得数组名又的确是个指针！
我们还可以发现数组名显得像指针的例子：

```
1. #include <string.h>
2. #include <iostream.h>
3. int main(int argc, char* argv[])
4. {
5.  char str1[10] = "I Love U";
6.  char str2[10];
7.  strcpy(str2,str1);
8.  cout << "string array 1: " << str1 << endl;
9.  cout << "string array 2: " << str2 << endl;
10. return 0;
11. }
```

标准 C 库函数 strcpy 的函数原形中能接纳的两个参数都为 char 型指针，而我们在调用中传给它的却是两个数组名！
函数输出：

```
string array 1: I Love U
string array 2: I Love U
```

数组名再一次显得像指针！
既然数组名不是指针，而为什么到处都把数组名当指针用？于是乎，许多程序员得出这样的结论：数组名（主）是（谓）不是指针的指针（宾）。
整个一魔鬼。
3. 数组名大揭密
那么，是揭露数组名本质的时候了，先给出三个结论：
(1)数组名的内涵在于其指代实体是一种数据结构，这种数据结构就是数组；
(2)数组名的外延在于其可以转换为指向其指代实体的指针，而且是一个指针常量；
(3)指向数组的指针则是另外一种变量类型（在 WIN32 平台下，长度为 4），仅仅意味着数组的存放地址！
3.1 数组名指代一种数据结构：数组
现在可以解释为什么第 1 个程序第 6 行的输出为 10 的问题，根据结论 1，数组名 str 的内涵为一种数据结构，即一个长度为 10 的 char 型数组，所以 sizeof(str)的结果为这个数据结构占据的内存大小：10 字节。
再看：

```
1. int intArray[10];
2. cout << sizeof(intArray) ;
```

第 2 行的输出结果为 40（整型数组占据的内存空间大小）。
如果 C/C++程序可以这样写：

```
1. int[10] intArray;
2. cout << sizeof(intArray) ;
```

我们就都明白了，intArray 定义为 int[10]这种数据结构的一个实例，可惜啊，C/C++目前并不支持这种定义方式。
3.2 数组名可作为指针常量
根据结论 2，数组名可以转换为指向其指代实体的指针，所以程序 1 中的第 5 行数组名直接赋值给指针，程序 2 第 7 行直接将数组名作为指针形参都可成立。
下面的程序成立吗？

```
1. int intArray[10];
2. intArray++;
```

读者可以编译之，发现编译出错。原因在于，虽然数组名可以转换为指向其指代实体的指针，但是它只能被看作一个

指针常量，不能被修改。
而指针，不管是指向结构体、数组还是基本数据类型的指针，都不包含原始数据结构的内涵，在 WIN32 平台下，sizeof 操作的结果都是 4。
顺便纠正一下许多程序员的另一个误解。**许多程序员以为 sizeof 是一个函数，而实际上，它是一个操作符**，不过其使用方式看起来的确太像一个函数了。语句 sizeof(int)就可以说明 sizeof 的确不是一个函数，因为函数接纳形参（一个变量），世界上没有一个 C/C++函数接纳一个数据类型（如 int）为“形参”。

3.3 数据名可能失去其数据结构内涵

到这里似乎数组名魔幻问题已经宣告圆满解决，但是平静的湖面上却再次掀起波浪。请看下面一段程序：

```
1. #include <iostream.h>
2. void arrayTest(char str[])
3. {
4.       cout << sizeof(str) << endl;
5. }
6. int main(int argc, char* argv[])
7. {
8.       char str1[10] = "I Love U";
9.       arrayTest(str1);
10.      return 0;
11. }
```

程序的输出结果为 4。不可能吧？
4，一个可怕的数字，前面已经提到其为指针的长度！
结论 1 指出，数据名内涵为数组这种数据结构，在 arrayTest 函数体内，str 是数组名，那为什么 sizeof 的结果却是指针的长度？这是因为：
(1)数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；
(2)很遗憾，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。
所以，数据名作为函数形参时，其全面沦落为一个普通指针！它的贵族身份被剥夺，成了一个地地道道的只拥有 4 个字节的平民。
以上就是结论 4。

4. 结论

本文以打破沙锅问到底的探索精神用数段程序实例论证了数据名和指针的区别。
最后，笔者再次表达深深的希望，愿我和我的同道中人能够真正以谨慎的研究态度来认真思考开发中的问题，这样才能在我们中间产生大师级的程序员，顶级的开发书籍。每次拿着美国鬼子的开发书籍，我们不免发出这样的感慨：我们落后太远了。

C/C++程序员应聘常见面试题深入剖析(1)

作者: 宋宝华 **e-mail:**21cnbao@21cn.com **出处:** 软件报

1.引言

本文的写作目的并不在于提供 C/C++程序员求职面试指导，而旨在从技术上分析面试题的内涵。文中的大多数面试题来自各大论坛，部分试题解答也参考了网友的意见。
许多面试题看似简单，却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 strcpy 函数都可看出面试者在技术上究竟达到了怎样的程度，我们能真正写好一个 strcpy 函数吗？我们都觉得自己能，可是我们写出的 strcpy 很可能只能拿到 10 分中的 2 分。读者可从本文看到 strcpy

函数从 2 分到 10 分解答的例子，看看自己属于什么样的层次。此外，还有一些面试题考查面试者敏捷的思维能力。
分析这些面试题，本身包含很强的趣味性；而作为一名研发人员，通过对这些面试题的深入剖析则可进一步增强自身的内功。

2. 找错题

试题 1：

```
void test1()
{
   char string[10];
   char* str1 = "0123456789";
    strcpy( string, str1 );
}
```

试题 2：

```
void test2()
{
   char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

试题 3：

```
void test3(char* str1)
{
   char string[10];
   if( strlen( str1 ) <= 10 )
    {
       strcpy( string, str1 );
    }
}
```

解答：

试题 1 字符串 str1 需要 11 个字节才能存放下（包括末尾的'\0'），而 string 只有 10 个字节的空间，strcpy 会导致数组越界；
对试题 2，如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string, str1)调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；
对试题 3，if(strlen(str1) <= 10)应改为 if(strlen(str1) < 10)，因为 strlen 的结果未统计'\0'所占用的 1 个字节。

剖析：

考查对基本功的掌握：
（1）字符串以'\0'结尾；

（2）对数组越界把握的敏感度；
（3）库函数 strcpy 的工作方式，如果编写一个标准 strcpy 函数的总分值为 10，下面给出几个不同得分的答案：

2 分

```
void strcpy( char *strDest, char *strSrc )
{
   while( (*strDest++ = * strSrc++) != ‘\0’ );
}
```

4 分

```
void strcpy( char *strDest, const char *strSrc )
//将源字符串加 const，表明其为输入参数，加 2 分
{
   while( (*strDest++ = * strSrc++) != ‘\0’ );
}
```

7 分

```
void strcpy(char *strDest, const char *strSrc)
{
//对源地址和目的地址加非 0 断言，加 3 分
  assert( (strDest != NULL) && (strSrc != NULL) );
  while( (*strDest++ = * strSrc++)  !=  ‘\0’ );
}
```

10 分

```
//为了实现链式操作，将目的地址返回，加 3 分！
char * strcpy( char *strDest, const char *strSrc )
{
  assert( (strDest != NULL) && (strSrc != NULL) );
  char *address = strDest;
  while( (*strDest++ = * strSrc++) != ‘\0’ );
  return address;
}
```

从 2 分到 10 分的几个答案我们可以清楚的看到，小小的 strcpy 竟然暗藏着这么多玄机，真不是盖的！需要多么扎实的基本功才能写一个完美的 strcpy 啊！

（4）对 strlen 的掌握，它没有包括字符串末尾的'\0'。

读者看了不同分值的 strcpy 版本，应该也可以写出一个 10 分的 strlen 函数了，完美的版本为：

```
int strlen( const char *str )    //输入参数 const
{
   assert( strt != NULL );    //断言字符串地址非 0
   int len;
   while( (*str++) != '\0' )
   {
      len++;
   }
   return len;
}
```

试题 4：

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

试题 5：

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

试题 6：

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

试题 7：

```
void Test( void )
{
    char *str = (char *) malloc( 100 );
    strcpy( str, "hello" );
    free( str );
    ...  //省略的其它语句
}
```

解答：

试题 4 传入中 GetMemory(char *p)函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的值，执行完

```
char *str = NULL;
GetMemory( str );
```

后的 str 仍然为 NULL；

试题 5 中

```
    char p[] = "hello world";
    return p;
```

的 p[]数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

试题 6 的 GetMemory 避免了试题 4 的问题，传入 GetMemory 的参数为字符串指针的指针，但是在 GetMemory 中执行申请内存及赋值语句

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

后未判断内存是否申请成功，应加上：

```
if ( *p == NULL )
 {
   ...//进行申请内存失败处理
  }
```

试题 7 存在与试题 6 同样的问题，在执行

```
char *str = (char *) malloc(100);
```

后未进行内存是否申请成功的判断；另外，在 free(str)后未置 str 为空，导致可能变成一个“野”指针，应加上：

```
str = NULL;
```

试题 6 的 Test 函数中也未对 malloc 的内存进行释放。

剖析：

试题 4～7 考查面试者对内存操作的理解程度，基本功扎实的面试者一般都能正确的回答其中 50~60 的错误。但是要完全解答正确，却也绝非易事。

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。

在看看下面的一段程序有什么错误：

```
swap( int* p1,int* p2 )
{
    int *p;
    *p = *p1;
    *p1 = *p2;
    *p2 = *p;
}
```

在 swap 函数中，p 是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在 VC++中 DEBUG 运行时提示错误“Access Violation”。该程序应该改为：

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

```
    int p;
    p = *p1;
    *p1 = *p2;
    *p2 = p;
}
```

C/C++程序员应聘常见面试题深入剖析(2)

作者：宋宝华　**e-mail**:21cnbao@21cn.com　　**出处**：软件报

3.内功题

试题 1：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）

解答：

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)

剖析：

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，得 0 分。

试题 2：以下为 Windows NT 下的 32 位 C++程序，请计算 sizeof 的值

```
void Func ( char str[100] )
{
    sizeof( str ) = ?
}
void *p = malloc( 100 );
sizeof ( p ) = ?
```

解答：

```
sizeof( str ) = 4
sizeof ( p ) = 4
```

剖析：

Func (char str[100])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。

数组名的本质如下：

（1）数组名指代一种数据结构，这种数据结构就是数组；
例如：
```
char str[10];
cout << sizeof(str) << endl;
```
输出结果为 10，str 指代数据结构 char[10]。
（2）数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；
```
char str[10];
str++;   //编译出错，提示 str 不是左值
```
（3）数组名作为函数形参时，沦为普通指针。
Windows NT 32 位平台下，指针的长度（占用内存的大小）为 4 字节，故 sizeof(str) 、sizeof (p) 都为 4。
试题 3：写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事？
```
least = MIN(*p++, b);
```
解答：
```
#define MIN(A,B)    ((A) <=  (B) ? (A) : (B))
```
MIN(*p++, b)会产生宏的副作用
剖析：
这个面试题主要考查面试者对宏定义的使用，宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。
程序员对宏定义的使用要非常小心，特别要注意两个问题：
（1）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：
```
#define MIN(A,B)    (A) <=  (B) ? (A) : (B)
#define MIN(A,B)    (A <=  B ? A : B )
```
都应判 0 分；
（2）防止宏的副作用。
宏定义#define MIN(A,B)   ((A) <= (B) ? (A) : (B))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。
除此之外，另一个应该判 0 分的解答是：
```
#define MIN(A,B)    ((A) <=  (B) ? (A) : (B));
```
这个解答在宏定义的后面加“;”，显示编写者对宏的概念模糊不清，只能被无情地判 0 分并被面试官淘汰。
试题 4：为什么标准头文件都有类似以下的结构？
```
  #ifndef __INCvxWorksh
  #define __INCvxWorksh
  #ifdef __cplusplus
  extern "C" {
  #endif
  /*...*/
  #ifdef __cplusplus
  }
```

```
#endif
#endif /* __INCvxWorksh */
```

解答：

头文件中的编译宏

```
#ifndef  __INCvxWorksh
#define  __INCvxWorksh
#endif
```

的作用是防止被重复引用。

作为一种面向对象的语言，C++支持函数重载，而过程式语言 C 则不支持。函数被 C++编译后在 symbol 库中的名字与 C 语言的不同。例如，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 symbol 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。_foo_int_int 这样的名字包含了函数名和函数参数数量及类型信息，C++就是考这种机制来实现函数重载的。

为了实现 C 和 C++的混合编程，C++提供了 C 连接交换指定符号 extern "C"来解决名字匹配问题，函数声明前加上 extern "C"后，则编译器就会按照 C 语言的方式将该函数编译为_foo，这样 C 语言中就可以调用 C++的函数了。

试题 5：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”

函数头是这样的：

```
//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 )
{
//请填充...
}
```

解答：

正确解答 1：

```
void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    strcpy ( tmp, pStr + n );
    strcpy ( tmp + steps, pStr);
    *( tmp + strlen ( pStr ) ) = '\0';
    strcpy( pStr, tmp );
}
```

正确解答 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

试题 6：已知 WAV 文件格式如下表，打开一个 WAV 文件，以适当的数据结构组织 WAV 文件头并解析 WAV 格式的各项信息。

WAVE 文件格式说明表

	偏移地址	字节数	数据类型	内　　容
文件头	00H	4	Char	"RIFF"标志
	04H	4	int32	文件长度
	08H	4	Char	"WAVE"标志
	0CH	4	Char	"fmt"标志
	10H	4		过渡字节（不定）
	14H	2	int16	格式类别
	16H	2	int16	通道数
	18H	2	int16	采样率（每秒样本数），表示每个通道的播放速度
	1CH	4	int32	波形音频数据传送速率
	20H	2	int16	数据块的调整数（按字节算的）
	22H	2		每样本的数据位数
	24H	4	Char	数据标记符 " data "
	28H	4	int32	语音数据的长度

解答：

将 WAV 文件格式定义为结构体 WAVEFORMAT：

```
typedef struct tagWaveFormat
{
        char cRiffFlag[4];
        UIN32 nFileLen;
        char cWaveFlag[4];
        char cFmtFlag[4];
        char cTransition[4];
        UIN16 nFormatTag ;
        UIN16 nChannels;
        UIN16 nSamplesPerSec;
        UIN32 nAvgBytesperSec;
        UIN16 nBlockAlign;
        UIN16 nBitNumPerSample;
        char cDataFlag[4];
```

```
        UIN16 nAudioLength;
} WAVEFORMAT;
```
假设 WAV 文件内容读出后存放在指针 buffer 开始的内存单元内，则分析文件格式的代码很简单，为：
```
WAVEFORMAT  waveFormat;
memcpy( &waveFormat, buffer,sizeof( WAVEFORMAT ) );
```
直接通过访问 waveFormat 的成员，就可以获得特定 WAV 文件的各项格式信息。

剖析：

试题 6 考查面试者组织数据结构的能力，有经验的程序设计者将属于一个整体的数据成员组织为一个结构体，利用指针类型转换，可以将 memcpy、memset 等函数直接用于结构体地址，进行结构体的整体操作。

透过这个题可以看出面试者的程序设计经验是否丰富。

试题 7：编写类 String 的构造函数、析构函数和赋值函数，已知类 String 的原型为：
```
class String
{
public:
    String(const char *str = NULL); // 普通构造函数
    String(const String &other); // 拷贝构造函数
    ~ String(void); // 析构函数
    String & operate =(const String &other); // 赋值函数
private:
    char *m_data; // 用于保存字符串
};
```
解答：
```
//普通构造函数
String::String(const char *str)
{
        if(str==NULL)
        {
            m_data = new char[1]; // 得分点：对空字符串自动申请存放结束标志'\0'的空
                                  //加分点：对 m_data 加 NULL 判断
            *m_data = '\0';
    }
        else
        {
        int length = strlen(str);
        m_data = new char[length+1]; // 若能加 NULL 判断则更好
        strcpy(m_data, str);
        }
}
// String 的析构函数
String::~String(void)
{
    delete [] m_data; // 或 delete m_data;
}
```

```
//拷贝构造函数
String::String(const String &other)        // 得分点：输入参数为 const 型
{
        int length = strlen(other.m_data);
        m_data = new char[length+1];          //加分点：对 m_data 加 NULL 判断
        strcpy(m_data, other.m_data);
}
//赋值函数
String & String::operate =(const String &other) // 得分点：输入参数为 const 型
{
        if(this == &other)                //得分点：检查自赋值
            return *this;
        delete [] m_data;              //得分点：释放原有的内存资源
        int length = strlen( other.m_data );
        m_data = new char[length+1];    //加分点：对 m_data 加 NULL 判断
        strcpy( m_data, other.m_data );
        return *this;                   //得分点：返回本对象的引用
}
```

剖析：

能够准确无误地编写出 String 类的构造函数、拷贝构造函数、赋值函数和析构函数的面试者至少已经具备了 C++基本功的 60%以上！

在这个类中包括了指针类成员变量 m_data，当类中包括指针类成员变量时，一定要重载其拷贝构造函数、赋值函数和析构函数，这既是对 C++程序员的基本要求，也是《Effective C++》中特别强调的条款。

仔细学习这个类，特别注意加注释的得分点和加分点的意义，这样就具备了 60%以上的 C++基本功！

试题 8：请说出 static 和 const 关键字尽可能多的作用

解答：

static 关键字至少有下列 n 个作用：

（1）函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；

（2）在模块内的 static 全局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；

（3）在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；

（4）在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

（5）在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

const 关键字至少有下列 n 个作用：

（1）欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；

（2）对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const；

（3）在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；

（4）对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量；

（5）对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。例

如：

```
const classA operator*(const classA& a1,const classA& a2);
```

operator*的返回结果必须是一个 const 对象。如果不是，这样的变态代码也不会编译出错：

```
classA a, b, c;
(a * b) = c; // 对 a*b 的结果赋值
```

操作(a * b) = c 显然不符合编程者的初衷，也没有任何意义。

剖析：

惊讶吗？小小的 static 和 const 居然有这么多功能，我们能回答几个？如果只能回答 1~2 个，那还真得闭关再好好修炼修炼。

这个题可以考查面试者对程序设计知识的掌握程度是初级、中级还是比较深入，没有一定的知识广度和深度，不可能对这个问题给出全面的解答。大多数人只能回答出 static 和 const 关键字的部分功能。

4. 技巧题

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	0x4000	0x4001	0x4002	0x4003

址				
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。

试题 2：写一个函数返回 1+2+3+…+n 的值（假定结果不会超过长整型变量的范围）

解答：

```
int Sum( int n )
{
    return  ( (long)1 + n) * n / 2;    //或 return  (1l + n) * n / 2;
}
```

剖析：

对于这个题，只能说，也许最简单的答案就是最好的答案。下面的解答，或者基于下面的解答思路去优化，不管怎么“折腾”，其效率也不可能与直接 return (1l + n) * n / 2 相比！

```
int Sum( int n )
{
  long sum = 0;
  for( int i=1; i<=n; i++ )
  {
     sum += i;
  }
  return sum;
}
```

所以程序员们需要敏感地将数学等知识用在程序设计中。

一道著名外企面试题的抽丝剥茧

宋宝华 21cnbao@21cn.com 软件报

问题：对于一个字节（8bit）的数据，求其中“1”的个数，要求算法的执行效率尽可能地高。

分析：作为一道著名外企的面试题，看似简单，实则可以看出一个程序员的基本功底的扎实程度。你或许已经想到很多方法，譬如除、余操作，位操作等，但都不是最快的。本文一步步分析，直到最后给出一个最快的方法，相信你看到本文最后的那个最快的方法时会有惊诧的感觉。

解答：

首先，很自然的，你想到除法和求余运算，并给出了如下的答案：

方法 1：使用除、余操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  for (i = 0; i < 8; i++)
  {
    if (a % 2 == 1)
    {
      num++;
    }
    a = a / 2;
  }
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

很遗憾，众所周知，除法操作的运算速率实在是很低的，这个答案只能意味着面试者被淘汰！

好，精明的面试者想到了以位操作代替除法和求余操作，并给出如下答案：

方法 2：使用位操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  for (i = 0; i < 8; i++)
  {
    num += (a >> i) &0x01;
  }
  /*或者这样计算 1 的个数：*/
  /* for(i=0;i<8;i++)
  {
  if((a>>i)&0x01)
  num++;
  }
```

```
   */
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

方法二中 num += (a >> i) &0x01;操作的执行效率明显高于方法一中的

```
if (a % 2 == 1)
    {
      num++;
    }
a = a / 2;
```

到这个时候，面试者有被录用的可能性了，但是，难道最快的就是这个方法了吗？没有更快的了吗？方法二真的高山仰止了吗？

能不能不用做除法、位操作就直接得出答案的呢？于是你想到把 0~255 的情况都罗列出来，并使用分支操作，给出如下答案：

方法 3：使用分支操作

```
#include
#define BYTE unsigned char
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  switch (a)
  {
    case 0x0:
      num = 0;
      break;
    case 0x1:
    case 0x2:
    case 0x4:
    case 0x8:
    case 0x10:
    case 0x20:
    case 0x40:
    case 0x80:
      num = 1;
      break;
    case 0x3:
    case 0x6:
    case 0xc:
    case 0x18:
```

```
    case 0x30:
    case 0x60:
    case 0xc0:
      num = 2;
      break;
      //...
  }
  printf("\nthe num of 1 in the BYTE is %d", num);
  return 0;
}
```

方法三看似很直接，实际执行效率可能还会小于方法二，因为分支语句的执行情况要看具体字节的值，如果 a=0，那自然在第 1 个 case 就得出了答案，但是如果 a=255，则要在最后一个 case 才得出答案，即在进行了 255 次比较操作之后！

看来方法三不可取！但是方法三提供了一个思路，就是罗列并直接给出值，离最后的方法四只有一步之遥。眼看着就要被这家著名外企录用，此时此刻，绝不对放弃寻找更快的方法。

终于，灵感一现，得到方法四，一个令你心潮澎湃的答案，快地令人咋舌，算法中不需要进行任何的运算。你有足够的信心了，把下面的答案递给面试官：

方法 4：直接得到结果

```
#include
#define BYTE unsigned char
/* 定义查找表 */
BYTE numTable[256] =
{
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3,
    3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3,
    4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4,
    3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3,
    4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6,
    6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4,
    5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3,
    4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4,
    4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
    7, 6, 7, 7, 8
};
int main(int argc, char *argv[])
{
  int i, num = 0;
  BYTE a = 0;
  /* 接收用户输入 */
  printf("\nPlease Input a BYTE(0~255):");
  scanf("%d", &a);
  /* 计算 1 的个数 */
  /* 用 BYTE 直接作为数组的下标取出 1 的个数，妙哉！ */
```

```
  printf("\nthe num of 1 in the BYTE is %d", checknum[a]);
  return 0;
}
```

这是个典型的空间换时间算法，把 0~255 中 1 的个数直接存储在数组中，字节 a 作为数组的下标，checknum[a] 直接就是 a 中“1”的个数！算法的复杂度如下：
时间复杂度：O(1)
空间复杂度：O(2n)
恭喜你，你已经被这家著名的外企录用！老总向你伸出手，说：“Welcome to our company”。

C/C++结构体的一个高级特性－－指定成员的位数

宋宝华 21cnbao@21cn.com sweek
在大多数情况下，我们一般这样定义结构体：

```
struct student
{
    unsigned int sex;
    unsigned int age;
};
```

对于一般的应用，这已经能很充分地实现数据了的“封装”。
但是，在实际工程中，往往碰到这样的情况：那就是要用一个基本类型变量中的不同的位表示不同的含义。譬如一个 cpu 内部的标志寄存器，假设为 16 bit，而每个 bit 都可以表达不同的含义，有的表示结果是否为 0，有的表示是否越界等等。这个时候我们用什么数据结构来表达这个寄存器呢？
答案还是结构体！
为达到此目的，我们要用到结构体的高级特性，那就是在基本成员变量的后面添加：

```
: 数据位数
```

组成新的结构体：

```
struct xxx
{
    成员 1 类型成员 1 : 成员 1 位数;
    成员 2 类型成员 2 : 成员 2 位数;
    成员 3 类型成员 3 : 成员 3 位数;
};
```

基本的成员变量就会被拆分！这个语法在初级编程中很少用到，但是在高级程序设计中不断地被用到！
例如：

```
struct student
{
    unsigned int sex : 1;
    unsigned int age : 15;
};
```

上述结构体中的两个成员 sex 和 age 加起来只占用了一个 unsigned int 的空间（假设 unsigned int 为 16 位）。
基本成员变量被拆分后，访问的方法仍然和访问没有拆分的情况是一样的，例如：

```
struct student sweek;
sweek.sex = MALE;
sweek.age = 20;
```
虽然拆分基本成员变量在语法上是得到支持的，但是并不等于我们想怎么分就怎么分，例如下面的拆分显然是不合理的：
```
struct student
{
    unsigned int sex : 1;
    unsigned int age : 12;
};
```
这是因为 1+12 = 13，不能再组合成一个基本成员，不能组合成 char、int 或任何类型，这显然是不能"自圆其说"的。

在拆分基本成员变量的情况下，我们要特别注意数据的存放顺序，这还与 CPU 是 Big endian 还是 Little endian 来决定。Little endian 和 Big endian 是 CPU 存放数据的两种不同顺序。对于整型、长整型等数据类型，Big endian 认为第一个字节是最高位字节（按照从低地址到高地址的顺序存放数据的高位字节到低位字节）；而 Little endian 则相反，它认为第一个字节是最低位字节（按照从低地址到高地址的顺序存放数据的低位字节到高位字节）。

我们定义 IP 包头结构体为：
```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
            version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
          ihl:4;
#else
#error    "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __u16     tot_len;
    __u16     id;
    __u16     frag_off;
    __u8    ttl;
    __u8    protocol;
    __u16     check;
    __u32     saddr;
    __u32     daddr;
    /*The options start here. */
};
```
在 Little endian 模式下，iphdr 中定义：
```
    __u8    ihl:4,
            version:4;
```
其存放方式为：

第 1 字节低 4 位 ihl

第 1 字节高 4 位 version （IP 的版本号）
若在 Big endian 模式下还这样定义，则存放方式为：
第 1 字节低 4 位 version （IP 的版本号）
第 1 字节高 4 位 ihl
这与实际的 IP 协议是不匹配的，所以在 Linux 内核源代码中，IP 包头结构体的定义利用了宏：

```
#if defined(__LITTLE_ENDIAN_BITFIELD)
...
#elif defined (__BIG_ENDIAN_BITFIELD)
...
#endif
```

来区分两种不同的情况。
由此我们总结全文的主要观点：
（1）C/C++语言的结构体支持对其中的基本成员变量按位拆分；
（2）拆分的位数应该是合乎逻辑的，应仍然可以组合为基本成员变量；
要特别注意拆分后的数据的存放顺序，这一点要结合具体的 CPU 的结构。

C/C++中的近指令、远指针和巨指针

宋宝华 email:21cnbao@21cn.com sweek
在我们的 C/C++学习生涯中、在我们大脑的印象里，通常只有指针的概念，很少听说指针还有远、近、巨之分的，从没听说过什么近指针、远指针和巨指针。
可以，某年某月的某一天，你突然看到这样的语句：

```
char near *p; /*定义一个字符型"近"指针*/
char far *p; /*定义一个字符型"远"指针*/
char huge *p; /*定义一个字符型"巨"指针*/
```

实在不知道语句中的"near"、"far"、"huge"是从哪里冒出来的，是个什么概念！本文试图对此进行解答，解除许多人的困惑。

这一点首先要从 8086 处理器体系结构和汇编渊源讲起。大家知道，8086 是一个 16 位处理器，它设定了四个段寄存器，专门用来保存段地址：CS（Code Segment）：代码段寄存器；DS（Data Segment）：数据段寄存器；SS（Stack Segment）：堆栈段寄存器；ES（Extra Segment）：附加段寄存器。8086 采用段式访问，访问本段（64K 范围内）的数据或指令时，不需要变更段地址（意味着段地址寄存器不需修改），而访问本段范围以外的数据或指令时，则需要变更段地址（意味着段地址寄存器需要修改）。

因此，在 16 位处理器环境下，如果访问本段内地址的值，用一个 16 位的指针（表示段内偏移）就可以访问到；而要访问本段以外地址的值，则需要用 16 位的段内偏移+16 位的段地址，总共 32 位的指针。

这样，我们就知道了远、近指针的区别：

- 近指针是只能访问本段、只包含本段偏移的、位宽为 16 位的指针；
- 远指针是能访问非本段、包含段偏移和段地址的、位宽为 32 位的指针。

近指针只能对 64k 字节数据段内的地址进行存取，如：

```
char near *p;
p=(char near *)0xffff;
```

远指针是 32 位指针，它表示段地址：偏移地址，远指针可以进行跨段寻址，可以访问整个内存的地址。如定