# static的用途

## 一．修饰变量

1、限制变量的作用域

2、修改变量的存储域（相对于局部变量而言）

3、具有记忆功能，保存上一次的值

## 二．修饰函数

1、在模块内静态函数只可被该模块的函数调用

# const的作用

1、表示为常量，不可修改

2、改变该变量的存储域

3、修饰函数参数时，防止参数被以外改变

# 实时系统

在规定的时间内完成特定的任务，具有实时性和可靠性

# **全局变量和局部变量的区别**

1、全局变量存储在静态数据区，局部变量存储在堆栈区

2、全局变量的生命周期是整个函数区间，局部变量的生命周期是声明该变量的函数区间

# 全局变量和静态全局变量的区别

1、全局变量和静态全局变量都存储在静态数据区

2、全局变量的作用域是整个函数，静态全局变量的作用雨是声明该变量的模块

# 数组与链表的区别

数组是一块连续的内存区间、大小固定，数据只能连续存储

链表可以是一块连续的内存区间也可以是不连续的内存区间，大小不固定，数据可以随机存储

内核系统分为

进程管理系统、内存管理系统、I/O管理系统、文件管理系统

# 进程与线程的区别

进程是拥有资源的基本单位，线程是调度和分配的基本单位

线程和进程都可以并发执行

堆与栈的区别

堆是用户申请和释放，栈是系统自行分配和释放空间有限

# 进程死锁的原因

资源的竞争和程序的顺序执行

# OSI（Open System Interconnection）模型

应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

TCP/IP模型

应用层、传输层、网络层、数据链路层、物理层

# Volatile变量的使用

告诉编译器这个变量随时可能改变，需要用到这个变量的时候直接从内存中读取

使用位置

1、并行设备的寄存器变量

2、中断函数中的非自动变量

3、多线程中的共享变量

一个参数既可以是const也可以为volatile，只读的状态寄存器

它是const是表示不应该试图去修改这个值；如果为volatile时

表示它有被意想不到的改变

# 中断与异常的区别

中断是外部硬件产生的电信号通过处理器的中断端口打断处理器的处理过程

异常是处理器内部执行到错误指令、或者在执行期间出现错误，必须靠内核处理的时候就会产生一个异常

# 嵌入式的中断函数

没有参数、没有返回值、不可执行不可重入的操作、不能执行浮点数的运算

# 三种基本的数据模型

层次模型、网状模型、关系模型

# vi的编辑器具工作模式

命令模式、编辑模式、底行模式

# 嵌入式系统的介绍

以应用为为中心，以计算机技术为基础，软硬件可裁剪，对功耗、性能、可靠性、成本有严格要求的计算机系统

# 可重入的问题

可重入的问题一般出现在多任务的系统中，可重入即可被打断的函数，也就是说一个函数可以在任何时候被打断

执行的过程

# TCP的三次握手

一、客户端向服务器发送一个同步信号

二、服务器接收到客户端的同步信号之后，给客户端回复一个ACK应答，同时也给客户端发送一个同步信号

三、客户端接收到服务器的同步信号和ACK后，给服务器回复一个ACK的应答

# TCP

传输控制协议，面向连接、可靠的字节流服务

# UDP

用户数据报协议，面向无连接的运输层协议

socket编程

客户端

socket->connect->write->close

## 服务器

socket->bind->accept->read->close

# 引用与指针的区别

1、引用必须被初始化，指针不一定需要初始化

2、引用被初始化之后不能在改变，指针初始化之后还能改变

3、不存在指向空的引用，但是存在只向空的指针

# 三种内存分配

1、从静态数据区分配：在程序编译的时候就已经分配好了，程序运行区间都存在

2、从栈上分配：在执行函数的时候，函数内部的局部变量可以在栈上申请空间，并随函数的结束而释放

3、从堆上分配：手动分配空间使用new和malloc，释放时使用deleted和free

# 指针函数

返回的类型是一个指针类型

函数指针

一个指针，这个指针指向的函数

# 野指针

野指针是很危险，if语句对它去不了任何的限制作用，因为野指针也是有指向某一个

地址，只是该地址有可能是合法也有可能是非法，最终有可能会导致程序崩溃

所以指针free或者delete完之后应该将指针指向NULL，避免产生野指针

# C与C++的区别

C是结构化的语言，面向过程，基于算法和数据结构，所考虑的如何通过一个过程或者函数的输入得到输出

C++是面向对象的编程语言，基于类、对象、继承，所考虑的是去创建一个对象模型，让这个模型契合与之

对应的问题，通过对象的状态得到输出或者实现过程的控制

sizeof与strlen的区别

1、sizeof是运算符，strlen是函数

2、sizeof的参数可以是任何类型的数据类型，strlen的参数只能是字符串

3、数组为参数时sizeof不会退化为指针，strlen会退化为指针

4、sizeof是编译时算出，为常量，strlen在运行时才会算出

# 重载问题

面向过程的C语言不支持重载，但是面向对象的C++支持重载。因为C语言和C++语言经过编译器编译后的symbol库内的

名称不一样。C++语言经过编译器编译后在symbol库中的名字包含了函数名和参数的类型、数量，靠这种机制实现重载功能

指针与数组的区别

指针可以随时指向任意类型的内存块；数组只能在静态数据区上创建或者栈上被创建

动态数据存放在堆栈区，栈是一种线性结构，堆是一种链式结构。线程拥有自己的私有栈

open与fopen的区别

1、前者属于低级的IO，后者属于高级的IO

2、前者返回的是文件的句柄（文件描述符），后者返回的是文件指针

3、前者无缓冲，后者有缓冲

4、前者与read、write、getc配合使用，后者与fread、fwrite、fopen配合使用

5、后者是在前者的基础上发展而来的

# 缓冲文件系统与非缓冲文件系统的区别

缓冲文件，在内存中开辟一个“缓冲区”，当程序读数据时，现将数据从磁盘中读到内存的“缓冲区”，缓冲区填满之后再将数据读到保存的变量中；当执行写文件时，现将数据写到内存的“缓冲区”中，缓冲区填满之后再将数据写到磁盘中。

缓冲文件系统借助文件结构指针来对文件进行管理，通过文件指针对文件进行访问。

非缓冲文件系统依赖操作系统，通过操作系统的功能对文件进行读写，是系统级的输入输出，它不设文件结构体指针，只能读写二进制文件，效率高、速度快。

# 指针常量与常量指针

下面代码的p有什么区别？

代码1：int i=10；int \* const p =&i；

代码2：int i=10；const int \* p=&i;

答：

代码1中 即指针常量。p是不可改变地址的指针，但是可以对它指向的内容进行修改。

代码2中 即常量指针。p是指向常量的指针，p所指向的地址内容是不能修改的，但是p本身可以修改。

# 指针运算

++（\*p）：先取指针指向的内容，然后+1，当前生效。++\*p等价

（\*p）++：先取指针指向的内容，然后+1，加1在语句执行后生效。\*p++等价

\*（p++）：p指针加1，先取P当前内容，然后给P+1

\*（++p）：p指针加1，先给p+1，然后取出内容

int 指针变量+1，地址加4

# 打印内存地址：

printf("%p",&a):自带0x。

printf("%p",&a)：不带0x十六进制。

# 字符与字符串

char c=‘a’   char \*cp=“a”

（1）c是字符变量，用于代替单个字符，cp是字符指针变量，用于传递字符串

（2）c替代字符常量‘a’  cp指向字符串常量

（3）‘a’占一个字节，“a”占俩字节

# 一维数组和指针

char \*p=“abcd”  char c[]=“abcd” 请问p++与c++是否相等？

p++：p是字符类型指针变量，p本身在内存中占4个字节，p是指向字符串常量的指针。代表字符串首地址，然后P地址+1。

c：c是数组名，也表示数组c的起始地址，该数组所占内存大小是字符串长度+1.\0 c是一个常量地址，c++是非法的。

# C语言预编译指令的作用是什么？有哪些预编译指令，作用分别是什么？

预处理过程扫描源代码，对其进行初步的转换，产生新的源代码提供给编译器。可见预处理过程先于编译器对源代码进行处理。

在C语言中，并没有任何内在的机制来完成如下一些功能：在编译时包含其他源文件、定义宏、根据条件决定编译时是否包含某些代码。要完成这些工作，就需要使用预处理程序。尽管在目前绝大多数编译器都包含了预处理程序，但通常认为它们是独立于编译器的。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对源代码进行响应的转换。预处理过程还会删除程序中的注释和多余的空白字符。

预处理指令是以#号开头的代码行。#号必须是该行除了任何空白字符外的第一个字符。#后是指令关键字，在关键字和#号之间允许存在任意个数的空白字符。整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。

# 下面是部分预处理指令：

        指令             用途

         #           空指令，无任何效果

         #include    包含一个源代码文件

         #define     定义宏

         #undef      取消已定义的宏

         #if         如果给定条件为真，则编译下面代码

         #ifdef      如果宏已经定义，则编译下面代码

         #ifndef     如果宏没有定义，则编译下面代码

         #elif       如果前面的#if给定条件不为真，当前条件为真，则编译下面代码

         #endif      结束一个#if……#else条件编译块

         #error      停止编译并显示错误信息

# 用预处理指令#define 声明一个常数，用以表明1年中有多少秒。

答：   #define SECONDS\_PER\_YEAR  (60 \* 60 \* 24 \* 365)UL

考点：

（1）#define 语法的基本知识（不能以分号结束，括号的使用）

（2）预处理器将计算常数表达式的值，因此，直接写出你是如何计算一年中有多少秒而不是计算出实际的值，这样更清晰明了。

（3）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）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，这说明考虑的很全面。

# 写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个。

答：#define MIN(A,B) （（A） <= (B) ? (A) : (B))

考点：

（1）标识#define在宏中应用的基本知识。这是很重要的。因为在  嵌入(inline)操作符 变为标准C的一部分之前，宏是方便产生嵌入代码的唯一方法，对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

（2）三重条件操作符的知识。这个操作符存在C语言中的原因是它使得编译器能产生比if-then-else更优化的代码，了解这个用法是很重要的。

（3）懂得在宏中把参数用括号括起来。

（4） 我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

        least = MIN(\*p++, b);

((\*p++) <= (b) ? (\*p++) : (b)) 这个表达式会产生副作用，指针p会作两次++自增操作。

# 预处理器标识#error的目的是什么？

停止编译并显示错误信息。

# 用变量a给出下面的定义

a) 一个整型数（An integer）

b)一个指向整型数的指针（ 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 toan intege）r

d)一个有10个整型数的数组（ An array of 10integers）

e) 一个有10个指针的数组，该指针是指向一个整型数的。（Anarray of 10 pointers to integers）

f) 一个指向有10个整型数数组的指针（ A pointerto an array of 10 integers）

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 functionthat takes an integer as an argument and returns an integer）

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（ An array of ten pointers to functions that take an integer argumentand return an integer ）

答案是：

a) int a; // An integer

b) int \*a; // A pointer to an integer

c) int \*\*a; // A pointer to a pointer to an integer

d) int a[10]; // An array of 10 integers

e) int \*a[10]; // An array of 10 pointers to integers

f) int (\*a)[10]; // A pointer to an array of 10 integers

g) int (\*a)(int); // A pointer to a function a that takes an integer argumentand returns an integer

h) int (\*a[10])(int); // An array of 10 pointers to functions that take aninteger argument and return an integer

# 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。

# 关键字const有什么含意？

一个变量不允许被改变，产生静态作用。使用const在一定程度上可以提高程序的安全性和可靠性。const推出的初始目的，正是为了取代预编译指令，消除它的缺点，同时继承它的优点。const修饰的数据类型是指常类型，常类型的变量或对象的值是不能被更新的。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。

# 关键字volatile有什么含意？

volatile是一个类型修饰符（type specifier），就像大家更熟悉的const一样，它是被设计用来修饰被不同线程访问和修改的变量。volatile的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。

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。简单地说就是防止编译器对代码进行优化。比如如下程序：

XBYTE[2]=0x55;

XBYTE[2]=0x56;

XBYTE[2]=0x57;

XBYTE[2]=0x58;

对外部硬件而言，上述四条语句分别表示不同的操作，会产生四种不同的动作，但是编译器却会对上述四条语句进行优化，认为只有XBYTE[2]=0x58（即忽略前三条语句，只产生一条机器代码）。如果键入volatile，则编译器会逐一的进行编译并产生相应的机器代码（产生四条代码）

# 位操作

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。

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;

voidset\_bit3(void)

{

a |= BIT3;

}

voidclear\_bit3(void)

{

a &= ~BIT3;

}

# 访问指定地址的内存

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;

一个较晦涩的方法是：

\*(int \* const)(0x67a9) =0xaa55;

建议你在面试时使用第一种方案。

# 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。

\_\_interruptdouble 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：

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。

\_\_interrupt用来声明一个函数是中断处理函数;在严格的ANSIC/C++模式下，也可以使用interrupt关键字来代替。中断处理函数要遵循特殊的寄存器保存规则和退出顺序，从而保证代码的安全。在C/C++程序中产生中断时，所有被中断子程序使用，或者被中断子程序调用的函数使用的状态都需要被保留。此外，\_\_interrupt定义的函数不能有参数，也没有返回值，即：

\_\_interrupt void int\_handler()

{

unsigned int flags;

...

}

# 运算中的类型转换

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。

# static关键字作用

（1） static修饰的局部变量它的数值是上一次函数调用结束之后的数值，在修饰变量的时候，static修饰的静态局部变量只执行一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。

（2）静态局部变量在定义的时候没有初始化，默认的初始值为0。

（3）static修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是extern外部声明也不可以。

（4）static修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。Static修饰的局部变量存放在全局数据区的静态变量区。初始化的时候自动初始化为0；

#include <stdio.h>

#include <stdlib.h>

int RecordBuffer[10];

int buffer\_write(unsigned int \*buffer , int size\_to\_write) ;

void fun();

int main()

{

fun();

fun();

system("PAUSE");

return 0;

}

void fun()

{

static int a = 0;

a++ ;

printf("a:%d\n",a);

}

# strlen与sizeof的区别

strlen：是计算字符串的长度，当遇到‘\0’时停止，不包括‘\0’。

sizeof：计算在内存中占得大小。要考虑字节对其等问题。32位系统，所有指针一般占4字节。

char \*a[3][4]占48个字节内存空间。

数组作为参数时，数组名退化为指针变量。

# 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）

解答：这一题主要容易错的地方就是：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。

#define SECONDS\_PER\_YEAR (60 \* 60 \* 24 \* 365)UL

# 2、写一个"标准"宏MIN，这个宏输入两个参数并返回较小的一个。

解答：这一题主要容易错的地方就是：懂得在宏中小心地把参数用括号括起来。

#define MIN(A,B) ((A)<=(B)?(A):(B))

当然，使用宏也是有副作用的。就拿这一个例子来说：该宏定义对MIN(\*p++, b)的作用结果是：((\*p++) <= (b) ? (\*p++) : (b)) 这个表达式会产生副作用，指针p会作两次++自增操作。

# 3、用变量a给出下面的定义：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。

解答：这一道题主要容易错的地方就是：函数指针、指针数组。

int (\*a[10])(int);

# 4、关键字static的作用是什么？

解答：在C语言中，关键字static有三个明显的作用：

在函数体中，一个被声明为静态的变量在这一函数被调用过程中只会被分配一次内存，且整个运行期间不会重新分配；

在函数体外、某个源文件内，一个被声明为静态的变量只可被该源文件内的所有函数访问，但不能被其他源文件的函数访问。它是一个本地的全局变量；

在某个源文件内，一个被声明为静态的函数仅仅只可以被这个源文件的其它函数调用。也就是说，这个函数被限制在声明它的源文件的本地范围之内使用。

# 5、关键字const的作用是什么？

解答：简单地说，const意味着常数。

const定义的变量，它的值不能被改变，在整个作用域中都保持固定；

同宏定义一样，可以避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改；

可以保护被修饰的东西，防止意外的修改，增强程序的健壮性。const是通过编译器在编译的时候执行检查来确保实现的。

const与指针

下面的声明都是什么意思：

const int a;

int const a;

const int \*a;

int \* const a;

const int \* const a;

int const \* const a;

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）；

最后两个意味着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）。

const与函数

const 通常用在函数形参中，如果形参是一个指针，为了防止在函数内部修改指针指向的数据，就可以用 const 来限制。比如在String的程序中有很多const修饰形参的情况：

void StringCopy(char\* strDestination, const char \*strSource);

const还可以表示该函数返回一个常量，放在函数的返回值的位置。比如：

const char \* GetString(void);

在类成员函数的声明和定义中，const放在函数的参数表之后，函数体之前，表示该函数的this指针是一个常量，不能修改该对象的数据成员。比如：

void getId() const;

# 6、关键字volatile有什么含意?并给出三个不同的例子。

解答：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：

存储器映射的硬件寄存器通常也要加voliate，因为每次对它的读写都可能有不同意义；

在中断函数中的交互变量，一定要加上volatile关键字修饰，这样每次读取非自动存储类型的值（全局变量，静态变量）都是在其内存地址中读取的，确保是我们想要的数据；

多任务环境下各任务间共享的标志应该加volatile。

一个参数既可以是const还可以是volatile吗？

可以的，例如只读的状态寄存器。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。软件不能改变，并不意味着我硬件不能改变你的值，这就是单片机中的应用。

参考文章：C语言中的volatile——让我保持原样。

一个指针可以是volatile 吗？

可以。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。

下面的函数有什么错误：

int square(volatile int \*ptr)

{

return \*ptr \* \*ptr;

}

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;

}

# 7、给定一个整型变量a，写两段代码，第一个设置a的bit3，第二个清除a的bit3。

解答：这道题清除a的bit3，使用“&=~”的方法。

#define BIT3 (0x1 << 3)

static int a;

void set\_bit3(void)

{

a |= BIT3;

}

void clear\_bit3(void)

{

a &= ~BIT3;

}

# 8、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。

解答：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（绝对地址）强制转换为一指针是合法的。

int \*ptr;

ptr = (int \*)0x67a9;

\*ptr = 0xaa55;

# 9、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;

}

解答：这个函数有太多的错误了，以至让人不知从何说起了：

ISR 不能传递参数，不能返回一个值；

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；

printf(char \* lpFormatString,…)函数会带来重入和性能问题，不能在ISR中采用。

关于这些要求的解释：

## a.为什么不能有返回值？

中断服务函数的调用是硬件级别的，当中断产生，pc指针强制跳转到对应的中断服务函数入口，进入中断具有随机性，并不是某段代码对其进行调用，那么如果有返回值它的返回值返回给谁?显然这个返回值毫无意义，如果有返回值，它必定需要进行压栈操作，这样一来何时出栈怎么出栈将变得无法解决。

## b.不能向ISR传递参数?

同理，也是由于这样会破坏栈的原因，因为函数传递参数必定会要求压栈出栈操作，由于进入中断服务函数的随机行，谁给它传递参数都成问题。

## c.ISR应尽可能的短小精悍?

如果某个中断频繁产生，而它对应的ISR相当的耗时，那么对中断的响应就会无限的延迟，会丢掉很多的中断请求。

## d.printf(char \* lpFormatString,…)函数会带来重入和性能问题，不能在ISR中采用。

这就涉及到一个中断嵌套问题，由于printf之类的glibc函数采用的是缓冲机制，这个缓冲区是共享的，相当于一个全局变量，第一层中断来时，它向缓冲里面写入一些部分内容，恰好这时来了个优先级更高的中断，它同样调用了printf，也向缓冲里面写入一些内容，这样缓冲区的内容就错乱了。

可重入函数主要用于多任务环境中，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。

## 中断服务函数与函数调用的联系与区别：

联系：两者都需要保护断点（即下一条指令地址）、跳至子程序或中断服务程序、保护现场、子程序或中断处理、恢复现场、恢复断点（即返回主程序）。两者都可实现嵌套，即正在执行的子程序再调另一子程序或正在处理的中断程序又被另一新中断请求所中断，嵌套可为多级。

区别：两者的根本区别主要表现在服务时间与服务对象不一样上。首先，调用子程序过程发生的时间是已知和固定的，即在主程序中的调用指令（CALL）执行时发生主程序调用子程序，调用指令所在位置是已知和固定的。而中断过程发生的时间一般的随机的，CPU在执行某一主程序时收到中断源提出的中断申请时，就发生中断过程，而中断申请一般由硬件电路产生，申请提出时间是随机的（软中断发生时间是固定的），也可以说，调用子程序是程序设计者事先安排的，而执行中断服务程序是由系统工作环境随机决定的；

其次，子程序完全为主程序服务的，两者属于主从关系，主程序需要子程序时就去调用子程序，并把调用结果带回主程序继续执行。而中断服务程序与主程序两者一般是无关的，不存在谁为谁服务的问题，两者是平行关系；

第三，主程序调用子程序过程完全属于软件处理过程，不需要专门的硬件电路，而中断处理系统是一个软、硬件结合系统，需要专门的硬件电路才能完成中断处理的过程；

第四，子程序嵌套可实现若干级，嵌套的最多级数由计算机内存开辟的堆栈大小限制，而中断嵌套级数主要由中断优先级数来决定，一般优先级数不会很大。

# 10、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");

}

解答：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。

还有一个重要的知识点：

通常情况下，在对int类型的数值作运算时，CPU的运算速度是最快的。在x86上，32位算术运算的速度比16位算术运算的速度快一倍。C语言是一个注重效率的语言，所以它会作整型提升，使得程序的运行速度尽可能地快。因此，你必须记住整型提升规则，以免发生一些整型溢出的问题。

整型提升是C程序设计语言中的一项规定，在表达式计算时（包括比较运算、算术运算、赋值运算等），比int类型小的类型（char, signed char, unsigned char, short, unsigned short等）首先要提升为int类型，然后再执行表达式的运算。

至于提升的方法，是根据原始类型进行位扩展（如果原始类型为unsigned char，进行零扩展，如果原始类型为signed char，进行符号位扩展）到32位。也就是说：

对于unsigned char，进行零扩展，即在左边的高位用 0 填充至32位；

对于signed char，进行符号位扩展。如果其符号位为0，则左边的高位用 0 填充至32位；如果其符号位为1，则左边的高位用 1 填充至32位。

# 11、评价下面的代码片断：

unsigned int compzero = 0xFFFF;

解答：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;

# 12、 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？

解答：动态分配将不可避免会产生问题：

内存泄露：内存泄露通常是程序自身编码缺陷造成，常见的 malloc内存后没有free等类似的操作， 系统在运行过程当中反复的malloc，吃掉系统内存，造成内核OOM，将某个进程需要申请内存的杀死而退出。

内存碎片：内存碎片是一个系统问题，反复的malloc和 free，而free后的内存又不能马上被系统回收利用。这个是因为负责动态分配内存的分配算法使得这些空闲的内存无法使用，这一问题的发生，原因在于这些空闲内存以小且不连续方式出现在不同的位置。

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");

函数malloc()的参数是可以时0的。

# 13、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？

解答：typedef更好。思考下面的例子：

dPS p1,p2;

tPS p3,p4;

如果是第一个define的扩展：

struct s \* p1, p2;

p1为指针，p2为结构体。很明显，不是我们想要的答案。

C语言测试是招聘嵌入式系统程序员过程中必须而且有效的方法。这些年，我既参加也组织了许多这种测试，在这过程中我意识到这些测试能为面试者和被面试者提供许多有用信息，此外，撇开面试的压力不谈，这种测试也是相当有趣的。

从被面试者的角度来讲，你能了解许多关于出题者或监考者的情况。这个测试只是出题者为显示其对ANSI标准细节的知识而不是技术技巧而设计吗？这是个愚蠢 的问题吗？如要你答出某个字符的ASCII值。这些问题着重考察你的系统调用和内存分配策略方面的能力吗？这标志着出题者也许花时间在微机上而不是在嵌入 式系统上。如果上述任何问题的答案是"是"的话，那么我知道我得认真考虑我是否应该去做这份工作。

从面试者的角度来讲，一个测试也许能从多 方面揭示应试者的素质：最基本的，你能了解应试者C语言的水平。不管怎么样，看一下这人如何回答他不会的问题也是满有趣。应试者是以好的直觉做出明智的选 择，还是只是瞎蒙呢？当应试者在某个问题上卡住时是找借口呢，还是表现出对问题的真正的好奇心，把这看成学习的机会呢？我发现这些信息与他们的测试成绩一 样有用。

有了这些想法，我决定出一些真正针对嵌入式系统的考题，希望这些令人头痛的考题能给正在找工作的人一点帮助。这些问题都是我这些年实际碰到的。其中有些题很难，但它们应该都能给你一点启迪。

这个测试适于不同水平的应试者，大多数初级水平的应试者的成绩会很差，经验丰富的程序员应该有很好的成绩。为了让你能自己决定某些问题的偏好，每个问题没有分配分数，如果选择这些考题为你所用，请自行按你的意思分配分数。

# 预处理器（Preprocessor）

## 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，直接写出你是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。

## 2 . 写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个。

#define MIN(A,B) （（A） <= (B) ? (A) : (B))

这个测试是为下面的目的而设的：

1) 标识#define在宏中应用的基本知识。这是很重要的。因为在 嵌入(inline)操作符 变为标准C的一部分之前，宏是方便产生嵌入代码的唯一方法，对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

2)三重条件操作符的知识。这个操作符存在C语言中的原因是它使得编译器能产生比if-then-else更优化的代码，了解这个用法是很重要的。

3) 懂得在宏中小心地把参数用括号括起来

4) 我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

least = MIN(\*p++, b);

## 3. 预处理器标识#error的目的是什么？

如果你不知道答案，请看参考文献1。这问题对区分一个正常的伙计和一个书呆子是很有用的。只有书呆子才会读C语言课本的附录去找出象这种问题的答案。当然如果你不是在找一个书呆子，那么应试者最好希望自己不要知道答案。

# 死循环（Infinite loops）

## 4. 嵌入式系统中经常要用到无限循环，你怎么样用C编写死循环呢？

这个问题用几个解决方案。我首选的方案是：

while(1)

{

}

一些程序员更喜欢如下方案：

for(;;)

{

}

这个实现方式让我为难，因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这个作为方案，我将用这个作为一个机会去探究他们这样做的基本原理。如果他们的基本答案是："我被教着这样做，但从没有想到过为什么。"这会给我留下一个坏印象。

第三个方案是用 goto

Loop:

...

goto Loop;

应试者如给出上面的方案，这说明或者他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的BASIC/FORTRAN程序员。

# 数据声明（Data declarations）

## 5. 用变量a给出下面的定义

a) 一个整型数（An integer）

b)一个指向整型数的指针（ 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 intege）r

d)一个有10个整型数的数组（ 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）

f) 一个指向有10个整型数数组的指针（ A pointer to an array of 10 integers）

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）

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（ An array of ten pointers to functions that take an integer argument and return an integer ）

答案是：

a) int a; // An integer

b) int \*a; // A pointer to an integer

c) int \*\*a; // A pointer to a pointer to an integer

d) int a[10]; // An array of 10 integers

e) int \*a[10]; // An array of 10 pointers to integers

f) int (\*a)[10]; // A pointer to an array of 10 integers

g) int (\*a)(int); // A pointer to a function a that takes an integer argument and returns an integer

h) int (\*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

人 们经常声称这里有几个问题是那种要翻一下书才能回答的问题，我同意这种说法。当我写这篇文章时，为了确定语法的正确性，我的确查了一下书。但是当我被面试 的时候，我期望被问到这个问题（或者相近的问题）。因为在被面试的这段时间里，我确定我知道这个问题的答案。应试者如果不知道所有的答案（或至少大部分答 案），那么也就没有为这次面试做准备，如果该面试者没有为这次面试做准备，那么他又能为什么出准备呢？

# Static

## 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。

# Const

## 7．关键字const有什么含意？

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。

# Volatile

## 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。搞嵌入式的家伙们经常同硬件、中断、RTOS等等打交道，所有这些都要求用到volatile变量。不懂得volatile的内容将会带来灾难。

假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，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。

2); 一个指针可以是volatile 吗？解释为什么。

3); 下面的函数有什么错误：

int square(volatile int \*ptr)

{

return \*ptr \* \*ptr;

}

下面是答案：

1)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); 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) 这段代码有点变态。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;

}

# 位操作（Bit manipulation）

## 9. 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。

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

## 10. 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。

中断（Interrupts）

## 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("/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。

代码例子（Code examples）

## 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。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。

## 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。不管如 何，你就当是这个娱乐吧...

# 动态内存分配（Dynamic memory allocation）

## 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。

# Typedef

## 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。

# 晦涩的语法

## 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。

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。

好了，伙计们，你现在已经做完所有的测试了。这就是我出的C语言测试题，我怀着愉快的心情写完它，希望你以同样的心情读完它。如果是认为这是一个好的测试，那么尽量都用到你的找工作的过程中去吧。天知道也许过个一两年，我就不做现在的工作，也需要找一个。