interview.huihut.com/#/?id=const-%e7%9a%84%e6%8c%87%e9%92%88%e4%b8%8e%e5%bc%95%e7%94%a8)

* 指针
  + 指向常量的指针（pointer to const）
  + 自身是常量的指针（常量指针，const pointer）
* 引用
  + 指向常量的引用（reference to const）
  + 没有 const reference，因为引用本身就是 const pointer

### 3.关键字static作用

在C语言中，关键字static有三个明显的作用：  
1) 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。  
2) 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。  
3) 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

### 4.位操作bit manipulation

嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量a，写两段代码，第一个设置a的bit 3，第二个清除a 的bit 3。在以上两个操作中，要保持其它位不变。  
      对这个问题有三种基本的反应  
1) 不知道如何下手。该被面者从没做过任何嵌入式系统的工作。  
2) 用bit fields。Bit fields是被扔到C语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了的你的代码是不可重用的。我最近不幸看到 Infineon为其较复杂的通信芯片写的驱动程序，它用到了bit fields因此完全对我无用，因为我的编译器用其它的方式来实现bit fields的。从道德讲：永远不要让一个非嵌入式的家伙粘实际硬件的边。  
3) 用 #defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。最佳的解决方案如下：  
  
#define BIT3 (0x1 << 3)  
static int a;  
  
void set\_bit3(void)  
{  
    a |= BIT3;  
}  
void clear\_bit3(void)  
{  
    a &= ~BIT3;  
}  
  
      一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数，这也是可以接受的。我希望看到几个要点：说明常数、|=和&=~操作。

### 5. 访问固定的内存位置（Accessing fixed memory locations）

嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为0x67a9的整型变量的值为0xaa66。编译器是一个纯粹的ANSI编译器。写代码去完成这一任务。  
      这一问题测试你是否知道为了访问一绝对地址把一个整型数强制转换（typecast）为一指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下：  
    int \*ptr;  
    ptr = (int \*)0x67a9;  
    \*ptr = 0xaa55;  
  
  A more obscure approach is:  
一个较晦涩的方法是：  
  
    \*(int \* const)(0x67a9) = 0xaa55;  
  
即使你的品味更接近第二种方案，但我建议你在面试时使用第一种方案。

### 6.中断interrupts

中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展—让标准C支持中断。具代表事实是，产生了一个新的关键字 \_\_interrupt。下面的代码就使用了\_\_interrupt关键字去定义了一个中断服务子程序(ISR)，请评论一下这段代码的。  
  
\_\_interrupt double compute\_area (double radius)  
{  
    double area = PI \* radius \* radius;  
    printf("\nArea = %f", area);  
    return area;  
}  
  
这个函数有太多的错误了，以至让人不知从何说起了：  
1) ISR 不能返回一个值。如果你不懂这个，那么你不会被雇用的。  
2) ISR 不能传递参数。如果你没有看到这一点，你被雇用的机会等同第一项。  
3) 在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈，有些处理器/编译器就是不允许在ISR中做浮点运算。此外，ISR应该是短而有效率的，在ISR中做浮点运算是不明智的。  
4) 与第三点一脉相承，printf()经常有重入和性能上的问题。如果你丢掉了第三和第四点，我不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。

### 7.可重入函数与不可重入函数

在实时系统的设计中，经常会出现多个任务调用同一个函数的情况。如果有一个函数不幸被设计成为这样：那么不同任务调用这个函数时可能修改其他任务调用这个函数的数据，从而导致不可预料的后果。**这样的函数是不安全的函数，也叫不可重入函数**。

肯定有一个安全的函数，这个安全的函数又叫可重入函数。那么什么是可重入函数呢？所谓可重入是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。

一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误；而不可重入的函数由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。

**也可以这样理解，重入即表示重复进入**，首先它意味着这个函数可以被中断，其次意味着它除了使用自己栈上的变量以外不依赖于任何环境（包括 static），这样的函数就是purecode（纯代码）可重入，可以允许有该函数的多个副本在运行，由于它们使用的是分离的栈，所以不会互相干扰。如果确实需要访问全局变量（包括 static），一定要注意实施互斥手段。可重入函数在并行运行环境中非常重要，但是一般要为访问全局变量付出一些性能代价。  
编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。  
说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

示例：假设 Exam 是 int 型全局变量，函数 Squre\_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

1. int Exam = 0;
2. unsigned int example( int para )
3. {
4. unsigned int temp;
5. Exam = para; // （\*\*）
6. temp = Square\_Exam( );
7. return temp;
8. }

此函数若被多个进程调用的话，其结果可能是未知的，因为当（\*\*）语句刚执行完后，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使 Exam 赋与另一个不同的 para 值，所以当控制重新回到 “temp = Square\_Exam( )” 后，计算出的temp很可能不是预想中的结果。此函数应如下改进。

1. int Exam = 0;
2. unsigned int example( int para )
3. {
4. unsigned int temp;
5. [申请信号量操作] //(1)  加锁
6. Exam = para;
7. temp = Square\_Exam( );
8. [释放信号量操作] //     解锁
9. return temp;
10. }

申请不到“信号量”，说明另外的进程正处于给 Exam 赋值并计算其平方过程中（即正在使用此信号），本进程必须等待其释放信号后，才可继续执行。若申请到信号，则可继续执行，但其它进程必须等待本进程释放信号量后，才能再使用本信号。

保证函数的可重入性的方法：

1）在写函数时候尽量使用局部变量（例如寄存器、堆栈中的变量）；

2）对于要使用的全局变量要加以保护（如采取关中断、信号量等互斥方法），这样构成的函数就一定是一个可重入的函数。

满足下列条件的函数多数是不可重入（不安全）的：

1）函数体内使用了静态的数据结构；

2）函数体内调用了malloc() 或者 free() 函数；

3）函数体内调用了标准 I/O 函数。

如何将一个不可重入的函数改写成可重入函数呢？把一个不可重入函数变成可重入的唯一方法是用可重入规则来重写它。其实很简单，只要遵守了几条很容易理解的规则，那么写出来的函数就是可重入的：

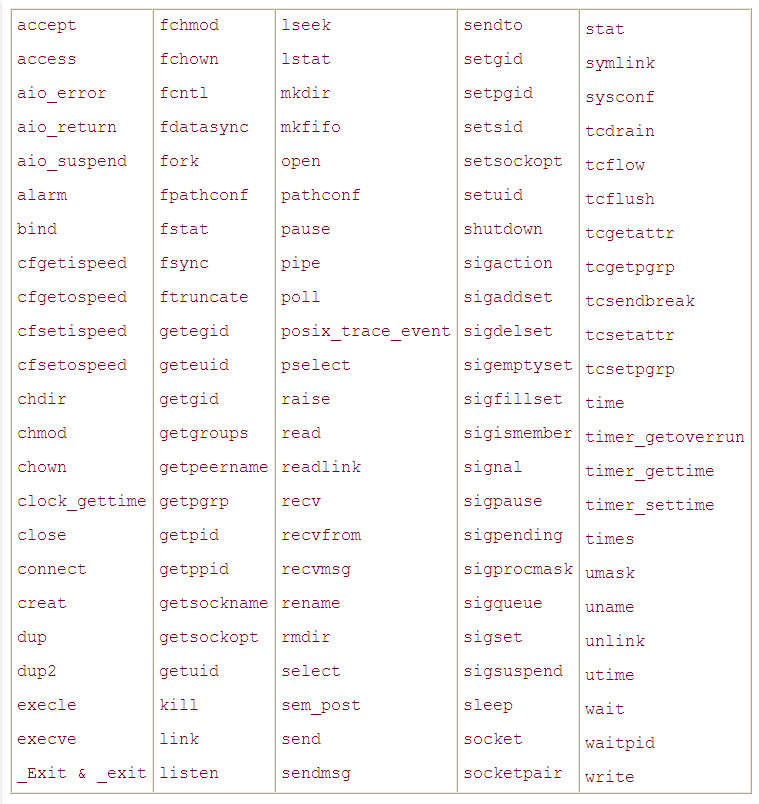
1）不要使用全局变量。因为别的代码很可能改变这些变量值。

2）在和硬件发生交互的时候，切记执行类似 disinterrupt() 之类的操作，就是关闭硬件中断。完成交互记得打开中断，在有些系列上，这叫做“进入/ 退出核心”。

3）不能调用其它任何不可重入的函数。

4）谨慎使用堆栈。

Linux常见的可重入函数



### 8.有符号和无符号整数转换问题

下面的代码输出是什么，为什么？  
  
void foo(void)  
{  
    unsigned int a = 6;  
    int b = -20;  
    (a+b > 6) ? puts("> 6") : puts("<= 6");  
}  
      这个问题测试你是否懂得C语言中的整数自动转换原则，我发现有些开发者懂得极少这些东西。不管如何，这无符号整型问题的答案是输出是 ">6"。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。因此-20变成了一个非常大的正整数，所以该表达式计算出的结果大于6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是丰常重要的。如果你答错了这个问题，你也就到了得不到这份工作的边缘。

### 9.处理器字长相关问题

评价下面的代码片断：  
  
unsigned int zero = 0;  
unsigned int compzero = 0xFFFF;  
/\*1''s complement of zero \*/  
  
对于一个int型不是16位的处理器为说，上面的代码是不正确的。应编写如下：  
  
unsigned int compzero = ~0;  
  
      这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里，好的嵌入式程序员非常准确地明白硬件的细节和它的局限，然而PC机程序往往把硬件作为一个无法避免的烦恼。  
      到了这个阶段，应试者或者完全垂头丧气了或者信心满满志在必得。如果显然应试者不是很好，那么这个测试就在这里结束了。但如果显然应试者做得不错，那么我就扔出下面的追加问题，这些问题是比较难的，我想仅仅非常优秀的应试者能做得不错。提出这些问题，我希望更多看到应试者应付问题的方法，而不是答案。不管如何，你就当是这个娱乐吧...

### 10.动态内存分配（Dynamic memory allocation）

 尽管不像非嵌入式计算机那么常见，嵌入式系统还是有从堆（heap）中动态分配内存的过程的。那么嵌入式系统中，动态分配内存可能发生的问题是什么？  
      这里，我期望应试者能提到内存碎片，碎片收集的问题，变量的持行时间等等。这个主题已经在ESP杂志中被广泛地讨论过了（主要是 P.J. Plauger, 他的解释远远超过我这里能提到的任何解释），所有回过头看一下这些杂志吧！让应试者进入一种虚假的安全感觉后，我拿出这么一个小节目：  
      下面的代码片段的输出是什么，为什么？  
  
char \*ptr;  
if ((ptr = (char \*)malloc(0)) == NULL)  
    puts("Got a null pointer");  
else  
    puts("Got a valid pointer");  
  
      这是一个有趣的问题。最近在我的一个同事不经意把0值传给了函数malloc，得到了一个合法的指针之后，我才想到这个问题。这就是上面的代码，该代码的输出是"Got a valid pointer"。我用这个来开始讨论这样的一问题，看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要，但解决问题的方法和你做决定的基本原理更重要些。

### 11.Typedef

 Typedef 在C语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如，思考一下下面的例子：  
  
#define dPS struct s \*  
typedef struct s \* tPS;  
  
      以上两种情况的意图都是要定义dPS 和 tPS 作为一个指向结构s指针。哪种方法更好呢？（如果有的话）为什么？  
这是一个非常微妙的问题，任何人答对这个问题（正当的原因）是应当被恭喜的。答案是：typedef更好。思考下面的例子：  
  
dPS p1,p2;  
tPS p3,p4;  
  
第一个扩展为  
  
struct s \* p1, p2;  
.  
上面的代码定义p1为一个指向结构的指，p2为一个实际的结构，这也许不是你想要的。第二个例子正确地定义了p3 和p4 两个指针。

### 12.晦涩的语法

C语言同意一些令人震惊的结构,下面的结构是合法的吗，如果是它做些什么？  
  
int a = 5, b = 7, c;  
c = a+++b;  
  
      这个问题将做为这个测验的一个愉快的结尾。不管你相不相信，上面的例子是完全合乎语法的。问题是编译器如何处理它？水平不高的编译作者实际上会争论这个问题，根据最处理原则，编译器应当能处理尽可能所有合法的用法。因此，上面的代码被处理成：  
  
c = a++ + b;  
  
      因此, 这段代码持行后a = 6, b = 7, c = 12。  
      如果你知道答案，或猜出正确答案，做得好。如果你不知道答案，我也不把这个当作问题。我发现这个问题的最大好处是这是一个关于代码编写风格，代码的可读性，代码的可修改性的好的话题

### 13.预处理器

1 . 用预处理指令#define 声明一个常数，用以表明1年中有多少秒（忽略闰年问题）  
      #define SECONDS\_PER\_YEAR (60 \* 60 \* 24 \* 365)UL  
我在这想看到几件事情：  
1) #define 语法的基本知识（例如：不能以分号结束，括号的使用，等等）  
2) 懂得预处理器将为你计算常数表达式的值，因此直接写出你如何计算一年中有多少秒而不是计算出实际的值，是更清晰而没有代价的。  
3) 意识到这个表达式将使一个16位机的整型数溢出-因此要用到长整型符号L,告诉编译器这个常数是的长整型数。  
4) 如果你在你的表达式中用到UL（表示无符号长整型），那么你有了一个好的起点。记住，第一印象很重要。

### 14.sizeof()和初不初始化，没有关系；strlen()和初始化有关

运行的结果为什么等于15或异常

#include "stdio.h"

#include "string.h"

void main()

{

char aa[10];

printf("%d",strlen(aa));

}

答案：sizeof()和初不初始化，没有关系；strlen()和初始化有关。

### 15.如何处理算术运算溢出问题

bool add (int a, int b,int \*c)

{

\*c=a+b;

**return (a>0 && b>0 &&(\*c<a || \*c<b) || (a<0 && b<0 &&(\*c>a || \*c>b)))**;

}

true为溢出，false为正常

## 代码分析篇

### 1.实例一：补码运算

struct bit

{    int a:3;

     int b:2;

     int c:3;

};

int main()

{

   bit s;

   char \*c=(char\*)&s;

    cout<<sizeof(bit)<<endl;

   \*c=0x99;

    cout << s.a <<endl <<s.b<<endl<<s.c<<endl;

      int a=-1;

    printf("%x",a);

   return 0;

}

输出为什么是？

答案：4

1

-1

-4

ffffffff

因为0x99在内存中表示为 100 11 001 , a = 001, b = 11, c = 100（在vc环境中，一般是由右到左进行分配的）

当c为有符合数时, c = 100, 最高1为表示c为负数，负数在计算机用补码表示，所以c = -4;同理

b = -1;

当c为有符合数时, c = 100,即 c = 4,同理 b = 3

* 正数补码即其二进制位
* 负数补码，转换为二进制位后，符号位不变，其余位取反，然后整体+1

### 2.实例二：数组越界+死循环

下面这个程序执行后会有什么错误或者效果:

#define MAX 255

int main()

{

    unsigned char A[MAX],**i**; //i被定义为unsigned char

    for (i=0;i<=MAX;i++)

       A[i]=i;

}

答案：**死循环加数组越界访问（C/C++不进行数组越界检查）**

MAX=255

数组A的下标范围为:0..MAX-1,这是其一..

其二.当i循环到255时,循环内执行:

   A[255]=255;

这句本身没有问题..但是返回for (i=0;i<=MAX;i++)语句时,

**由于unsigned char的取值范围在(0..255),i++以后i又为0了..无限循环下去**.

## 算法篇

### 1.两路归并排序

**线性表a、b为两个有序升序的线性表，编写一程序，使两个有序线性表合并成一个有序升序线性表h**

|  |
| --- |
| **Linklist \*unio(Linklist \*p,Linklist \*q){**  **linklist \*R,\*pa,\*qa,\*ra;**  **pa=p;**  **qa=q;**  **R=ra=p;**  **while(pa->next!=NULL&&qa->next!=NULL){**  **if(pa->data>qa->data){**  **ra->next=qa;**  **qa=qa->next;**  **}**  **else{**  **ra->next=pa;**  **pa=pa->next;**  **}**  **}**  **if(pa->next!=NULL)**  **ra->next=pa;**  **if(qa->next!=NULL)**  **ra->next==qa;**  **return R;**  **}** |

### 2.用递归算法判断数组a[N]是否为一个递增数组。

**递归的方法，记录当前最大的，并且判断当前的是否比这个还大，大则继续，否则返回false结束：**

**bool fun( int a[], int n )**

**{**

**if( n= =1 )**

**return true;**

**if( n= =2 )**

**return a[n-1] >= a[n-2];**

**return fun( a,n-1) && ( a[n-1] >= a[n-2] );**

**}**

### 3.单连表的建立，把'a'--'z'26个字母插入到连表中，并且倒叙，还要打印！

**方法1：**

**typedef struct val**

**{    int date\_1;**

**struct val \*next;**

**}\*p;**

**void main(void)**

**{    char c;**

**for(c=122;c>=97;c--)**

**{ p.date=c;**

**p=p->next;**

**}**

**p.next=NULL;**

**}**

**}**

方法2：

**node \*p = NULL;**

**node \*q = NULL;**

**node \*head = (node\*)malloc(sizeof(node));**

**head->data = ' ';head->next=NULL;**

**node \*first = (node\*)malloc(sizeof(node));**

**first->data = 'a';first->next=NULL;head->next = first;**

**p = first;**

**int longth = 'z' - 'b';**

**int i=0;**

**while ( i<=longth )**

**{**

**node \*temp = (node\*)malloc(sizeof(node));**

**temp->data = 'b'+i;temp->next=NULL;q=temp;**

**head->next = temp; temp->next=p;p=q;**

**i++;**

**}**

**print(head);**

### 4.一个递归反向输出字符串的例子,可谓是反序的经典例程.

void inverse(char \*p)

{

     if( \*p = = '\0' )

return;

     inverse( p+1 );

     printf( "%c", \*p );

}

int main(int argc, char \*argv[])

{

     inverse("abc\0");

     return 0;

}

### 5.