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。

#error 命令是 C/C++语言的预处理命令之一，当预处理器预处理到#error 命令时将停止编译并输出用户自定义的错误消息。

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

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有函数访问，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。

上海某全球五百强面试题（嵌入式）

1.static 变量和 static 函数各有什么特点？

3.描述一下嵌入式基于 ROM 的运行方式基于 RAM 的运行方式有什么区别。

4.task 有几种状态？

5.task 有几种通讯方式？

6.C 函数允许重入吗？

7.嵌入式操作系统和通用操作系统有什么差别？

嵌入式软件工程师笔试题

1、将一个字符串逆序

2、将一个链表逆序

3、计算一个字节里（byte）里面有多少 bit 被置 1

4、搜索给定的字节(byte)

5、在一个字符串中找到可能的最长的子字符串

6、字符串转换为整数

7、整数转换为字符串

```
/*
* 题目：将一个字符串逆序
* 完成时间：2006.9.30 深圳极讯网吧
* 版权归刘志强所有
```

```
* 描述：写本程序的目的是希望练一下手，希望下午去面试能成功，不希望国庆节之后再去找工作拉！
*/
#include
using namespace std;
//#define NULL ((void *)0)
char * mystrrev(char * const dest,const char * const src)
{
if (dest==NULL && src==NULL)
   return NULL;
char *addr = dest;
int val_len = strlen(src);
dest[val_len] = '\0';
int i;
for (i=0; inext;
while(q!=NULL)
{
temp=q->next;
q->next=p;
p=q;
q=temp;
}
```

这样增加个辅助的指针就行乐。

ok 通过编译的代码：

```
#include

#include

#include

typedef struct List{

int data;

struct List *next;

}List;

List *list_create(void)

{

struct List *head,*tail,*p;

int e;

head=(List *)malloc(sizeof(List));

tail=head;

printf("\nList Create,input numbers(end of 0):");

scanf("%d",&e);

while(e){

p=(List *)malloc(sizeof(List));

p->data=e;

tail->next=p;

tail=p;

scanf("%d",&e);}

tail->next=NULL;

return head;
```

```
}
List *list_reverse(List *head)
{
List *p,*q,*r;
p=head;
q=p->next;
while(q!=NULL)
{
r=q->next;
q->next=p;
p=q;
q=r;
}
head->next=NULL;
head=p;
return head;
}
void main(void)
{
struct List *head,*p;
int d;
head=list_create();
printf("\n");
```

```
for(p=head->next;p;p=p->next)
printf("--%d--",p->data);
head=list_reverse(head);
printf("\n");
for(p=head;p->next;p=p->next)
printf("--%d--",p->data);
}
```

编写函数数 N 个 BYTE 的数据中有多少位是 1。

解：此题按步骤解：先定位到某一个 BYTE 数据；再计算其中有多少个 1。叠加得解。

```
#incluede
#define N 10
//定义 BYTE 类型别名
#ifndef BYTE
typedef unsigned char BYTE;
#endif
int comb(BYTE b[],int n)
{
int count=0;
int bi,bj;
BYTE cc=1,tt;
//历遍到第 bi 个 BYTE 数据
for(bi=0;bi>1;
tt=tt/2;
```

```
}
}
return count;
}
//测试
int main()
{
BYTE b[10]={3,3,3,11,1,1,1,1,1,1};
cout
```

1。编写一个 C 函数，该函数在一个字符串中找到可能的最长的子字符串，且该字符串是由同一字符组成的。

```
char * search(char *cpSource, char ch)
{
char *cpTemp=NULL, *cpDest=NULL;
int iTemp, iCount=0;
while(*cpSource)
{
if(*cpSource == ch)
{
iTemp = 0;
cpTemp = cpSource;
while(*cpSource == ch)
++iTemp, ++cpSource;
```

```
if(iTemp > iCount)

iCount = iTemp, cpDest = cpTemp;

if(!*cpSource)

break;

}

++cpSource;

}

return cpDest;

}

#include

#include

//

// 自定义函数 MyAtoI

// 实现整数字符串转换为证书输出

// 程序不检查字符串的正确性,请用户在调用前检查

//

int MyAtoI(char str[])

{

int i;

int weight = 1; // 权重

int rtn = 0; // 用作返回

for(i = strlen(str) - 1; i >= 0; i--)

{
```

```
    rtn += (str - '0')* weight; //
    weight *= 10; // 增重
}
return rtn;
}
void main()
{
char str[32];
printf("Input a string :");
gets(str);
printf("%d\n", MyAtoI(str));
}
#include
#include
void reverse(char s[])
{   //字符串反转
    int c, i=0, j;
    for(j=strlen(s)-1;i0);
    //如果是负数，补上负号
    if(sign
```

试题 1：请写一个 C 函数，若处理器是 Big_endian 的，则返回 0；若是 Little_endian 的，则返回 1

解答：

```
int checkCPU( )
{
    {
        union w
        {
            int    a;
            char b;
        } c;
        c.a = 1;
        return(c.b ==1);
    }
}
```

剖析：

嵌入式系统开发者应该对 Little-endian 和 Big-endian 模式非常了解。采用 Little-endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big-endian 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001
存放内容	0x12	0x34

32bit 宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在 Big-endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

联合体 union 的存放顺序是所有成员都从低地址开始存放，面试者的解答利用该特性，轻松地获得了 CPU 对内存采用 Little-endian 还是 Big-endian 模式读写。如果谁能当场给出这个解答，那简直就是一个天才的程序员。

2.

char str[] = “Hello” ;	void Func (char str[100])
char *p = str ;	{
int n = 10;	请计算
请计算	sizeof(str) = 4 （2 分）
sizeof (str) = 6 （2 分）	}
sizeof (p) = 4 （2 分）	void *p = malloc(100);
sizeof (n) = 4 （2 分）	请计算
	sizeof (p) = 4 （2 分）

3、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”？ （5 分）

答：C++语言支持函数重载，C 语言不支持函数重载。函数被 C++编译后在库中的名字与 C 语言的不同。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。

4.有关内存的思考题

void GetMemory(char *p)	char *GetMemory(void)
{	{

```
p = (char *)malloc(100);
}
void Test(void)
{
char *str = NULL;
GetMemory(str);
strcpy(str, "hello world");
printf(str);
}
```

请问运行 Test 函数会有什么样的结果？

答:程序崩溃。

因为 GetMemory 并不能传递动态内存，

Test 函数中的 str 一直都是 NULL。

strcpy(str, "hello world");将使程序崩溃。

```
char p[] = "hello world";
return p;
}
void Test(void)
{
char *str = NULL;
str = GetMemory();
printf(str);
}
```

请问运行 Test 函数会有什么样的结果？

答：可能是乱码。

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。

```
void GetMemory2(char **p, int num)
{
*p = (char *)malloc(num);
}
```

```
void Test(void)
{
char *str = (char *) malloc(100);
strcpy(str, “hello”);
```

```
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果？

答：

（1）能够输出 hello

（2）内存泄漏

```
free(str);
if(str != NULL)
{
strcpy(str, “world”);
printf(str);
}
}
```

请问运行 Test 函数会有什么样的结果？

答：篡改动态内存区的内容，后果难以预料，非常危险。

因为 free(str);之后，str 成为野指针，

if(str != NULL)语句不起作用。