/\*\*c面试题库整理

目的：提高学员c面试能力

时间：2013.03 整理人：hejie

\*\*/

/\*\*面试题库修改1：

修改日志：时间 2013.03.14 修改人：hejie

修改内容：1.调整不合理分类

2.删除部分重复题目

3.添加部分题目，题库更加丰富

4.修改答案剖析，使之更合理，标准

\*\*/

**基础部分**

**关键字**

**试题1：关键字const有什么含意？**

答案：

（1）可以修饰const 常变量

（2）const可以修饰函数的参数、返回值，甚至函数的定义体。被const修饰的东西 都受到强制

**试题2：分析以下代码定义，说明其特性**

**const int a;**

**int const a;**

**const int \*a;**

**int \* const a;**

**int const \* a const;**

答案： 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）。

**思考3：const修饰的常量与宏的区别**

答案：

const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

**试题4： 关键字volatile有什么含意?并给出三个不同的例子。**

答案：

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。

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保护，可以预防意外的变动，能提高程序的健壮性。

**试题5：**

**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;**

**}**

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;

}

**试题6. 关键字static的作用是什么？**

答案：

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。

总结：

static全局变量与普通的全局变量有什么区别：static全局变量只初使化一次，防止在其他文件单元中被引用;

　　static局部变量和普通局部变量有什么区别：static局部变量只被初始化一次，下一次依据上一次结果值；

　　static函数与普通函数有什么区别：static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝.

**试题7. 如何引用一个已经定义过的全局变量？**

　　答案：

　　可以用引用头文件的方式，也可以用extern关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变理，假定你将那个变写错了，那么在编译期间会报错，如果你用extern方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。

**试题8：在C++ 程序中调用被 C编译器编译后的函数，为什么要加 extern “C”？**

例子：

**#ifndef \_\_INCvxWorksh**

**#define \_\_INCvxWorksh**

**#ifdef \_\_cplusplus**

**extern "C" {**

**#endif**

**/\*...\*/**

**#ifdef \_\_cplusplus**

**}**

**#endif**

**#endif /\* \_\_INCvxWorksh \*/**

答案：

由于c语言是没有重载函数的概念的，所以c编译器编译的程序里，所有函数只有函数名对应的入口。而由于c++语言有重载函数的概念，如果只有函数名对应的入口，则会出现混淆，所以c++编译器编译的程序，应该是函数名+参数类型列表对应到入口。

假设某个函数的原型为： void foo(int x, int y);

该函数被C编译器编译后在库中的名字为\_foo，而C++编译器则会产生像\_foo\_int\_int之类的名字。

C++提供了C连接交换指定符号extern“C”来解决名字匹配问题。

**思考9：如何判断一段程序是由C 编译程序还是由C++编译程序编译的？**

（1）如果是要你的代码在编译时发现编译器类型，就判断\_cplusplus或\_STDC\_宏。

#ifdef \_\_cplusplus

       cout<<"c++";

#else

       cout<<"c";

#endif

 如果要判断已经编译的代码的编译类型，就用nm查一下输出函数符号是否和函数名相同。  
（2）注意，因为mian函数是整个程序的入口，所以mian是不能有重载的，所以，如果一个程序只有main函数，是无法确认是c还是c++编译器

编译的可以通过nm来查看函数名入口     
 如一个函数     
 int foo(int i, float j)

c编译的程序通过nm查看

foo         0x567xxxxxx    (地址)

c++编译程序，通过nm查看

foo(int, float)    0x567xxxxxx

  另外，如果要在c++编译器里使用通过c编译的目标文件，必须通知c++编译器， extern   "c"  foo;

**预处理命令（宏）**

**试题10：什么是预编译，何时需要预编译？**

**答:**

就是指程序执行前的一些预处理工作,主要指#表示的.

１)、总是使用不经常改动的大型代码体。

２)、程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

**试题11.说出以下 预处理器标识的目的意义**

指令 用途

**#**

**#include**

**#define**

**#undef**

**#if**

**#ifdef**

**#ifndef**

**#elif**

**#endif**

**#error**

答案：

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 编译时检测错误，

使程序员更好的掌握代码-

**试题12：头文件中的 ifndef/define/endif 干什么用？**

答：防止该头文件被重复引用。

**试题13：＃include “filename.h”和＃include <filename.h>的区别?**

答案：

对于＃include <filename.h>编译器从标准库开始搜索filename.h

对于＃include “filename.h”编译器从用户工作路径开始搜索filename.h

**试题14： 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）**

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

剖析：

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) 再思考进一步用到UL（表示无符号长整型）

**试题15. 写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个。**

答案：

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

剖析：

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

注：谨慎地将宏定义中的“参数”和整个宏用用括弧括起来。所以，严格地讲，下述解答是错误的：

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

#define MIN(A,B) (A ＜= B ? A : B )

#define MIN(A,B) ((A) ＜= (B) ? (A) : (B)); //这个解答在宏定义的后面加“;”

**试题16： #define MIN(A,B) ((A) ＜= (B) ? (A) : (B))对MIN(\*p++, b)的有什么后果？**

答案：

((\*p++) ＜= (b) ? (\*p++) : (\*p++))

　　这个表达式会产生副作用，指针p会作多次++自增操作。

**试题17.分析以下代码，说出输出结果**

#define swap(a,b) a=a+b;b=a-b;a=a-b;

void main()

{

int x=5, y=10;

swap (x,y);

printf(“%d %dn”,x,y);

}

**答案**： 10, 5

**试题18：宏定义的多语句错误，分析以宏定义**

**#define D(a,b) a+b;\**

**a++;**

分析:

应用时：if（XXX）

D(a.b);

else

解决办法 用do{ } while(0)

#define D(a,b) do{a+b;\

a++;}while(0)

思考while(0)后没有分号

**试题19：分析一下两个定义，哪种方法更好呢？（如果有的话）为什么？**

**#define dPS struct s \***

**typedef struct s \* tPS;**

**分析**：

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。

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。

假如定义函数指针：

typedef void (\*fun)(void);

#define FUN(x) void(\*x)(void)

**运算符·表达式·数据类型·优先级**

**试题20：分别给出BOOL，int，float，指针变量 与“零值”比较的 if 语句（假设变量名为var）**

分析：

　BOOL型变量：if(!var)

　int型变量： if(var==0)

　float型变量：

　const float EPSINON = 0.00001;

　if ((x ＞= - EPSINON) && (x ＜= EPSINON)

　指针变量：　　if(var==NULL)

分析：

　　考查对0值判断的“内功”，BOOL型变量的0判断完全可以写成if(var==0)，而int型变量也可以写成if(!var)，

指针变量的判断也可以写成if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。

　　一般的，如果想让if判断一个变量的“真”、“假”，应直接使用if(var)、if(!var)，表明其为“逻辑”判断；如果用if判断一个数值型变量(short、int、long等)，

应该用if(var==0)，表明是与0进行“数值”上的比较；而判断指针则适宜用if(var==NULL)，这是一种很好的编程习惯。

　　浮点型变量并不精确，所以不可将float变量用“==”或“！=”与数字比较，应该设法转化成“＞=”或“＜=”形式。如果写成if (x == 0.0)，则判为错

**试题21：**

**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。**

分析：

#define BIT3 (0x1 << 3)

static int a;

void set\_bit3(void)

{

a |= BIT3;

}

void clear\_bit3(void)

{

a &= ~BIT3;

}

**补充22：了解可以位操作的另一个知识位域**

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。

例如在存放一个开关量时，只有0和1 两种状态， 用一位二进位即可。

为了节省存储空间，并使处理简便，Ｃ语言又提供了一种数据结构，称为“位域”或“位段”。

所谓“位域”是把一个字节中的二进位划分为几个不同的区域， 并说明每个区域的位数。

每个域有一个域名，允许在程序中按域名进行操作。 这样就可以把几个不同的对象用一个字节的二进制位域来表示。

一、位域的定义和位域变量的说明位域定义与结构定义相仿，其形式为：

struct 位域结构名 { 位域列表 }; 其中位域列表的形式为： 类型说明符 位域名：位域长度

　　例如：

　　struct bs

　　{

　　　int a:8;

　　　int b:2;

　　　int c:6;

　　};

　　位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：

　　struct bs

　　{

　　　int a:8;

　　　int b:2;

　　　int c:6;

　　}data;

　　说明data为bs变量，共占两个字节。其中位域a占8位，位域b占2位，位域c占6位。对于位域的定义尚有以下几点说明：

　　一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。

也可以有意使某位域从下一单元开始。例如：

　　struct bs

　　{

　　　unsigned a:4

　　　unsigned :0 /\*空域\*/

　　　unsigned b:4 /\*从下一单元开始存放\*/

　　　unsigned c:4

　　}

　　在这个位域定义中，a占第一字节的4位，后4位填0表示不使用，b从第二字节开始，占用4位，c占用4位。

　　由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过8位二进位。

　　位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

　　struct k

　　{

　　　int a:1

　　　int :2 /\*该2位不能使用\*/

　　　int b:3

　　　int c:2

　　};

　　从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。

　　位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名?位域名 位域允许用各种格式输出。

**试题23：分析以下代码打印输出结果**

typedef struct

{

int a:2;

int b:2;

int c:1;

}test;

test t;

t.a = 1;

t.b = 3;

t.c = 1;

printf("%d",t.a);

printf("%d",t.b);

printf("%d",t.c);

分析：

t.a为01,输出就是1

t.b为11，输出就是－1

t.c为1，输出也是-1

**字节对齐**

**试题31：**

**typedef union {long i; int k[5]; char c;} DATE;**

**struct data { int cat; DATE cow; double dog;} too;**

**DATE max;**

**则语句 printf("%d",sizeof(struct date)+sizeof(max));的执行结果是：\_\_\_\_\_\_**

分析：

编译器自动对齐的原因：为了提高程序的性能，数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；然而，对齐的内存访问仅需要一次访问

DATE是一个union, 变量公用空间. 里面最大的变量类型是int[5], 占用20个字节. 所以它的大小是20

data是一个struct, 每个变量分开占用空间. 依次为int4 + DATE20 + double8 = 32.

所以结果是 20 + 32 = 52.

当然...在某些16位编辑器下, int可能是2字节,那么结果是 int2 + DATE10 + double8 = 20

**数据类型提升（隐形数据转换）**

**试题25：**

**void foo(void)**

**{**

**unsigned int a = 6;**

**int b = -20;**

**(a+b > 6) ? puts("> 6") : puts("<= 6");**

**}**

分析：

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。

**试题26：分析以下代码，写出打印结果**

**main()**

**{**

**int x=20,y=35;**

**x=y++ + x++;**

**y= ++y + ++x;**

**printf(“%d%dn”,x,y);**

**}**

**答案**: 5794

**试题27：分析以下代码的输出结果**

**int a=1 ,b=2,c=3**

**while(a<b<c){**

**int t;**

**t=a;a=b;b=t;c--;**

**}**

**printf("%d\n",c)**

答案：为0；a<b<c 解释为(a<b)<c;

**试题 若w=1,x=2,y=3,z=4,则条件表达式w<x?w:y<z?y：z的值是**

答案 1

**能力部分**

**内存章节**

**试题28：linux 的进程内存分布**

分析：

对于linux的进程内存分布，主要是由从小到大的地址空间分布，从低地址到高地址依次是：文本段（text），数据段，BSS段，堆，栈。

各个区段详细如下：

文本段：文本段中存放的是代码，只读数据，字符串常量（我们通常说保存在文字常量中，实际就是在文本段）

数据段：数据段用来存放可执行文件中已经初始化的全局变量，全局变量又可细分为全局变量和程序分配的static静态变量

BSS：BSS段包含了程序中未初始化的全局变量，在内存中全局变量全部初始化为0

堆（heap）：堆主要用来存放进程中动态分配的内存段，其大小不固定，可动态扩张或缩减。当进程使用malloc等函数分配内存时，

新分配的内存就被动态添加到堆上，相当于堆被扩张。当利用free等函数释放内存时，被释放的内存被从堆中剔除，相应于堆被缩减

堆的物理内存是由程序申请，并由程序释放

栈：栈是用户程序存放临时空间的局部变量，也就是我们所说的{}中定义的变量（但不包括static声明的变量，static意味着变量被存储到数据段）。

除此以外，在函数被调用时其参数也被压入发起调用的进程栈中，并且待到调用结束后，

函数的返回值也被压入栈中，由于栈的先进后出原则，所以栈特别方便用来保存或恢复调用现场，

从这个意义上讲，我们可以把堆栈看做一个寄存，交换临时数据的内存区

**试题29：描述内存分配方式以及它们的区别?**

分析：

1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。

2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集。

3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc 或new 申请任意多少的内存，程序员自己负责在何时用free 或delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多。

**试题30：简述变量的作用域，生存周期，内存的分配方式（全局变量，静态全局变量，局部变量，静态局部变量的区别）**

**分析：**

全局变量具有全局作用域。全局变量只需在一个源文件中定义，就可以作用于所有的源文件。当然，其他不包括全局变量定义的源文件需要用extern关键字再次声明这个全局变量。

静态局部变量具有局部作用域。它只被初始化一次，自从第一次初始化直到程序结束都一直存在，他和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。

局部变量也只有局部作用域，他是自动对象，他在程序运行期间不是一直存在，而是只在函数执行期间存在，函数的一次调用结束后，变量就被撤销，其所占用的内存也被收回。

静态全局变量也具有全局作用域，他与全局变量的区别在于如果程序包含多个文件的话，他作用于定义它的文件里，不能作用到其他文件里，即被static关键字修饰过的变量具有文件作用域。这样即使两个不同的源文件都定义了相同的静态全局变量，他们也是不同的变量。

从分配内存空间看：

全局变量、静态局部变量、静态全局变量都在静态存储区分配空间，而局部变量在栈分配空间。

全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上没有什么不同。区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

**控制语句**

**试题32：语句for( ；1 ；)有什么问题？它是什么意思？**

　　答：和while(1)相同。

**试题33：do……while和while……do有什么区别？**

　　答：前一个循环一遍再判断，后一个判断以后再循环

**试题34：请简述以下两个for循环的优缺点**

**for (i=0; i<N; i++)**

**{**

**if (condition)**

**DoSomething();**

**else**

**DoOtherthing();**

**}**

**if (condition)**

**{**

**for (i=0; i<N; i++)**

**DoSomething();**

**}**

**else**

**{**

**for (i=0; i<N; i++)**

**DoOtherthing();**

**}**

**分析：**

第一个

优点：程序简洁

缺点：多执行了N-1次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。

第二个

优点：循环的效率高

缺点：程序不简洁

**试题35：以下程序的输出结果是 \_\_\_\_\_\_\_\_**

**#include <stdio.h>**

**void main( )**

**{**

**int a=-1,b=1,k;**

**if((++a<0)&&!(b--<=0))**

**printf("%d %d\n",a,b);**

**else**

**printf("%d %d\n", b, a);**

**}**

**答案：1，0 //短路效应**

**试题36：以下程序的输出结果是 \_\_\_\_\_\_\_\_**

**#include <stdio.h>**

**void main()**

**{**

**int a,b,c,d,x;**

**a=c=0;**

**b=1;d=20;**

**if(a) d=d-10;**

**else if(!b)**

**if(!c)x=15;**

**else x=25;**

**printf("%d\n",d);**

**}**

**答案：20 //if..... else**

**数组与指针**

**试题37：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）**

**分析：**

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

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

**试题38. 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0xaa55。**

分析：

int \*ptr;

ptr = (int \*)0x67a9;

\*ptr = 0xaa55;

一个较晦涩的方法是：

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

}

**试题39：分析代码**

unsigned char \*p1;

　　unsigned int \*p2;

　　p1=(unsigned char \*)0x801000;

　　p2=(unsigned int \*)0x810000;

　　请问p1+5= ？

　　　　p2+5= ？

**试题40：前面练习了宏，用宏得到指针地址上的一个字节**

#define MEM\_B(X) (\*((char \*const )(X)))

**试题41：**

**char szstr[10];**

**strcpy(szstr,"0123456789");**

剖析：

数组宽度的问题，预留'\0'

**试题42：**

**char a[] = "hello world";**

**char b[] = "hello world";**

**const a1[] = "hello world";**

**const b1[] = "hello world";**

**char \*p = "hello world";**

**char \*q = "hello world";**

**写出结果：**

**sizeof(a) = ？;**

**sizeof(p) = ？;**

**a == b ? 1 : 0;**

**a1 == b1 ? 1 : 0;**

**p == q ? 1 : 0**

**void Func(char a[100])**

**{**

**sizeof(a) = ？;**

**}**

**思考43：void func(char a[]) 与 void Func(char a[100]) 有什么区别？**

分析：

sizeof(a) = 12;

sizeof(p) = 4;

a == b ? 1 : 0;---0

a1 == b1 ? 1 : 0;---0

p == q ? 1 : 0；---1

void Func(char a[100])

{

sizeof(a) = ？; // 4 字节而不是100 字节

}

函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

**补充思考44：sizeof与strlen的区别**

sizeof()和初不初始化，没有关系；

　　 strlen()和初始化有关。

**试题45：分析以下代码**

**#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;**

**}**

剖析：

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。

**试题46，分析以下代码**

**main()**

**{**

**int a[5]={1,2,3,4,5};**

**int \*ptr=(int \*)(&a+1);**

**printf("%d,%d",\*(a+1),\*(ptr-1));**

**}**

分析：

　　输出：2,5

　　\*(a+1）就是a[1]，\*(ptr-1)就是a[4],执行结果是2，5

　　&a+1不是首地址+1，系统会认为加一个a数组的偏移，是偏移了一个数组的大小（本例是5个int）

　　int \*ptr=(int \*)(&a+1);

　　则ptr实际是&(a[5]),也就是a+5

　　原因如下：

　　&a是数组指针，其类型为 int (\*)[5];

　　而指针加1要根据指针类型加上一定的值，不同类型的指针+1之后增加的大小不同。

　　a是长度为5的int数组指针，所以要加 5\*sizeof(int)

　　所以ptr实际是a[5]

　　但是prt与(&a+1)类型是不一样的(这点很重要)

　　所以prt-1只会减去sizeof(int\*)

　　a,&a的地址是一样的，但意思不一样，a是数组首地址，也就是a[0]的地址，&a是对象（数组）首地址，

a+1是数组下一元素的地址，即a[1],&a+1是下一个对象的地址，即a[5].

**试题47：分析以下代码的输出结果**  
**main()**

**{**

**char \*\*p;**

**char \*m[] = {“Welcome \n”, “to \n”,“join \n”, “us! \n” };**

**p = m;**

**printf(“%s\n”,\*p++);**

**printf(“%c\n”,\*\*p);**

**}**

**答案：“Welcome”和“t”**

**分析：\*\*p是个二级指针， char \*m[]是个指针数组，数组里面的都存放一个指向char 的数组**

**函数与指针**

**试题48：要对绝对地址0x100000赋值，我们可以用(unsigned int\*)0x100000 = 1234;那么要是想让程序跳转到绝对地址是0x100000去执行，应该怎么做？**

分析：

　　\*((void (\*)( ))0x100000 ) ( );

　　首先要将0x100000强制转换成函数指针,即:

　　(void (\*)())0x100000

　　然后再调用它:

　　\*((void (\*)())0x100000)();

　　用typedef可以看得更直观些:

　　typedef void(\*)() voidFuncPtr;

　　\*((voidFuncPtr)0x100000)();

**试题49：用变量a给出下面的定义**

**1) 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）**

**2) 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 ）**

分析：

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

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

**试题50：看看下面的一段程序有身那么错误？**

**swap(int \*p1,int \*p2){**

**int \*p;**

**\*p=\*p1;**

**\*p1=\*p2;**

**\*p2 = \*p;**

**}**

**int swap2(int a, int b)**

**{**

**int temp;**

**temp=a;**

**b=a;**

**a=temp;**

**return 0;**

**}**

**答案：p为野指针。明白函数传递是值传递。**

**联系数据交换的例子，清楚空间换时间，时间换空间的理念**

**如：直接交换，int p =a;a=b;b=p;空间换时间**

**a = a+b;b= a-b;a = a-b;时间换空间，当然还有其他的例子**

**51，用函数指针简略实现回调函数机制**

**typedef void ( \*FUN )(void) ；**

**void caller(FUN ptr)**

**{**

**( \*ptr)(); /\* 调用ptr指向的函数 \*/**

**}**

**void func();**

**int main()**

**{**

**FUN p = func;**

**caller(p); /\* 传递函数地址到调用者 \*/**

**}**

**综合分析**

**试题52：请问以下代码有什么问题：**

**int main()**

**{**

**char a;**

**char \*str=&a;**

**strcpy(str,"hello");**

**printf(str);**

**return 0;**

**}**

剖析：

　 没有为str分配内存空间，将会发生异常。

问题出在将一个字符串复制进一个字符变量指针所指地址。

虽然可以正确输出结果，但因为越界进行内在读写而导致程序崩溃。

**试题53：分析一下代码，有什么错？**

**char\* s="AAA";**

**printf("%s",s);**

**s[0]='B';**

**printf("%s",s);**

剖析：

"AAA"是字符串常量。s是指针，指向这个字符串常量，所以声明s的时候就有问题。

　　cosnt char\* s="AAA";

　　然后又因为是常量，所以对是s[0]的赋值操作是不合法的。

**试题54：分析一下代码问题，及输出结果**

**main()**

**{**

**char \*p1;**

**char \*p2;**

**p1=(char \*)malloc(25);**

**p2=(char \*)malloc(25);**

**strcpy(p1,”Cisco”);**

**strcpy(p2,“systems”);**

**strcat(p1,p2);**

**printf(“%s”,p1);**

**}**

剖析：

输出结果：Ciscosystems

注意：初始化指针NULL，free防止内存泄露

strcpy的使用的条件

**试题55：分析一下代码输出结构**

**main()**

**{**

**char \*p1=“name”;**

**char \*p2;**

**p2=(char\*)malloc(20);**

**memset (p2, 0, 20);**

**while(\*p2++ = \*p1++);**

**printf(“%sn”,p2);**

**}**

剖析：

无效输出，乱码。

p2已更改。

代码规范，char \*p2 = NULL;

free

未判断内存是否申请成功

**试题56：分析一下代码，请问运行Test函数会有什么样的结果？**

**void GetMemory(char \*p)**

**{**

**p = (char \*)malloc(100);**

**}**

**void Test(void)**

**{**

**char \*str = NULL;**

**GetMemory(str);**

**strcpy(str, "hello world");**

**printf(str);**

**}**

分析：

段错误。因为GetMemory并不能传递动态内存，Test函数中的 str一直都是 NULL。

未判断内存是否申请成功

**试题57：分析一下代码，请问运行Test函数会有什么样的结果？**

**char \*GetMemory(void)**

**{**

**char p[] = "hello world";**

**return p;**

**}**

**void Test(void)**

**{**

**char \*str = NULL;**

**str = GetMemory();**

**printf(str);**

**}**

剖析：

输出是乱码。

因为GetMemory返回的是指向“栈内存”的指针，该指针的地址不是 NULL，但其原现的内容已经被清除，新内容不可知。

未判断内存是否申请成功

**思考58：如何使p 不被释放**

**试题59：分析一下代码，请问运行Test函数会有什么样的结果？**

**void GetMemory2(char \*\*p, int num)**

**{**

**\*p = (char \*)malloc(num);**

**}**

**void Test(void)**

**{**

**char \*str = NULL;**

**GetMemory(&str, 100);**

**strcpy(str, "hello");**

**puts(str);**

**}**

剖析：

（1）能够输出hello

（2）内存泄漏，没有free

未判断内存是否申请成功

**试题60：　问输出结果是什么？**

**void GetMemory(char \*\*p,int num)**

**{**

**\*p=(char \*)malloc(num);**

**}**

**int main()**

**{**

**char \*str=NULL;**

**GetMemory(&str,100);**

**strcpy(str,"hello");**

**free(str);**

**if(str!=NULL)**

**{**

**strcpy(str,"world");**

**}**

**printf("\n str is %s",str);**

**getchar();**

**}**

剖析：

输出str is world。

free 只是释放的str指向的内存空间,它本身的值还是存在的.所以free之后，成为野指针，有一个好的习惯就是将str=NULL.

此时str指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出world来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。

当你malloc一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），

然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。

但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的。

注：

对内存操作的考查主要集中在：

（1）指针的理解；

（2）变量的生存期及作用范围；

（3）良好的动态内存申请和释放习惯。

**代码实现**

**试题61.热身练习：冒泡排序法**

**void fun(int a[],int n){**

**int i,j,k;**

**for(i =1;i<n;i++)**

**for(j=0;j<n-1;j++)**

**if(a[j]>a[j+i]){**

**k=a[j];**

**a[j]=a[j+1];**

**a[j+1]=k;**

**}**

**}**

**试题62：编写一个函数，作用是把一个char组成的字符串循环右移n个。比如原来是“abcdefghi”如果n=2，移位后应该是“hiabcdefgh”**

参考代码：

解答1：

//pStr是指向以'\0'结尾的字符串的指针

//steps是要求移动的n

void LoopMove ( char \*pStr, int steps )

{

　int n = strlen( pStr ) - steps;

　char tmp[MAX\_LEN];

　strcpy ( tmp, pStr + n );

　strncpy ( tmp + steps, pStr,n);

　\*( tmp + strlen ( pStr ) ) = '\0';

　strcpy( pStr, tmp );

}

解答2：

void LoopMove ( char \*pStr, int steps )

{

　int n = strlen( pStr ) - steps;

　char tmp[MAX\_LEN];

　memcpy( tmp, pStr + n, steps );

　memcpy(pStr + steps, pStr, n );

　memcpy(pStr, tmp, steps );

}

　　剖析：

　　这个试题主要考查面试者对标准库函数的熟练程度，在需要的时候引用库函数可以很大程度上简化程序编写的工作量。

　　最频繁被使用的库函数包括：

　　（1） strcpy

　　（2） memcpy

　　（3） memset

还要考虑时间与空间的复杂度的问题，时间换空间还是空间换时间的问题。

**试题63．编写my\_memcpy函数，实现与库函数memcpy类似的功能，不能使用任何库函数；**

void\* memcpy1(void\* dest, void\* source, size\_t count)

{

//copy from lower address to higher address

assert((dest !=NULL)&&(source != NULL));

char \*des = (char \*)dest;

char \*src = (char \*)source;

while (count--)

\*des++ = \*src++;

return dest ;

}**试题64．编写my\_strcpy函数，实现与库函数strcpy类似的功能，不能使用任何库函数；**

答：char\* my\_strcpy(char\* strdest, const char\* strsrc)

{

assert((strdest != NULL) && (strsrc != NULL));

char\* address = strdest;

while((\*strdest++ = \*strsrc++) != '\0');

strdest = '\0';

return address;

}

**编写65一个my\_itoa的函数，实现与库函数itoa类似的功能**

int getlen(char \*s){

int n;

for(n = 0; \*s != '\0'; s++)

n++;

return n;

}

void reverse(char s[])

{

int c,i,j;

for(i = 0,j = getlen(s) - 1; i < j; i++,j--){

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

void my\_itoa(int n,char s[])

{

int i,sign;

if((sign = n) < 0)

n = -n;

i = 0;

do{/\*以反序生成数字\*/

s[i++] = n%10 + '0';/\*get next number\*/

}while((n /= 10) > 0);/\*delete the number\*/

if(sign < 0)

s[i++] = '-';

s[i] = '\0';

reverse(s);

}

**试题66****. 代码是把一个字符串倒序，如“abcd”倒序后变为“dcba”**

　int getlen(char \*s){

int n;

for(n = 0; \*s != '\0'; s++)

n++;

return n;

}

void reverse(char s[])

{

int c,i,j;

for(i = 0,j = getlen(s) - 1; i < j; i++,j--){

c = s[i];

s[i] = s[j];

s[j] = c;

}

**试题67、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：

typedef enum{false = 0, true} bool;

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] );

}

试题：已知链表的头结点head,写一个函数把这个链表逆序 ( Intel)

Node \* ReverseList(Node \*head) //链表逆序  
{  
if ( head == NULL || head->next == NULL )  
return head;  
Node \*p1 = head ;  
Node \*p2 = p1->next ;  
Node \*p3 = p2->next ;  
p1->next = NULL ;  
while ( p3 != NULL )  
{  
p2->next = p1 ;  
p1 = p2 ;  
p2 = p3 ;  
p3 = p3->next ;  
}  
p2->next = p1 ;  
head = p2 ;  
return head ;  
}

试题：

**67、华为面试题：怎么判断链表中是否有环？**

【参考答案】答：用两个指针来遍历这个单向链表，第一个指针p1，每次走一步；第二个指针p2，每次走两步；当p2 指针追上 p1的时候，就表明链表当中有环路了。

int testLinkRing(Link \*head)

{

Link \*t1=head,\*t2=head;

while( t1->next && t2->next)

    {

 t1 = t1->next;

if (NULL == (t2 = t2->next->next))

return 0;//无环

if (t1 == t2)

return 1;

}

return 0;

}

**试题68：编程实现合并两个有序(假定为降序)单链表的函数，输入为两个有序链表的头结点，函数返回合并后新的链表的头节点，**

**要求：不能另外开辟新的内存存放合并的链表。**

参考代码1：

node merge\_sorted\_list(const node head1,const node head2)

{

if((NULL == head1) && (NULL == head1))

{

return NULL;

}

else if(NULL == head1)

{

return head2;

}

else if(NULL == head2)

{

return head1;

}

else

{

node head = NULL,p1 = NULL,p2 = NULL;

if(head1->value >= head2->value)

{

head = head1;

p1 = head1->next;

p2 = head2;

}

else

{

head = head2;

p2 = head2->next;

p1 = head1;

}

node p = head;

while((NULL != p1) && (NULL != p2))

{

if(p1->value >= p2->value)

{

p->next = p1;

p = p1;

p1 = p1->next;

}

else

{

p->next = p2;

p = p2;

p2 = p2->next;

}

}

p->next = p1 ? p1 : p2;

return head;

}

}

采用递归的方法实现：

Node \* MergeRecursive(Node \*head1 , Node \*head2)

{

if((NULL == head1) && (NULL == head1))

{

return NULL;

}

if ( head1 == NULL )

return head2 ;

if ( head2 == NULL)

return head1 ;

Node \*head = NULL ;

if ( head1->value > head2->value )

{

head = head1 ;

head->next = MergeRecursive(head1->next,head2);

}

else

{

head = head2 ;

head->next = MergeRecursive(head1,head2->next);

}

return head ;

}

**网络相关**

**1.ISO的七层模型是什么？tcp/udp是属于哪一层？tcp/udp有何优缺点？**

分析：

　　应用层

　　表示层

　　会话层

　　运输层

　　网络层

　　物理链路层

　　物理层

　　tcp /udp属于运输层

　　TCP 服务提供了数据流传输、可靠性、有效流控制、全双工操作和多路复用技术等。

　　与 TCP 不同， UDP 并不提供对 IP 协议的可靠机制、流控制以及错误恢复功能等。由于 UDP 比较简单， UDP 头包含很少的字节，比 TCP 负载消耗少。

　　tcp: 提供稳定的传输服务，有流量控制，缺点是包头大，冗余性不好

　　udp: 不提供稳定的服务，包头小，开销小

**2：请问交换机和路由器分别的实现原理是什么？分别在哪个层次上面实现的？**

交换机用在局域网中，交换机通过纪录局域网内各节点机器的MAC地质(物理地址)就可以实现传递报文,无需看报文中的IP地质。路由器识别不同网络的方法是通过识别不同网络的网络ID号(IP地址的高端部分)进行的，所以为了保证路由成功，每个网络都必须有一个唯一的网络编号。路由器通过察看报文中IP地址，来决定路径，向那个子网(下一跳)路由，也就是说交换机工作在数据链路层看MAC地址，路由器工作在网际层看IP地质

但是由于现在网络设备的发展，很多设备既有交换机的功能有由路由器的功能(交换试路由器)使得两者界限越来越模糊。

**3.TCP/IP 建立连接的过程?(3-way shake)**

答：

在TCP/IP协议中，TCP协议提供可靠的连接服务，采用三次握手建立一个连接。

　　第一次握手：建立连接时，客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状

态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个

SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

　　第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)

，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。