5. 用变量a给出下面的定义

a) 一个整型数（An integer） **a) int a; // An integer**

b) 一个指向整型数的指针（A pointer to an integer）  **int \*a; // A pointer to an integer**

c) 一个指向指针的的指针，它指向的指针是指向一个整型数（A pointer to a pointer to an integer） **int \*\*a; // A pointer to a pointer to an integer**

d) 一个有10个整型数的数组（An array of 10 integers） **int a[10]; // An array of 10 integers**

e) 一个有10个指针的数组，该指针是指向一个整型数的（An array of 10 pointers to integers） **int \*a[10]; // An array of 10 pointers to integers**

f) 一个指向有10个整型数数组的指针（A pointer to an array of 10 integers） **int (\*a)[10];**

g) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer） **int (\*a)(int);**

h) 一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数int (\*a[10])(int);

**6. 关键字static的作用是什么？**

这个简单的问题很少有人能回答完全。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。

大多数应试者能正确回答第一部分，一部分能正确回答第二部分，同是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数据和代码范围的好处和重要性。

**7．关键字const是什么含意？**

我只要一听到被面试者说：“const意味着常数”，我就知道我正在和一个业余者打交道。去年Dan Saks已经在他的文章里完全概括了const的所有用法，因此ESP(译者：Embedded Systems Programming)的每一位读者应该非常熟悉const能做什么和不能做什么.

如果你从没有读到那篇文章，只要能说出const意味着“只读”就可以了。尽管这个答案不是完全的答案，但我接受它作为一个正确的答案。（如果你想知道更详细的答案，仔细读一下Saks的文章吧。）如果应试者能正确回答这个问题，我将问他一个附加的问题：下面的声明都是什么意思？

|  |
| --- |
| Const int a;  int const a;  const int \*a;  int \* const a;  int const \* a const; |

前两个的作用是一样**，a是一个常整型数**。第三个意味着a是一个**指向常整型数的指针**（也就是，**整型数是不可修改的**，但**指针可以）**。第四个意思a是一个指向整型数的**常指针**（也就是说，指针指向的**整型数是可以修改的**，但指针**是不可修改的**）。最后一个意味着a是一个指向常整型数的常指针（也就是说，指针指向的**整型数是不可修改的，同时指针也是不可修改的**）。如果应试者能正确回答这些问题，那么他就给我留下了一个好印象。顺带提一句，也许你可能会问，即使不用关键字const，也还是能很容易写出功能正确的程序，那么我为什么还要如此看重关键字const呢？我也如下的几下理由：

1). **关键字const的作用是为给读你代码的人传达非常有用的信息**，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。（当然，懂得用const的程序员很少会留下的垃圾让别人来清理的。）

2). **通过给优化器一些附加的信息，使用关键字const**也许能产生更紧凑的代码。

3). **合理地使用关键字const可以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。简而言之，这样可以减少bug的出现。

**8. 关键字volatile有什么含意 并给出三个不同的例子。**

一个定义为volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子：

1). **并行设备的硬件寄存器**（如：状态寄存器）

2). 一个中断服务子程序中会**访问到的非自动变量(**Non-automatic variables)

3). 多线程应用中**被几个任务共享的变**量

回答不出这个问题的人是不会被雇佣的。我认为这是区分C程序员和嵌入式系统程序员的最基本的问题。嵌入式系统程序员经常同硬件、中断、RTOS等等打交道，**所用这些都要求volatile变量。不懂得volatile内容将会带来灾难。**

假设被面试者正确地回答了这是问题，看一下这家伙是不是直正懂得volatile完全的重要性。

1). 一个参数既可以是const还可以是volatile吗？解释为什么。**是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。**

2). 一个指针可以是volatile 吗？解释为什么。**是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个buffer的指针时**。

3). 下面的函数有什么错误：

|  |
| --- |
| int square(volatile int \*ptr)  {  return ptr \*ptr;  } |

这段代码的有个恶作剧。这段代码的目的是用来返指针\*ptr指向值的平方，但是，由于\*ptr指向一个volatile型参数，编译器将产生类似下面的代码：

|  |
| --- |
| int square(volatile int \*ptr)  {  int a,b;  a = \*ptr;  b = \*ptr;  return a \* b;  } |

由于\*ptr的值可能被意想不到地该变，因此a和b可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

|  |
| --- |
| long square(volatile int \*ptr)  {  int a;  a = \*ptr;  return a \* a;  } |

**11. 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(” Area = %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。**

**12 . 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。

**13. 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。

**14. 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。

**15. 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。

16. 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。

**其他常识**

（2）char \* const p char const \* p const char \*p 上述三个有什么区别？

答案：

char \* const p; // 常量指针，p的值不可以修改

char const \* p；//指向常量的指针，指向的常量值不可以改

const char \*p； //和char const \*p

14.int (\*s[10])(int) 表示的是什么啊？

答案：int(\*s[10])(int) 函数指针数组，每个指针指向一个int func(int param)的函数。

20.列举几种进程的同步机制，并比较其优缺点。

答案：原子操作，信号量机制，自旋锁 管程，会合，分布式系统

23.死锁的4个必要条件

答案：互斥、请求保持、不可剥夺、环路

24.死锁的处理

答案：鸵鸟策略、预防策略、避免策略、检测与解除死锁

28.数组和链表的区别

答案：数组：数据顺序存储，固定大小连表：数据可以随机存储，大小可动态改变