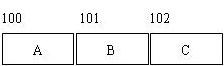
**一．指针到底是什么**

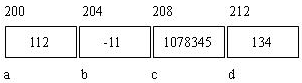
指针（pointer）到底是什么，弄清楚这个问题很重要，这是我们所讨论的话题的源头，而在阐述指针是什么之前，我们需要先来看一下变量的概念。

我们知道，计算机的内存（primary storage）被划分为多个存储单元，这些存储单元可以以单个或者顺序相连组成一个更大单元的方式被使用。每一个单独的存储单元都是一个字节（byte），它通常由8个位（bit）组成，每一个位可以表示的值只有0或1。每一个存储单元都被一个及其分配的标识唯一地表示，而这个标识就是地址。

下图表示了存储单元单独被操作时的情形，矩形表示存储单元，矩形内的内容是存储在这个内存单元的具体的值。矩形上方的数就是每个内存单元的地址。因为每个单元为一个字节，而每个字符型常量（character constant）所占据的正是一个字节，如下所示：



再来看下面的图：

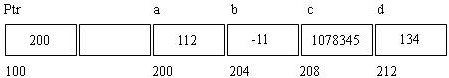


这次的情况是顺序连成组进行操作，对于整型常量（integer constant），在32位计算机中需要四个字节来存储（有一点要声明，208位置的那个矩形里的1078345超出了int类型的范围，是long int类型，但ANSI C只规定了long型数据长度不小于int型，int型数据长度不小于short型，并规定int型为16位，long型为32位，然而很多编译器采取的策略是使long和int型数据占据相同的内存字节数，即全为32位），所以地址以4个单位增长(也就是说现在的一个矩形表示4个内存单元)，这次矩形下面多了几个小写字母，存储在矩形里面的值不是固定唯一的，而是可变的。我们可以把矩形认为是一个变量（variable），每次我们要引用矩形里的值时，机器都是通过地址来定位(那个矩形)并取得其中的值的，而对于我们来说要记住这些地址几乎是不可能的，所以高级语言提供了用名字来访问内存位置的特性，它们就是变量名，即上图的a，b，c，d。

现在用变量名替换掉上图中的地址：

http://p.blog.csdn.net/images/p_blog_csdn_net/porscheyin/EntryImages/20081206/1-3.gif

大家要注意，变量名与地址的关联是由编译器为我们实现的，具体的实现方式我们无需关心，但要清楚硬件仍然是通过地址访问内存位置的。接下来，继续看图：



来看新增的Ptr，同样是个变量，它也有地址，它的值是变量a的地址。至此可以给出指针的定义了：**指针是一种用于存放另一个变量的地址的变量。**上图中的Ptr就是一个指针，并且我们说它指向了变量a（因为Ptr的值是变量a的地址），要注意指针中只能存放地址，不能将一个整型量或者其他非地址类型（整型数0及具有0值的整形常量表达式除外，后面的文章会细致讲解）的数据赋给一个指针！

还有，指针这个词，由于是对pointer这个词翻译得来，完整的叫法应该是指针变量，由于指针变量中存的地址，而在很多英文资料中，指针(pointer)的涵义不完全都是指针变量，有时也指地址，请大家在阅读或参考资料时注意区分！

**（注：本章及后续章节中用以表示内存的矩形中的值及其地址只是为了简洁易读从而采用十进制数表示。）**

**二．指针的定义及运算**

上一讲已经说过，指针是一种变量，它也有自己的地址，但由于它是专门用来存放地址的变量，所以把它认为是种特殊的变量，既然有着特殊的身份，那么也理应受到特殊的待遇，下面来看看它享受了那些优待。

**1．指针的定义**

在C语言中，定义一个普通的变量（如整型数），我们这样做：int i; 而定义一个指针变量（指针）我们需要这样做：int \*p ;  还记得吗，一个矩形中的值是有类型的，可能是整型，可能是字符型……，它们原本是“清白”的，无类型的，是我们通过一些手段使它们有了类型。当我们做出int i; 这样一个定义时，编译器就会分配一个地址(例如200)并和i 关联起来，而int将限定编译器把这个区域中的内容作为整型数看待。

**矩形内的值被视为int型**

现在我们又有了int \*p;这个定义，假设p是指向变量i的（见下图），p中存的是变量i的地址。\* 表示p是一个指针，而int表示p中所存的地址对应的变量（即变量i）的类型是int。

**p指向i , int \*p；中的int是指针p所指向的变量的类型**

我们将int称为指针p的基类型，或指针p所指向的变量的类型。

类似地，我们可以有： char  \*s ;   （ s是指向char型变量的指针 ）

  float  \*f ;    （ f是指向float型变量的指针 ）

  double  \*d ;  （ d是指向double型变量的指针 ）

由此得到声明一个指针变量（指针）的一般形式 ： 基类型  \* 指针名；

有一点要注意，在定义指针时，以下两种方式都是允许的，例如：

int  \*ptr1;

int\*  ptr2;

但一般比较倾向用第一种，因为可以避免以下的误解：

int\*  prt1,   ptr2;

这样的定义方式，容易使人误以为ptr2也是一个指针，事实上并不是，prt2是一个int型变量，以下的定义方式中ptr1与ptr2才都是指针：

int\* ptr1,  \*ptr2;

**2.指针的运算**

**<1>．&（address-of operator）取地址操作符：**

究竟如何使一个指针指向一个变量呢？后面的语句给出了解答：int \*p = &i；& 用于取一个对象的地址**（本文说的对象是泛指的某一事物，如变量，数组等，和C++中的对象概念不同）**，这里用于将i的地址赋给p, 那么指针p就指向了变量i 。上述的语句也可以分开写，如：int \*p;  p = &i;

**小扩展：**（下面大括号中的内容，出涉指针的朋友可以跳过，当然也可以作为扩展知识）

**{**&的实质：当对一个T类型对象进行 & 操作时，返回的是一个“指向T的指针”类型的常量，即指针常量（pointer constant），在我们使用&运算符时我们并不关心它是如何实现的，因为有编译器帮我们隐藏了这些细节。

可当我们想要对一个指针赋一个绝对地址的时候，这个问题就体现出来了，而且我们不得不去关注，在C语言中没有一种内建（built-in）的方法去表示指针常量，所以当我们使用它的时候通常先写成整型常量的形式，然后再通过强制类型转换把它转换成相应的类型，如：int \* , double \* , char \*等。 所以后面所示的做法是不行的： int \*p = 0x12345678 ; 正确的方式应为：int \*p = (int \*) 0x12345678; 也许大家还记得我在第一讲中说的要注意指针中只能存放地址，不能将一个非0值整型常量表达式或者其他非地址类型的数据赋给一个指针，原因就在此。在大多数计算机中，内存地址确实是以无符号整型数来表示的，而且多以16进制表示，但我们在C语言中不能用整型数去表示地址，只能用指针常量来表示，因为它是被用来赋给一个指针的。

对于这个赋值问题还可以换一个角度去理解，在C语言中，使用赋值操作符时，赋值操作符左边和右边的表达式类型应该是相同的，如果不是，赋值操作符将试图把右边表达式的值转换为左边的类型。所以如果写出int \*p = 0x12345678 ; 这条语句编译器会报错：'=' : cannot convert from ' const int ' to ' int\* ' ，因为赋值操作符左边和右边的表达式的类型应该相同，而0x12345678是int型常量，p是一个指向int型的指针，两者类型不同，所以正确的方式是：int \*p = (int \*) 0x12345678 ; **}**

**<2>．\* (Dereference operator) 解引用操作符**

\* 在定义时用来说明一个变量是指针，而在定义了一个指针之后，我们使用（引用）指针时，\*p表示的是p所指向的对象（即i）。也就是说，对于一个已定义的指针使用 \* 操作符，将访问这个指针所指向的对象，我们来看下面的程序：

#include<stdio.h>

int main()

{

int i;                    /\*定义一个int型变量i\*/

int \*p;                  /\* 定义一个指向int类型的指针p \*/

i = 2 ;                 /\* 初始化i为2 \*/

p = &i ;               /\* 将i的地址赋给p ，即使p指向i \*/

printf("%d\n", i ) ;       /\*输出i的值 \*/

printf("%d\n", \*p ) ;     /\* 输出p所指向的存储单元的值，即i的值\*/

return 0 ;            /\* 标准C语言主函数应返回一个值，用以通知操作系统程序执行成功与否，通常0表示成功\*/

}

程序输出结果为：

2

2

对于 \* 操作符，由于它有两个等价的术语dereference和indirection ，所以在国内的书籍中你会看到各种翻译方法，如：解引用、解除引用、反引用、反向引用、间接引用、间接访问……

只要你知道它是用来访问一个指针所指向的对象的，那么不管它叫什么都不重要了。还是那句话，弄懂是什么，不要在乎叫什么，如果你理解了它的真正含义，大可以简洁地称它为“星号”操作符！

**3.指针的初始化**

ANSI C定义了零指针常量的概念：一个具有0值的整形常量表达式，或者此类表达式被强制转换为void \*类型，则称为空指针常量，它可以用来初始化或赋给任何类型的指针。也就是说，我们可以将0、0L、'/0'、2–2、0\*5以及(void\*)0赋给一个任何类型的指针，此后这个指针就成为一个空指针，由系统保证空指针不指向任何对象或函数。

ANSI C还定义了一个宏**NULL**，用来表示空指针常量。大多数C语言的实现中NULL是采用后面这种方式定义的：#define  NULL  ((void\*)0)。

对指针进行初始化时常用的有以下几种方式：

  1.采用NULL或空指针常量，如：int \*p = NULL;或 char \*p = 2-2; 或float\*p = 0;

  2.取一个对象的地址然后赋给一个指针，如：int i = 3;  int \*ip = &i；

  3.将一个指针常量赋给一个指针，如：long \*p = (long \*)0xfffffff0;

  4.将一个T类型数组的名字赋给一个相同类型的指针，如：char ary[100]; char \*cp = ary;

  5.将一个指针的地址赋给一个指针，如：int i = 3;  int \*ip = &i；int \*\*pp = &ip;

  6.将一个字符串常量赋给一个字符指针，如：char \*cp = “abcdefg”;

对指针进行初始化或赋值的实质是将**一**个**地址或同类型(或相兼容的类型)的指针**赋给它，而不管这个地址是怎么取得的。要注意的是：**对于一个不确定要指向何种类型的指针，在定义它之后最好把它初始化为NULL，并在解引用这个指针时对它进行检验，防止解引用空指针。另外，为程序中任何新创建的变量提供一个合法的初始值是一个好习惯，它可以帮你避免一些不必要的麻烦。**

**4．void \*型指针**

ANSI C定义了一种void \*型指针，表示定义一个指针，但不指定它指向何种类型的数据。void \*型指针作为一种通用的指针，可以和其它任何类型的指针(函数指针除外)相互转化而不需要类型强制转换，但不能对它进行解引用及下标操作。C语言中的malloc函数的返回值就是一个void \*型指针，我们可以把它直接赋给一个其他类型的指针，但从安全的编程风格角度以及兼容性上讲，最好还是将返回的指针强制转换为所需的类型，另外，malloc在无法满足请求时会通过返回一个空指针来作为“内存分配失败”的信号，所以要注意返回值指针的判空。

**5．指向指针的指针**

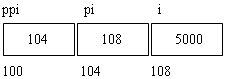
在指针初始化的第5种方式中提到了用一个指针的地址来初始化一个指针。回忆一下上一讲的内容：指针是一种变量，它也有自己的地址，所以它本身也是可用指针指向的对象。我们可以将指针的地址存放在另一个指针中，如：

int i = 5000;

int \*pi = &i;

int \*\*ppi =&pi;

此时的ppi即是一个指向指针的指针,下图表示了这些对象：



i的地址为108，pi的内容就是i的地址，而pi的地址为104，ppi的内容即是pi的地址。对ppi解引用照常会得到ppi所指的对象，所获得的对象是指向int型变量的指针pi。想要真正地访问到i.，必须对ppi进行两次解引用，如下面代码所示：

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

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

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

以上三条语句的输出均为5000。

三．指针与数组的“爱恨情仇”

本将中指针的算术运算本应放在第二讲中，但考虑到它与数组关系密切故将其纳入本讲。

1.指针的算术运算

在上一讲指针初始化的第4种方式中提到了可以将一个T类型数组的名字赋给一个相同类型的指针，这说明指针可以和数组发生联系，在后面我们会看到这种联系是十分密切的。当有语句char ary[100] = {'a', 'b', 'c', 'd','e', 'f'}; char \*cp = ary; 后，cp就指向了数组array中的第一个元素。我们可以通过指针来访问数组的元素：printf("%d", \*cp); 此语句的作用是打印出cp所指向的元素的值，也就是数组的第一个元素。现在通过cp = &array[3];使cp指向数组中的第4个元素，然后我们就可以对它进行各种操作了。

实际中经常会用指针来访问数组元素，当两个指针指向同一个数组时，会用到指针的算术运算：

<1>．指针+整数 或 指针-整数

指针与一个整数相加的结果是一个另一个指针。例如将上面的cp加1，运算后产生的指针将指向数组中的下一个字符。事实上当指针和一个整数相加减时，所做的就是指针加上或减去步长乘以那个整数的积。所谓步长就是指针所指向的类型的大小(即指针移动一个位置时要跳过几个字节)。下面的例子会让大家更加明了：

intia[100] = {0, 1, 2, 3, 4, 5};           double da[100] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0};

int \*ip =ia;                                     double \*dp = da;

ip += 3;                                           dp += 3;

ip加上3实际进行的操作是ip + 4 \* 3，因为ip指向的元素的类型为int；而dp加3实际进行的操作是dp + 8 \* 3，因为dp指向的元素的类型为double；这正是指针需要定义基类型的原因，因为编译器要知道一个指针移动时的步长。

要注意的是指针的算术运算只有在原始指针和计算出来的新指针都指向同一个数组的元素或指向数组范围的下一位置时才是合法的；另外，这种形式也适用于使用malloc动态分配获得的内存。

下面这段代码在大多数编译器上都是可以运行的，但它却是不安全的，因为b元素后面的内存区域所存储的内容是不确定的，有可能是受系统保护的，如果又编写了对p解引用的语句，那么很可能会造成运行时错误:

   int b;

          int \*p = &b;

   p += 2;

          printf("%p\n", p);

<2>。指针间的减法

当两个指针指向同一数组或有一个指针指向该数组末端的下一位置时，两个指针还可以做减法运算。

          intia[100] = {0, 1, 2, 3, 4, 5};

          int \*ip =ia;

          int \*ig =ia +3;

          ptrdiff\_t n = ig – ip;

n应该为3，表示这两个指针所指向的元素的距离为3，ptrdiff\_t是一个标准库类型，它是一个无符号整数，可以为负数。注意指针进行减法得到的结果指示出两指针所指向元素间的距离，即它们之间相隔几个数组元素，与步长的概念无关。

另外，一个指针可以加减0，指针保持不变；如果一个指针具有0值(空指针)，则在该指针上加0也是合法的，结果得到另一个值为0的指针；对两个空指针做减法运算，得到的结果也是0。注意：ANSI C标准没有定义两个指针相加的运算，如果两个指针相加，绝大多数编译器会在编译期报错。

2．指针与数组的爱恨情仇

数组和指针有着千丝万缕的联系，它们之间的问题困惑着不少朋友，有的朋友对它们的概念不是很清楚，所以可能会导致误用，从而出错。下面就对数组名是什么，数组什么时候和指针相同等相关问题做出解释。

<1>．数组名

声明中：当我们声明一个数组时，编译器将根据声明所指定的元素数量及类型为数组保留内存空间，然后再创建数组名，编译器会产生一个符号表，用来记录数组名和它的相关信息，这些信息中包含一个与数组名相关联的值，这个值是刚刚分配的数组的第一个元素的首地址(一个元素可能会占据几个地址，如整型占4个，此处是取起始地址)。现在声明一个数组：int ia[100]; 编译器此时为它分配空间，假设第一个数组元素的地址为0x22ff00；那么编译器会进行类似#define ia 0x22ff00的操作，这里只是模拟，真实情况并非完全一样，我们在编程时无需关注编译器所做的事情，但要知道此时(声明时)数组名只是一个符号，它与数组第一个元素的首地址相关联。注意：数组的属性和指针的属性不相同，在声明数组时，同时分配了用于容纳数组元素的空间；而声明一个指针时，只分配了用于容纳指针本身的空间。

表达式中：当我们在表达式中使用数组名，如：ia[10] = 25;时，这个名字会被编译器转换为指向数组第一个元素的常量指针(指针本身的值不可变)，它的值还是数组的第一个元素的首地址(一个指针常量)，编译器的动作类似于int \*const  ia = (void \*)0x22ff00; 这里我们应重点关注的是：数组名是一个常量指针(常指针)，即指针自身的值不能被改变。如果有类似ia++或ia+=3这类的语句是绝对不对的，会产生编译错误。注意：当数组名作为sizeof操作符的操作数时，返回的是整个数组的长度，也就是数组元素的个数乘以数组元素类型的大小；另外，在对数组名实施&操作时，返回的是一个指向数组的指针，而非具有某个指针常量值的指针(这个问题在后面会详细论述)。

通过数组名引用数组元素时：在前面讲过的指针算术运算中指针加上一个整型数，结果仍然是指针，并且可以对这个指针直接解引用，不用先把它赋给一个新指针。如int last = \*(ia + 99);此时的ia已经是一个常指针了，这个表达式计算出ia所指向元素后面的第99个元素的地址，然后对它解引用得到相应的值。这个表达式等价于int last = ia[99];事实上每当我们采用[ ]的方式引用数组元素时，如：ia[99]，在编译器中都会转换成指针形式，也就是\*(ia + 99) (这里ia + 99和&ia[99]的值都为数组最后一个元素的首地址，所以\*(ia + 99)和\*&ia[99]得到的结果是一样的，较难理解的是\*&ia[99]，按照优先级和结合性规则，先对ia[99]取地址再解引用，有些编译器见到这种表达式会直接优化成ia[99]。)现在可以看出来在表达式中，指针和数组名的使用可以互换，但唯一要注意的就是：数组名是常指针，不能对它的值进行修改。ia + 99是可以的，但ia++是不行的，它的意思是ia = ia +1;修改了ia的值。

作为函数参数：先来了解一下函数的实参与形参。实参(argument)是在实际调用时传递给函数的值；形参(parameter)是一个变量，在函数定义或者原型中声明。C语言标准规定作为形参的数组声明转换为指针。在声明函数形参的特定情况下，编译器会把数组形式改写成指向数组第一个元素的指针。所以不管下面哪种声明方式，都会被转换成指针：

       void array\_to\_pointer(int \*ia){……} //无需转换

       void array\_to\_pointer(int ia[ ]){……} //被转换成\*ia

       void array\_to\_pointer(int ia[100 ]){……} //被转换成\*ia

那么如果有下面的操作

       void array\_test(int ia[100])

{

        doubleda[10];

  printf("%d", sizeof( ia ));

  ia++;

  //da++;   //编译错误，数组名是常指针

       }

输出的结果为4，此时的ia是作为函数形参而声明的数组，已经被转换为了一个不折不扣的指针(不再是常指针了)，因此ia++;是合法的，不会引发编译错误。为什么C语言要把数组形参当作指针呢？因为C语言中所有非数组形式的数据实参(包括指针)均以值传递形式调用(所谓值传递就是拷贝出一个实参的副本并把这个副本赋值给形参，从此实参与形参是各不相干的，形参值的变化不会影响实参)。如果要拷贝整个数组，在时间和空间上的开销都很大，所以把作为形参的数组和指针等同起来是出于效率原因的考虑。我们可以把形参声明为数组(我们打算传递给函数的东西)或者指针(函数实际接收到的东西)，但在函数内部，编译器始终把它当作一个指向数组第一个元素(数组长度未知)的指针。在函数内部，对数组参数的任何引用都将产生一个对指针的引用。我们没有办法传递一个数组本身，因为它总是被自动转换为指向数组首元素的指针，而在函数内部使用指针时，能对数组进行的操作几乎和传递数组没有区别，唯一不同的是：使用sizeof(形参数组名)来获得数组的长度时，得到的只是一个指针的大小，正如上面所述的ia。但要注意：以上讨论的都是数组名作为函数形参的特殊情况，当我们在函数体内声明一个数组时，它就是一个普通的数组，它的数组名仍是一个常指针，所以上面的da++;仍会引起编译错误，请大家不要混淆。

    还有一点，既然是值传递，那么理所当然地，在用数组名作为实参调用函数时，实参数组名同样会被转换为指向数组第一个元素的指针。

<2>．指向数组的指针

好了，关于数组名的讨论可以告一段落了，现在来看指针与数组的另一种联系。在前面说过，当对一个一维数组的数组名进行 &操作时，返回的是一个指向数组的指针。现在我们就来看看什么是指向数组的指针。在C语言中，所谓的多维数组实际上只是数组的数组，也就是说一个数组中的每个元素还是数组，由于二维数组较为常用，所以本文着重讨论二维数组，更多维数组的原理与二维数组相同。所谓二维数组(数组的数组)，就是每个元素都是一个一维数组的一维数组。另外，请大家先有一个感性的认识：指向数组的指针主要用来对二维数组进行操作，大家不理解没有关系，我会在后面详细说明。

通常我们声明一个指向一维数组中的元素的指针是这样做的：int ia[100], \*ip = ia; ip指向这个数组的第一个元素，通过指针的算术运算，可以让ip指向数组中的任一元素。对于二维数组，我们的目的同样是让一个指针指向它的每一个元素，只不过这次的元素类型是一个数组，所以在声明这个指针时稍有不同，假设有二维数组int matrix[50][100], C语言采用如下的方式来声明一个指向数组的指针。int (\*p) [100];比普通声明稍复杂一些，但并不难理解。由于括号的优先级是最高的，所以首先执行解引用，表明了p是一个指针，接下来是数组下标的引用，说明p指向的是某种类型的数组，前面的int表明p指向的这个数组的每个元素都是整数。对于这个声明还可以换一个角度来理解：现在要声明的是一个指针，因此在标识符p前面加上\*。如果从内向外读p的声明，可以理解为\*p是int[100]类型，即p是一个指向含有100个元素的数组的指针。

有些朋友可能对于一个用来操纵二维数组的指针只使用一个下标表示困惑，为什么声明不是int (\*p) [50][100]呢？现在来回顾一下操纵一维数组的指针声明int \*ip = ia;它表示ip指向了一个数组的第一个元素，通过对指针的算术运算可以使它指向数组中的任何一个元素，编译器不需要知道指针ip指向的是一个多长的数组。对于二维数组道理相同，int (\*p) [100] = matrix; matrix可以看成是一个长度为50的一维数组，每个元素都是一个int[100]型的数组，p同样指向了matrix数组的第一个元素(第一个int[100]型的数组)，通过对p的算术运算也可以使它指向matrix数组中的任意一个元素而不需要知道matrix是一个多长的数组，但一定需要知道matrix中每个数组元素的长度，所以就有了int (\*p) [100]这种形式的声明。由此可知，如果进行p + n (n为整数)这样的运算，每次的步长就是n \* 100 \* sizof (int)，相当于跳过了矩阵中的n行，因为每行都有100个元素并且元素为整型,所以跳过了n \* 100 \* sizof (int)个字节，指向这些字节之后的位置。现在，对指向数组指针的声明方式的疑惑我认为已经讲清楚了。下面来看一个关于数组长度的问题。

在C语言中没有一种内建的机制去检查一个数组的边界范围，完全是由程序员自己去控制，这是C语言设计的一种哲学或者说一种理念：给程序员最大的自由度，程序员应该知道自己在做什么。凡事有利有弊，自由度大了，出错的几率就高了。很有朋友(包括我自己)在初用数组时应该会或多或少地遇到过数组越界的问题。在前面的论述中提到了通过对指针的算术运算可以使它指向数组中的任何一个元素包括超出数组范围的第一个元素，这个超出范围的第一个元素实际上是不存在的，这个“元素”的地址在数组所占的内存之后，它是数组的第一个出界点，这个地址可以赋给指向数组元素的指针，但ANSI C仅允许它进行赋值或比较运算，不能对保存这个地址的指针进行解引用或下标运算。

<3>．再回首——数组名

现在又要开始数组名的讨论了，之所以再回首而没有一气呵成，是因为在一维数组名和二维数组名之间需要一个过渡知识，就是指向数组的指针。在表达式中一维数组名会转换为指向数组第一个元素的指针，二维数组也是一样的，请大家牢记在C语言中二维数组就是数组的数组，所以也会被转换为指向第一个元素的指针，它的第一个元素是一个数组，所以最终的结果就是二维数组名被转换成指向数组的指针。

来看int(\*p) [100] = matrix;此时的matrix被转换为一个指向数组的指针，对于matrix[n]，是matrix数组的第n+1个元素的名字，也就是matrix数组中50个有着100个整型元素的数组之一，所以可以有p = &matrix[n]; 即p指向了一个数组元素，也就是矩阵中的某一行，matrix[n]本身是一个一维数组的数组名，它会被转换为指向数组第一个元素的指针，因此可以有int \*column\_p = matrix[n];这个表达式是最常见的也最容易理解。如果对matrix[n]进行sizeof操作结果是100\*sizeof(int);而sizeof(matrix)结果是50\*100\*sizeof (int)。

总结一下，p和matrix都是指向矩阵的一行(一个整型数组)，p+ m 或者matrix + m都将使指针跳跃m行(m个整型数组)，column\_p和matrix[n]都指向某行(一个整型数组)的第一个元素，column\_p + m和matrix[n] + m都将使指针跳跃m个整型元素。假若要访问二维数组matrix中第1行第1列(注意数组下标从0开始)的元素可以有以下的几种方式(i为int型变量)：相当于：

matrix[50]={matrix[0],...,matrix[49]}(matrix+1相当于第二个元素的地址，而第二个元素又是数组matrix[1]的数组名。所以\*(matrix+1) == matrix[1])

通过数组名引用         通过指针p的引用         通过指针column\_p的引用

i = matrix [0][0];          i = \*(\*(p+0)+0);           column\_p = matrix[0];

i = \*(matrix [0]+0);        i = \*(p[0] + 0);             i = \*(column\_p+0);

i = \*(\*(matrix+0)+0);      i = (\*(p + 0))[0];            i = column\_p[0];

上面的各种表达式中的“+0”均可以省略掉，但如果数字不是0就不能省略了，由此在引用第1行第1列的元素时会产生一些简化的表达式，如下：

通过数组名引用         通过指针p的引用         通过指针column\_p的引用

i = matrix [0][0];          i = \*\*p;                  column\_p= matrix[0];

i = \*matrix [0];           i =\*p[0];                 i= \*column\_p;

i = \*\*matrix;             i = (\*p )[0];               i = column\_p[0];

现在来看下面语句的输出，它们可能会让你感到困惑：

printf("%p\n", &matrix);     对应的指针操作：无

printf("%p\n", matrix);       对应的指针操作：printf("%p\n", p);

printf("%p\n",&matrix[0]);   对应的指针操作：printf("%p\n", p);

printf("%p\n",matrix[0]);     对应的指针操作：printf("%p\n", column\_p);

printf("%p\n",&matrix[0][0]); 对应的指针操作：printf("%p\n", column\_p);

在我机器上的输出是：

0022B140

0022B140

0022B140

0022B140

0022B140

输出的值虽然一样，但这些参数的类型却不完全相同。下面一一做出解释：

&matrix： 对二维数组名取地址，返回一个指向二维数组的指针；

matrix：  二维数组名会被转换为指向第一行(第一个数组)的指针，与&matrix[0]等价；

&matrix[0]：对第一个一维数组的数组名取地址，返回一个指向一维数组的指针；

matrix[0]：  二维数组中第一个一维数组的数组名，与&matrix[0][0]是等价的；

&matrix[0][0]：对第一行第一列元素取地址，返回一个指向整型元素的指针。

在ANSIC标准中没有说明对一个数组名进行&操作是否合法，但现在的编译器大都认为是合法的，并且会返回一个指向数组的指针。简单地说就是：对一个n维数组的数组名取地址得到的是一个指向n维数组的指针。

另外，上例中相对应的指针表示方式我也写了出来。对于&matrix没有相对应的指针表示方式，因为我们没有定义那种类型的指针，用p是表示不出来的，如果对p进行&的话，得到的是p这个指针的地址，而不是matrix的地址，两者完全不同，值也不会相同的。

再次提醒大家：无论是matrix还是matrix[0]，它们都是数组名，都会被转化为一个常指针，不能修改它们自身的值。对于&matrix、&matrix[0]、&matrix[0][0]，它们得到的都是常量(指针常量)，表示的是物理内存的地址，同样不能修改它们的值。本质上讲就是你不能也不可能修改一个物理内存的地址。

<4>．指针数组

在声明一个指向数组的指针时千万不要丢到那个括号，如：int (\*p) [100];如果丢掉了括号那就完全改变了意图，从而意外地声明了一个指针数组。指针数组要比指向数组的指针好理解，而且前面已经有了一些铺垫，这个概念相信大家可以很轻松地搞定。

所谓指针数组就是一个数组它的所有元素都是指针，这与普通的数组没什么区别，不过元素是指针罢了。下面来声明一个指针数组char \*cars[10];这种方式可能不太利于理解，如果写成char\* cars[10];的形式，可读性就很强了，它明确表示了cars是一个具有10个元素的数组，每个元素的类型都是char\*。下面举一个完整的例子并用它来结束这段指针与数组的爱恨情仇。

#include<stdio.h>

voiddisplay\_car\_brands(const char \*brand\_table[], int size)  // 有关参数和局部变量中的const解析请参见[第6章Section 3](http://blog.csdn.net/porscheyin/archive/2008/12/06/3461670.aspx).

{

       const char \*\*cbp;

       for(cbp = brand\_table;  cbp < brand\_table+ size;  cbp++) {

              printf("%s\n", \*cbp);

              printf("%c\n",\*\*cbp);

       }

}

int main( )

{

       const char \*cars[] = {     "ASTONMARTIN",

                                  "AUDI",

                                  "BENZ",

                                  "BENTLEY",

                                  "BMW",

                                  "BUGATTI",

                                  "FERRARI",

                                  "JAGUAR",

                                 "LAMBORGHINI",

                                 "MASERATI",

                                  "MAYBACH",

                                  "ROLLSROYCE"

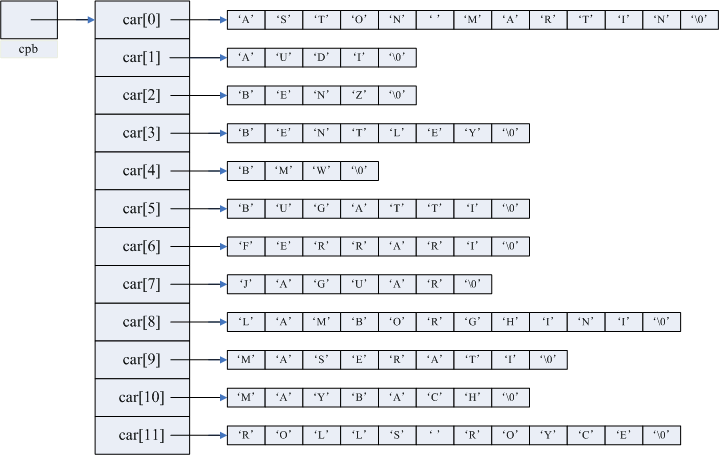
                            };

       int array\_size = sizeof(cars)/sizeof(cars[0]);

       display\_car\_brands(cars,array\_size);

       return 0;

}



首先我们定义了一个指针数组，每个元素都是一个指向char类型的指针，并将它初始化。初始化后的数组有12个指针元素，分别指向以上的各个字符串(即保存着每个字符串首字符的地址)。在display\_car\_brands()中定义了一个二级指针cbp，指向指针数组的第一个元素，通过自增cbp，遍历每一个数组元素，正如上图所示(这种数组就是所谓的锯齿型数组，也叫交错数组，即jagged array)。

对cbp解引用得到每个数组元素的值(即每一个字符串首字符的地址)，然后通过printf("%s\n", \*cbp);语句来输出每个字符串，注意\*cbp得到的是字符串首字符的地址，通过%s格式项接收一个地址以输出整个字符串。接下来的printf("%c\n", \*\*cbp);输出每个串的第一个字母，\*\*cbp首先得到每个字符串首字符的地址，再对该地址解引用得到相应的字符。因此程序的输出为：

ASTON MARTIN

A

AUDI

A

BENZ

B

BENTLEY

B

BMW

B

BUGATTI

B

FERRARI

F

JAGUAR

J

LAMBORGHINI

L

MASERATI

M

MAYBACH

M

ROLLS ROYCE

R

**四．分清函数指针和指针函数**

关于指针和数组斩不断理还乱的恩怨还真是说了不少，不过现在应该已经理清了。有了上一讲的基础，本讲的内容相对来说就比较容易理解了。

**1．指向函数的指针(函数指针)**

来分析这样一个声明，void (\*f) ( );虽然()的优先级高于\*，但由于有括号存在，首先执行的是解引用，所以f是一个指针；接下来执行( )，表明f指向一个函数，这个函数不返回任何值。现在得出结论：f是一个指向不接受参数且不返回任何值的函数的指针，简称函数指针(pointer to function)。

对比一下int(\*p) [100]，p是一个指向含有100个整型元素的数组的指针，它们有一个共同的特点：指针声明符(\*)和标识符(f或p)都被限制在一个括号中，由于括号的优先级是最高的，所以我们从标识符开始由内向外分析，即可得到以上结果。

**<1>．初始化**

注意指向函数的指针(函数指针)指向的是函数而非普通的变量，它所指向的函数也是有特定类型的，函数的类型由它的返回值类型以及形参列表确定，和函数名无关。对函数指针初始化时可以采用相同类型函数的函数名或函数指针(当然还有零指针常量)。假如有函数void test ( )，int wrong\_match (int)和函数指针void (\*ptf) ( )。

下面的初始化是错误的，因为函数指针的类型与函数的类型不匹配：

f = wrong\_match;

f = & wrong\_match;

ptf = wrong\_match;

ptf = & wrong\_match;

以下初始化及赋值是合法的：

f = test;

f = &test;

ptf = test;

ptf = &test;

f = pf;

要做出解释的是test和&test都可以用来初始化函数指针。**C语言规定函数名会被转换为指向这个函数的指针，除非这个函数名作为& 操作符或sizeof操作符的操作数(注意：函数名用于sizeof的操作数是非法的)。**也就是说f = test;中test被自动转换为&test，而f= &test;中已经显示使用了&test，所以test就不会再发生转换了。因此直接引用函数名等效于在函数名上应用 & 运算符，两种方法都会得到指向该函数的指针。

**<2>．通过函数指针调用函数**

通过函数指针调用函数可以有两种方法，直接使用函数指针或在函数指针前使用解引用运算符，如下所示：

f = test;

ptf = test;

f ( );

(\*f) ( );  //指针两侧的括号非常重要，表示先对f解引用，然后再调用相应的函数

ptf ( );

(\*ptf) ( ); //括号同样不能少

以上语句都能达到调用test函数的作用。ANSI C标准将f ( )认为是(\*f)( )的简写形式，并且推荐使用f ( )形式，因为它更符合函数调用的逻辑。**要注意的是：如果指向函数的指针没有初始化，或者具有0值(零指针常量)，那么该指针不能在函数调用中使用。只有当指针已经初始化，或被赋值后指向某个函数才能安全地用来调用函数。**

**<3>．探究函数名**

现在有如下程序：

#include<stdio.h>

void test( )

{

    printf("test called!\n");

}

int main()

{

   void (\*f) ( );

    f = test;

   f ( );

    (\*f)( );

    //test++;             // **error，标准禁止对指向函数的指针进行自增运算**

          //test = test + 2;        //**error，不能对函数名赋值，函数名也不能用于进行算术运算**

          printf("%p\n", test);

          printf("%p\n", &test);

          printf("%p\n", \*test);

  return 0;

}

在我机器上的运行结果为：

test called!

test called!

004013EE

004013EE

004013EE

这个程序中较难理解的是3个输出语句都可以得到函数的入口地址。首先来看函数名test，它与数组名类似(注意：只是类似)，是一个符号用来标识一个函数的入口地址，在使用中函数名会被转换为指向这个函数的指针，指针的值就是函数的入口地址，&test在前面已经说了：显示获取函数的地址。对于\*test，可以认为由于test已经被转换成了函数指针,指向这个函数，所以\*test就是取这个指针所指向的函数名，而又根据函数名会被转换指向该函数的指针的规则，这个函数也转变成了一个指针，所以\*test最终也是一个指向函数test的指针。对它们采用%p格式项输出，都会得到以16进制数表示的函数test的入口地址。注意函数的地址在编译期是未知的，而是在链接时确定的。

**2．返回指针的函数(指针函数)**

类比指针数组(还记得吗)，理解指针函数将会更加轻松。所谓指针函数，就是返回指针的函数，函数可以不返回任何值，也可以返回整型值，实型值，字符型值，当然也可以返回指针值。一个指针函数的声明：int \*f(int i, int j); 回想一下指针数组的声明：char \*cars[10];同样的把它写成好理解的形式(非业界惯例)int\* f(int i, int j);这样一来已经十分明了了，由于( )的优先级高于\*，因此f先与()结合，所以f是一个具有两个int型参数，返回一个指向int型指针的函数。

C语言的库函数中有很多都是指针函数，比如字符串处理函数，下面给出一些函数原型：

char \*strcat( char \*dest, const char \*src );

char \*strcpy( char \*dest, const char \*src );

char \*strchr( const char \*s, int c );

char \*strstr( const char \*src, const char\*sub );

注意函数的返回值不仅仅局限于指向变量的指针，也可以是指向函数的指针。初遇这种函数的声明可能会痛苦一点儿，但练习两三次应该是可以理解并掌握的。首先来看这个声明：int (\*function(int)) (double\*,char);要了解此声明的含义，首先来看function(int)，将function声明为一个函数，它带有一个int型的形式参数，这个函数的返回值为一个指针，正是我们本将开头讲过的函数指针int (\*) (double\*, char);这个指针指向一个函数，此函数返回int型并带有两个分别是double\*型和char型的形参。如果使用typedef可以将这个声明简化：

typedef int (\*ptf) (double\*, char);

ptf function(int );

**要说明一下，对于typedef int (\*ptf) (double\*,char); 注意不要用#define的思维来看待typedef，如果用#define的思维来看的话会以为(\*ptf)(double\*, char)是int的别名，但这样的别名看起来好像又不是合法的名字，于是会处于迷茫状态。实际上，上面的语句把ptf定义为一种函数指针类型的别名，它和函数指针类型int (\*) (double\*, char);等价，也就是说ptf现在也是一种类型。**

**3．函数指针和指针函数的混合使用**

函数指针不仅可以作为返回值类型，还可以作为函数的形式参数，如果一个函数的形参和返回值都是函数指针，这个声明看起来会更加复杂，例如：

void (\*signal (int sig, void (\*func) (intsiga)) ) ( int siga );看上去确实有些恼人，我们来一步一步的分析。现在要分析的是signal，因为紧邻signal的是优先级最高的括号，首先与括号结合，所以signal为一个函数，括号内为signal的两个形参，一个为int型，一个为指向函数的指针。接下来从向左看，\*表示指向某对象的指针，它所处的位置表明它是signal的返回值类型，现在可以把已经分析过的signal整体去掉，得到void (\*) ( int siga )，很清晰了吧。又是一个函数指针，这个指针与signal形参表中的第二个参数类型一样，都是指向接受一个int型形参且不返回任何值的函数的指针。同样地，用typedef可以将这个声明简化：

typedef void (\*p\_sig) (int);

p\_sig signal(int sig, p\_sig func);

这个signal函数是C语言的库函数，在signal.h中定义，用来处理系统中产生的信号，是UNIX/Linux编程中经常用到的一个函数，所以在此单独拿出来讲解一下。

**4．函数指针数组**

还有一种较为常用的关于函数指针的用法——函数指针数组。假设现在有一个文件处理程序，通过一个菜单按钮来选择相应的操作(打开文件，读文件，写文件，关闭文件)。这些操作都实现为函数且类型相同，分别为：

void open( );

void read( );

void write( );

void close( );

现在定义一个函数指针类型的别名PF：typedefvoid (\*PF) ( );把以上4种操作取地址放入一个数组中，得到：

PF file\_options[ ] = {

                &open,

                &read,

                &write,

                &close

};

这个数组中的元素都是指向不接受参数且不返回任何值的函数的指针，因此这是一个函数指针数组。接下来，定义一个函数指针类型的指针action并初始化为函数指针数组的第一个元素：PF\* action = file\_options;，如果不好理解，可以类比一下int ia[4] = {0, 1, 2, 3}; int \*ip = ia;，这里PF相当于int，这样应该比较好懂了。通过对指针action进行下标操作可以调用数组中的任一操作，如：action[2]( )会调用write操作，以此类推。在实际中，指针action可以和鼠标或者其他GUI对象相关联，以达到相应的目的。

**5．关于指针的复杂声明**

第4点中的函数指针数组采用了typedef来声明，这是应该提倡的方法，因为它可读性更高。如果不使用typedef，那么分析起来就会比较复杂，结果是void (\*file\_options[ ]) ( );对于C语言的复杂声明我不想讲太多，因为在实际中用到的机会并不多，并且推荐大家多用typedef来简化声明的复杂度。对于分析复杂声明有一个极为有效的方法——右左法则。右左法则的大致描述为：从未定义的变量名开始阅读声明，先向右看，然后向左看。当遇到括号时就调转阅读的方向。括号内的所有内容都分析完毕就跳出括号。这样一直继续下去，直到整个声明都被分析完毕。来分析一个的例子：int \* (\* (\*fp) (int) ) [10];

阅读步骤：

1．从未定义的变量名开始阅读 --------------------------------------------fp

2．往右看，什么也没有，遇到了)，因此往左看，遇到一个\* ------一个指向某对象的指针

3．跳出括号，遇到了(int) -----------------------------------一个带一个int参数的函数

4．向左看，发现一个\* ---------------------------------------（函数）返回一个指向某对象的指针

5．跳出括号，向右看，遇到[10] ------------------------------一个10元素的数组

6．向左看，发现一个\* ---------------------------------------一个指向某对象指针

7．向左看，发现int -----------------------------------------int类型

所以fp是指向函数的指针,该函数返回一个指向数组的指针，此数组有10个int\*型的元素。

对此我不再多举例了，下面给出一些声明，有兴趣的朋友可以试着分析一下，答案我会在下一讲中给出：

1． int \*( \*( \*a[5]) ( ) ) ( );

2． void \* (\*b) ( char, int (\*) ( ) );

3． float ( \*(\*c[10]) (int\*) ) [5];

4． int ( \*(\*d)[2][3] ) [4][5];

5． int (\*(\*(\*e) ( int\* ))[15]) (int\*);

6． int ( \*(\*f[4][5][6]) (int\*) ) [10];

7． int \*(\*(\*(\*g)( ))[10]) ( );

**五．指针与结构**

**——理解C++和数据结构的基础**

指针不仅可以指向变量、数组、函数，还可以和结构(structure)联系起来，这使得C语言的威力倍增，初学C语言的朋友对结构可能不太重视，对它的理解也不够深入，但事实上，结构是一个非常重要的工具，有了它我们可以很轻松的构建一些仅靠其它C语言特性做起来很复杂的程序。深入地理解结构会对你理解C++的面向对象有很大帮助，并且会让在你学习数据结构时有一份惬意的心情。(本讲中默认读者已经对结构有了基本的认识)

附：在本讲的最后，会提供上一讲复杂声明的答案。

**1．深入理解结构**

**<1>．为什么需要结构**

C语言中有很多内置类型，如int型、double型、char型等，但仅仅使用这些类型并不能很好地表达我们的意图。假如我们想表示一辆汽车，汽车有很多自身的属性，如：最大功率、车重、颜色、最高时速、价格等等。这些属性都应该和汽车紧密相连，我们每构造一辆车时，这些固有的属性都应该一并被定义。有了这种需求，就催生了结构。结构有时也被翻译为结构体。当我们定义了一个结构后，就意味着定义了一种新的类型，结构是C语言中为数不多的能让我们自己掌控所定义类型的语言特性，在C中使用结构可以把不同类型的值存储在一起。

**<2>．使用结构**

首先看一个例子，我将用这个例子贯穿本讲的内容：

struct car

{

char name[50];

int max\_power;

      int weight;

      char color[20];

      int max\_speed;

      int price;

      struct car \*next;

} one\_car, famous\_cars[12];

关键字struct表示将要定义一个结构，编译器会将从struct直到分号为止的部分识别为一个结构。花括号中存放我们要进行组合的内容，car是一个可选(optional)的结构标签(tag)，也叫结构名，有了这个标签我们就可以在将来的程序中用struct car作为struct{int max\_power; int weight; char color[20]; int max\_kph; float price;}的简写形式。右花括号后的one\_car是一个structcar类型的变量，而famous\_cars[12]是一个数组，这个数组包含了20个structcar类型的元素。

结构中的所有内容都称作成员，所有成员必须在结构的内部声明，一旦结构定义完成后，就没有任何办法可以增加成员了。当定义了一个结构后不仅定义了一个新的类型，同时也定义了一个新的作用域，在struct car中有7个成员，这些成员的作用范围只是这个结构中，我们在结构外是看不到也用不了的，如果想用怎么办呢。可以通过成员操作符(**.**)来访问结构中的成员。(**.**)的左操作数是结构变量名，右操作数是成员名，如果我想把one\_car的颜色设置成红色，那么可以写one\_car**.**color= "red";现在我还想将12辆名车中的第一辆车的价格调整到525万元，如：famous\_cars[0]**.**price = 525;此时的famous\_cars[0]和one\_car是同一类的对象，都是struct car类型的变量，只不过famous\_cars[0]是隶属于famous\_cars[12]这个数组的一个元素，而one\_car是个自由身罢了。

**<3>．结构的自引用**

大家应该早已注意到struct car中的最后一个成员struct car \*next了，有些朋友对此不太理解。结构可以包含任意类型的成员，包括其他类型的结构，当然也可以包含自身类型的结构，但在定义这种结构时，只能写成包含指向自身类型的指针，而不能写成以下这种形式：

struct car

{

 char name[50];

 int max\_power;

       int weight;

       char color[20];

       int max\_speed;

       int price;

**struct car** next;     **//错，此时类型不完整，不能包含自身类型的变量**

};

因为next是另一个完整的结构变量，它的内部同样会包含一个成员struct car**next**，这个成员还会包括一个成员struct car next，如此下去永无休止，编译器将无法确定这个结构struct car的大小。为了解决这个问题，我们可以把第一个成员声明为struct car \*next;，一个struct car结构的指针就是用来存储同类型结构变量的地址的，是有固定长度的，此时编译器可以轻松确定struct car的大小。

**2．指向结构的指针**

刚刚说过的struct car \*next就是一个指向结构的指针，它既可以在结构内作为一个结构成员，也可以作为一个自由的对象出现在结构定义后面的任何地方，现在我们再来定义一个指向struct car类型的指针car\_pointer：structcar \*car\_pointer = &one\_car;。现在我们拥有了一个指向结构的指针，如何通过这个指针来访问结构变量中的成员呢？同样是通过解引用操作符，比如现在想将one\_car的发动机进行调教，把最大功率增加100千瓦：(\*car\_pointer)**.**max\_power += 100;首先通过\*操作符获取结构变量one\_car，然后再对它的max\_power成员进行修改。特别要注意括号不能丢，因为成员操作符**.**的优先级高于\*操作符，如果没有括号的话将会导致错误。正是由于这个有些令人厌烦的访问方式，C语言提供了更加快捷且易于理解的方式——(->)操作符。通过这个操作符，我们就无需再受丢括号的困扰了。 -> 操作符和 **.**操作符具有同样的优先级。使用方法：car\_pointer-> max\_power += 100;

**3．链式存储**

有了以上的基础，现在可以来讨论通过结构与指向结构的指针构建的一种存储数据的方式——链式存储了。假设现在要组织一次名车的巡礼活动，车辆的排列顺序按品牌的字典序进行。为了保证车队的连贯性，车与车之间通过绳索连接起来。这幅图景如果用程序来表现是这样的：

#include <stdio.h>

#include <stdlib.h>

struct car

{

   char name[50];

   int max\_power;

   int weight;

   char color[20];

   int max\_speed;

   int price;

   struct car \*next;

};

int main()

{

   int i = 0;

**struct car\*car\_pointer = (struct car \*)malloc(sizeof(struct car));**

   struct car \*record = car\_pointer;

   struct car \*remove = car\_pointer;

   struct car \*current = car\_pointer;

   for (i; i < 2; i++)

   {

**car\_pointer->next = (struct car \*)malloc(sizeof(structcar));**

       printf("Please input car's name : ");

       scanf("%s",car\_pointer->name);

       printf("%s's max\_power : ",car\_pointer->name);

       scanf("%d",&car\_pointer->max\_power);

       printf("%s's weight : ",car\_pointer->name);

       scanf("%d",&car\_pointer->weight);

       printf("%s's color : ",car\_pointer->name);

       scanf("%s",car\_pointer->color);

       printf("%s's max\_speed : ",car\_pointer->name);

       scanf("%d",&car\_pointer->max\_speed);

       printf("%s's price : ",car\_pointer->name);

       scanf("%d",&car\_pointer->price);

       printf("\n");

       car\_pointer = car\_pointer->next;

   }

**car\_pointer->next =NULL;**

   printf("巡礼车辆如下：\n");

**for (record;record->next != NULL; record = record->next)**

   {

       printf("%s\n",record->name);

       printf("%s's max\_power : %dKW\n", record->name, record->max\_power);

       printf("%s's weight : %dKG\n", record->name, record->weight);

       printf("%s's color : %s\n",record->name, record->color);

       printf("%s's max\_speed : %dKM/H\n", record->name, record->max\_speed);

       printf("%s's price : %d万元(￥)\n",record->name, record->price);

       printf("\n");

   }

**while(current->next != NULL)**

   {

       remove = current;

       current = current->next;

       free(remove);

   }

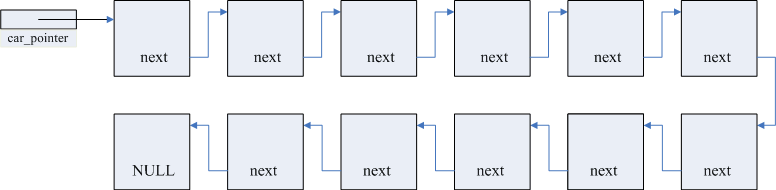
   return 0;

}

首先，我们定义一个结构，然后在主函数中开辟一段能容纳结构struct car的内存，并定义一个指向此结构类型的指针car\_pointer将其指向刚才所开辟的内存空间。接下来定义几个同类型的指针并初始化为car\_pointer留作他用。在链式存储中，每一块这样的内存空间都被称作结点，代表着链中的一个实体。接下来的for循环用来控制创建多少个结点，每一次循环，都创建一个新的结点，并用前一个结点的next成员指向新创建的结点，再通过相应的输入来初始化每个结点的各个成员。在循环结束后，将最后一个结点的next成员置为NULL。

然后，用先前创建的指针record来控制这个车队的输出，因为此时car\_pointer已经走到了车队的末尾，无法再用它从头进行遍历了。

最后将动态分配的内存释放掉。注意此时同样需要另一个指向车队头部的指针current，用来遍历车队链表，并创建一个remove指针指向要删除的结点。如果我们不设置remove指针而是直接将current所指的结点释放掉，就找不到下一个结点了。本例使用链式存储结构所创建的车队链表如下图所示：



**4．用typedef给结构创建别名**

本将的最后一个话题，我们再来谈谈typedef。关键字typedef用来给一种类型起一个别名，理所当然地可以给一个结构类型定义一个别名，用法上没有什么区别。如果想给struct car定义一个别名Car可以这样处理：

typedef struct car

{

   char name[50];

   int max\_power;

   int weight;

   char color[20];

   int max\_speed;

   int price;

   struct car \*next;

} Car;

通过typedef引入Car这个名字作为struct car {……}的简写形式。此后我们就可以用Car来定义变量了，如Car one\_car;或Car famous\_cars[12]; 当然也可以按如下的方式来起别名，而且会更清晰易读：

struct car

{

   char name[50];

   int max\_power;

   int weight;

   char color[20];

   int max\_speed;

   int price;

   struct car \*next;

};

typedef struct car Car;

typedef struct car \*CarPointer;

此时的CarPointer是struct car \*类型的别名，即指向struct car的指针类型。理解了这种表示法对于理解C++中的类是很有帮助的。

**差一点儿忘了，在第四讲中复杂声明还有悬而未决的问题，现在给出解答，如果有不明白或者有兴趣深入研究的朋友可以联系我：邮箱：porscheyin@hotmail.com，以下是原题及答案：**

1．  int \*( \*( \*a[5]) ( ) ) ( );

Answer：a是一个数组，它的5个元素都是指向函数的指针，该函数仍旧返回一个指向函数的指针。

2．  void \* (\*b) ( char, int (\*) ( ) );

Answer：b是一个函数指针，该函数接受两个形参，分别是char型和一个函数指针，返回值类型为void \*。

3．  float ( \*(\*c[10]) (int \*) ) [5];

Answer：c是一个数组，它的10个元素都是指向函数的指针，所指函数接受int \*型形参并返回一个指向数组的指针，这个数组包含5个float元素。

4．  int ( \*(\*d)[2][3] ) [4][5];

Answer：d是一个指向数组的指针，此数组的元素又是一个指向数组的指针。

5．  int (\*(\*(\*e) ( int\* ))[15]) (int\*);

Answer：e是一个函数指针，该函数的返回值是一个指向数组的指针，所指向数组的元素又是函数指针，指向的函数具有int \*型形参，返回值类型为int。

6．  int ( \*(\*f[4][5][6]) (int\*) ) [10];

Answer：f是一个数组，这个数组的元素是函数指针，这类函数具有int \*型形参并返回指向数组的指针，所指向的数组的元素是具有10个int型元素。

7．  int \*(\*(\*(\*g)( ))[10]) ( );

Answer：g是个指向函数的指针，它所指向的函数返回一个指向包含10个元素的数组的指针，数组元素的类型是指向函数的指针，所指向的函数不接受参数且返回值类型为int \*。

另外，对于本讲中的巡礼车队程序，如果进行相应输入后，会得到类似下面的输出。我选了一些最新的世界知名车辆，它们的品牌与第四讲中指针数组例子中的车辆品牌一致，我只是在每个品牌中选了一个车系，下面是输出结果：

巡礼车辆如下：

ASTON-MATIN--DB9

ASTON-MATIN--DB9's max\_power : 335KW

ASTON-MATIN--DB9's weight : 1760 KG

ASTON-MATIN--DB9's color : GREEN

ASTON-MATIN--DB9's max\_speed : 300 KM/H

ASTON-MATIN--DB9's price : 304万元(￥)

AUDI--R8

AUDI--R8's max\_power : 309 KW

AUDI--R8's weight : 1560 KG

AUDI--R8's color : SILVER

AUDI--R8's max\_speed : 301 KM/H

AUDI--R8's price : 180万元(￥)

BENZ--SLR-722

BENZ--SLR-722's max\_power : 478 KW

BENZ--SLR-722's weight : 1550 KG

BENZ--SLR-722's color : BLACK

BENZ--SLR-722's max\_speed : 337 KM/H

BENZ--SLR-722's price : 888万元(￥)

BENTLEY--CONTINENTAL-GTC

BENTLEY--CONTINENTAL-GTC'smax\_power : 411 KW

BENTLEY--CONTINENTAL-GTC's weight: 2495 KG

BENTLEY--CONTINENTAL-GTC's color :AZURE

BENTLEY--CONTINENTAL-GTC'smax\_speed : 312 KM/H

BENTLEY--CONTINENTAL-GTC's price :358万元(￥)

BMW--M6

BMW--M6's max\_power : 373 KW

BMW--M6's weight : 1785 KG

BMW--M6's color : BLUE

BMW--M6's max\_speed : 250 KM/H

BMW--M6's price : 229万元(￥)

BUGATTI--VEYRON-16.4

BUGATTI--VEYRON-16.4's max\_power :736 KW

BUGATTI--VEYRON-16.4's weight : 1888 KG

BUGATTI--VEYRON-16.4's color :RED&BLACK

BUGATTI--VEYRON-16.4's max\_speed :407 KM/H

BUGATTI--VEYRON-16.4's price :2500万元(￥)

FERRARI--599GTB

FERRARI--599GTB's max\_power : 456KW

FERRARI--599GTB's weight : 1690 KG

FERRARI--599GTB's color : RED

FERRARI--599GTB's max\_speed : 330 KM/H

FERRARI--599GTB's price : 376万元(￥)

JAGUAR--XK

JAGUAR--XK's max\_power : 224 KW

JAGUAR--XK's weight : 1595 KG

JAGUAR--XK's color : GREEN

JAGUAR--XK's max\_speed : 250 KM/H

JAGUAR--XK's price : 142万元(￥)

LAMBORGHINI--MURCIELAGO-LP640

LAMBORGHINI--MURCIELAGO-LP640'smax\_power : 471 KW

LAMBORGHINI--MURCIELAGO-LP640'sweight : 1665 KG

LAMBORGHINI--MURCIELAGO-LP640'scolor : BLACK

LAMBORGHINI--MURCIELAGO-LP640'smax\_speed : 340 KM/H

LAMBORGHINI--MURCIELAGO-LP640'sprice : 438万元(￥)

MASERATI--GRANTURISMO

MASERATI--GRANTURISMO's max\_power: 298 KW

MASERATI--GRANTURISMO's weight : 1955 KG

MASERATI--GRANTURISMO's color :RED

MASERATI--GRANTURISMO's max\_speed: 285 KM/H

MASERATI--GRANTURISMO's price :219万元(￥)

MAYBACH--62S

MAYBACH--62S's max\_power : 450 KW

MAYBACH--62S's weight : 2855 KG

MAYBACH--62S's color : WHITE

MAYBACH--62S's max\_speed : 250 KM/H

MAYBACH--62S's price : 900万元(￥)

ROLLS-ROYCE--PHANTOM-101EX

ROLLS-ROYCE--PHANTOM-101EX'smax\_power : 338 KW

ROLLS-ROYCE--PHANTOM-101EX'sweight : 2590 KG

ROLLS-ROYCE--PHANTOM-101EX's color: BLUE

ROLLS-ROYCE--PHANTOM-101EX'smax\_speed : 250 KM/H

ROLLS-ROYCE--PHANTOM-101EX's price: 680万元(￥)

六．使用指针时的“陷阱”

“C语言诡异离奇，陷阱重重，却获得了巨大成功！”——C语言之父Dennis M.Ritchie。Ritchie大师的这句话体现了C语言的灵活性以及广泛的使用，但也揭示了C是一种在应用时要时刻注意自己行为的语言。C的设计哲学还是那句话：使用C的程序员应该知道自己在干什么。有时用C写的程序会出一些莫名其妙的错误，看似根源难寻，但仔细探究会发现很多错误的原因是概念不清。在我们经常掉进去的这些“陷阱”中，围绕着指针的数量为最。这一讲将对使用指针时遇到的一些问题做出分析，以避免在日后落入此类“陷阱”之中。

1．指针与字符串常量

在第二讲指针的初始化中提到可以将一个字符串常量赋给一个字符指针。但有没有朋友想过为什么能够这样进行初始化呢？回答这个问题之前，我们先来搞清楚什么是字符串常量。字符串常量是位于一对双引号内部的字符序列(可以为空)。

当一个字符串常量出现于表达式中，除以下三种情况外：

1．作为 &操作符的操作数；

2．作为sizeof操作符的操作数；

3．作为字符数组的初始化值

字符串常量都会被转化为由一个指针所指向的字符数组。例如：char \*cp = "abcdefg";不满足上述3个条件，所以"abcdefg"会被转换为一个没有名字的字符数组，这个数组被abcdefg和一个空字符'/0'初始化，并且会得到一个指针常量，它的值为第一个字符的地址，不过这些都是由编译器来完成的。现在可以解释用一个字符串常量初始化一个字符指针的原因了，一个字符串常量的值就是一个指针常量。那么对于下面的语句，朋友们也不该感到迷惑了：

printf("%c\n",\*"abcdefg");

        printf("%c\n", \*("abcdefg"+ 1));

        printf("%c\n","abcdefg"[5]);

        \*"abcdefg"：字符串常量的值是一个指针常量，指向的是字符串的第一个字符，对它解引用即可得到a；

    \*("abcdefg"+ 1)：对这个指针进行算术运算则其指向下一个字符，再对它解引用，得到b；  
    "abcdefg"[5]：既然"abcdefg"是一个指针，那么"abcdefg"[5]就可以写成\*("abcdefg" + 5)，所以得到f。

回忆一下大家所学的初始化数组的方法：char ca[ ] = {'a','b','c','d','e','f','g','/0'};这种方法实在太笨拙了，所以标准提供了一种快速方法用于初始化字符数组：char ca[ ] = "abcdefg";这个字符串常量满足了上面的第3条：用来初始化字符数组，所以不会被转换为由一个指针所指向的字符数组。它只是用单个字符来初始化字符数组的简便写法。再来对比以下两个声明：

char ca[ ] = "abcdefg";

char \*cp = "abcdefg";

它们的含义并不相同，前者是初始化一个字符数组的元素，后者才是一个真正的字符串常量，如下图所示：

                 char ca[ ] = "abcdefg";

http://p.blog.csdn.net/images/p_blog_csdn_net/porscheyin/EntryImages/20081206/%E6%95%B0%E7%BB%84ca.gif

                                                                         图1

                char \*cp ="abcdefg";

http://p.blog.csdn.net/images/p_blog_csdn_net/porscheyin/EntryImages/20081206/%E6%8C%87%E9%92%88cp.gif

                                                         图2

要注意的是：用来初始化字符数组的字符串常量，编译器会在栈中为字符数组分配空间，然后把字符串中的所有字符复制到数组中；而用来初始化字符指针的字符串常量会被编译器安排到只读数据存储区，但也是按字符数组的形式来存储的，如图2。我们可以通过一个字符指针读取字符串常量但不能修改它，否则会发生运行时错误。正如下面的例子：

1．charca[ ] = "abcdefg";

        2．char\*cp = "abcdefg";

        3．ca[0]= 'b';

        4．printf("%s\n", ca );

        5．cp[0]= 'b';

        6．printf("%s\n", cp );

此程序第3行修改的不是只读数据区中的字符串常量，而是由字符串常量复制而来的存在于栈中的字符数组ca的一个元素。但第5行却修改了用于初始化字符指针的位于只读数据区的字符串常量，所以会发生运行时错误。大家不要认为所有的字符串常量都存储在不同的地址，标准C允许编译器为两个包含相同字符的字符串常量使用相同的存储地址，而且现实中大多数厂商的编译器也都是这么做的。来看下面的程序：

        charstr1[] = "abc";

        charstr2[] = "abc";

        char\*str3 = "abc";

        char\*str4 = "abc";

        printf("%d\n", str1 == str2 );

        printf("%d\n",str3 == str4 );

    输出的结果是：0 1

str1,str2是两个不同的字符数组，分别被初始化为"abc"，它们在栈中有各自的空间；而str3,str4是两个字符指针分别被初始化为包含相同字符的字符串常量，它们指向相同的区域。

2．strlen( )和sizeof

请看下面程序：

char a[1000];

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

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

这段代码的输出可不一定是1000, 0。sizeof(a)的结果一定是1000，但strlen(a)的结果就不能确定了。根本原因在于：strlen( )是一个函数，而sizeof是一个操作符，这导致了它们的种种不同：

1．sizeof可以用类型(需要用括号括起来)或变量做操作数，而strlen( )只接受char\*型字符指针做参数，并且该指针所指向的字符串必须是以'/0'结尾的；

2．sizeof是操作符，对数组名使用sizeof时得到的是整个数组所占内存的大小，而把数组名作为参数传递给strlen( )后数组名会被转换为指向数组第一个元素的指针；

3．sizeof的结果在编译期就确定了，而strlen( )是在运行时被调用。

由于上例中的数组a[1000]没有初始化，所以数组内的元素及元素个数都是不确定的，可能是随机值，所以用strlen(a)会得到不同的值，这取决于产生的随机数，但sizeof的结果一定是1000，因为sizeof是在编译时获取char a[1000]中char和1000这两个信息来计算空间的。

3．const指针与指向const的指针

对于常量指针(const pointer)和指针常量大家应该可以分清楚了。常量指针：指针本身的值不可以改变，可以把const理解为只读的，如：int \*const c\_p；指针常量：一个指针类型的常量，如：(int \*)0x123456ff。现在引入一个新的概念：指向const的指针，即一个指针它所指向的是一个const对象，如：const int \*p\_to\_const;表明p\_to\_const是一个指向constint型变量的指针，p\_to\_const自身的值是可以改变的，但是不能通过对p\_to\_const解引用来改变所指的对象的值，看下面的例子会更加清晰：

int \*p = NULL;                                        //定义一个整型指针并初始化为NULL

int i = 0;                                                 //定义一个整型变量并初始化为0

const int ci = 0;                                     //定义一个只读的整型变量并初始化，程序中不能再对它赋值

const int \*p\_to\_const = NULL;              //定义一个指向只读整型变量的指针，初始化为NULL

p = &i;                                                   //ok，让p指向整型变量i

p\_to\_const = &ci;                                 //ok，让p\_to\_const指向ci

\*p = 5;                                                  //ok，通过指针p修改i的值

        \*p\_to\_const = 5;                                  //error，p\_to\_const所指向的是一个只读变量，不能通过p\_to\_const对ci进行修改

        p\_to\_const = &i;                                  //ok，让指向const对象的指针指向普通对象

        p\_to\_const = p;                                   //ok，将指向普通对象的指针赋给指向const对象的指针

        p = (int \*) &ic;                                      //ok，强制转化为(int \*)型，赋值操作符两侧操作数类型相同

p = (int \*) p\_to\_const;                         //ok，同上

p = &ic;                                              //error，错误原因下述

p = p\_to\_const;                                  //error，同上

对于最后两行的赋值，需要说明一下。C语言中对于指针的赋值操作(包括实参与形参之间的传递)应该满足：两个操作数都是指向有限定符或都是指向无限定符的类型相兼容的指针；或者左边指针所指向的类型具有右边指针所指向的类型的全部限定符。例如const int \*表示“指向一个具有const限定符的int类型的指针”，即const所修饰的是指针所指向的类型，而非指针。因此，p = &ic; 中的&ic得到的是一个指向const int型变量的指针，类型和p\_to\_const一样。p\_to\_const所指向的类型为const int，而p所指向的类型为int，p在赋值操作符左边，p\_to\_const在赋值操作符右边，左边指针所指向的类型并不具有右边指针所指向类型的全部限定符，所以会出错。

    小扩展：{让我们再深入一些，如果现在有一个指针int \*\*bp和一个指针const int \*\*cbp那么这样的赋值也是错误的：cbp = bp;因为const int \*\*表示“指向有const限定符的int类型的指针的指针”。int \*\* 和const int \*\*都是没有限定符的指针类型，它们所指向的类型是不一样的(int \*\*指向int \*，而const int \*\*指向const int \*)，所以它们是不兼容的，根据指针赋值条件来判断，这两个指针之间不能相互赋值。

实际上和const int \*\*相兼容的类型是const int \* \*const，所以下面代码是合法的：

const int \* \*const const\_p\_to\_const = &p\_to\_const;

/\*定义一个指向有const限定符的int类型的指针的常指针，它必需在定义时初始化，程序中不能再对它赋值。由于既不能修改指针的值也不能通过指针改变所指对象的值，所以在实际中，这种指针的用途并不广\*/

const int \*\*cpp;

cpp = const\_p\_to\_const;

左操作数cpp所指向的类型是const int\*，右操作数const\_p\_to\_const指向类型也为const int\*，满足指针赋值条件：左边指针所指向的类型具有右边指针所指向类型的全部限定符，只不过const\_p\_to\_const是一个const指针，不能被再赋值，所以反过来是不能进行赋值的。还要注意被const限定的对象只能并且必需在声明时初始化。}

4．C语言中的值传递

在第3将中提到过C语言只提供函数参数的传值调用机制，即函数调用时，拷贝出一个实参的副本并把这个副本赋值给形参，从此实参与形参是各不相干的，形参在函数中的改变不会影响实参。我在前面说过C语言中所有非数组形式的数据实参(包括指针)均以传值形式调用，这并不与C语言只提供传值调用机制矛盾，对于数组形参会被转换为指向数组首元素的指针，当我们用数组名作为实参时，实际进行的也是值传递。请看程序：

#include <stdio.h>

void pass\_by\_value(char parameter[])

{

printf("形参的值：  %p\n",parameter);

printf("形参的地址：%p\n", &parameter);

printf("%s\n",parameter);

}

int main( )

{

          charargument[100] = "C语言只有传值调用机制！";

   printf("实参的值：  %p\n",argument);

   pass\_by\_value(argument);

          return0;

}

在我机器上的输出结果为：实参的值：  0022FF00

形参的值：  0022FF00

形参的地址：0022FED0

C语言只有传值调用机制！

当执行pass\_by\_value(argument);时，实参数组名argument被转换为指向数组第一个元素的指针，这个指针的值为(void \*)0022FF00，然后把这个值拷贝一份赋给形式参数parameter，形参parameter虽然被声明为字符数组，但是会被转换为一个指针，它是创建在栈上的一个独立对象(它有自己独立的地址)并接收实参值的那份拷贝。从而我们看到了实参与形参具有相同的值，并且形参有一个独立的地址。再来看一个简单的例子：

#include <stdio.h>

void pointer\_plus(char \*p)

{

           p+= 3;

}

int main( )

{

          char\*a = "abcd";

     pointer\_plus(a);

          printf("%c\n", \*a);

          return0;

}

如果哪位朋友认为输出是d，那么你还是没有搞清楚值传递的概念，此程序中将a拷贝一份赋给p，从此a和p就没有关系了，在函数pointer\_plus中增加p的值实际上增加的是a的那份拷贝的值，根本不会影响到a，在主函数中a仍旧指向字符串的第一个字符，因此输出为a。如果想让pointer\_plus改变a所指向的对象，采用二级指针即可，程序如下：

#include <stdio.h>

void pointer\_plus(char\*\*p)

{

           \*p += 3;

}

int main( )

{

          char\*a = "abcd";

     pointer\_plus(&a);

          printf("%c\n", \*a);

          return0;

}

5．垂悬指针(Dangling pointer)

垂悬指针是我们在使用指针时经常出现的，所谓垂悬指针就是指向了不确定的内存区域的指针，通常对这种指针进行操作会使程序发生不可预知的错误，因此我们应该避免在程序中出现垂悬指针，一些好的编程习惯可以帮助我们减少这类事件的发生。

造成垂悬指针的原因通常分为三种，对此我们一个一个地进行讨论。

第一种：在声明一个指针时没有对其初始化。在C语言中不会对所声明的自动变量进行初始化，所以这个指针的默认值将是随机产生的，很可能指向受系统保护的内存，此时如果对指针进行解引用，会引发运行时错误。解决方法是在声明指针时将其初始化为NULL或零指针常量。大家应该养成习惯为每个新创建的对象进行初始化，此时所做的些许工作会为你减少很多烦恼。

第二种：指向动态分配的内存的指针在被free后，没有进行重新赋值就再次使用。就像下面的代码：

          int \*p =(int \*)malloc(4);

         \*p = 10;

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

         free(p);

         ……

         ……

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

这就可能会引发错误，首先我们声明了一个p并指向动态分配的一块内存空间，然后通过p对此空间赋值，再通过free()函数把p所指向的那段内存释放掉。注意free函数的作用是通过指针p把p所指向的内存空间释放掉，并没有把p释放掉，所谓释放掉就是将这块内存中的对象销毁，并把这块内存交还给系统留作他用。指针p中的值仍是那块内存的首地址，倘若此时这块内存又被指派用于存储其他的值，那么对p进行解引用就可以访问这个当前值，但如果这块内存的状态是不确定的，也许是受保护的，也许不保存任何对象，这时如果对p解引用则可能出现运行时错误，并且这个错误检测起来非常困难。所以为了安全起见，在free一个指针后，将这个指针设置为NULL或零指针常量。虽然对空指针解引用是非法的，但如果我们不小心对空指针进行了解引用，所出现的错误在调试时比解引用一个指向未知物的指针所引发的错误要方便得多，因为这个错误是可预料的。

第三种：返回了一个指向局部变量的指针。这种造成垂悬指针的原因和第二种相似，都是造成一个指向曾经存在的对象的指针，但该对象已经不再存在了。不同的是造成这个对象不复存在的原因。在第二种原因中造成这个对象不复存在的原因是内存被手动释放掉了，而在第三种原因中是因为指针指向的是一个函数中的局部变量，在函数结束后，局部变量被自动释放掉了(无需程序员去手动释放)。如下面的程序：

#include<stdio.h>

#include<stdlib.h>

int\*return\_pointer()

{

   int i=3;

   int \*p =&i;

   return p;

}

int main()

{

   int \*rp = return\_pointer();

   printf("%d\n", \*rp);

   return 0;

}

在return\_pointer函数中创建了一个指针p指向了函数内的变量i (在函数内创建的变量叫做局部变量)，并且将这个指针作为返回值。在主函数中有一个指针接收return\_pointer的返回值，然后对其解引用并输出。此时的输出可能是3，也可能是0，也可能是其他值。本质原因就在于我们返回了一个指向局部变量的指针，这个局部变量在函数结束后会被编译器销毁，销毁的时间由编译器来决定，这样的话p就有可能指向不保存任何对象的内存，也可能这段内存中是一个随机值，总之，这块内存是不确定的，p返回的是一个无效的地址