指针详解

（摘自网络，版权已失。潘晓光辛苦排版整理）

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。 要搞清一个指针需要搞清指针的四方面的内容：指针的类型，指针所指向的 类型，指针的值或者叫指针所指向的内存区，还有指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

```
(1)int*ptr;
(2)char*ptr;
(3)int**ptr;
(4)int(*ptr)[3];
(5)int*(*ptr)[4];
```

➢ 指针的类型

从语法的角度看，你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。让我们看看例一中各个指针的类型：

```
(1)int*ptr;//指针的类型是 int*
(2)char*ptr;//指针的类型是 char*
(3)int**ptr;//指针的类型是 int**
(4)int(*ptr)[3];//指针的类型是 int(*)[3]
(5)int*(*ptr)[4];//指针的类型是 int*(*)[4]
```

怎么样？找出指针的类型的方法是不是很简单？

➢ 指针所指向的类型

当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。例如：

```
(1)int*ptr;//指针所指向的类型是 int
(2)char*ptr;//指针所指向的的类型是 char
(3)int**ptr;//指针所指向的的类型是 int*
(4)int(*ptr)[3];//指针所指向的的类型是 int()[3]
(5)int*(*ptr)[4];//指针所指向的的类型是 int*()[4]
```

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对 C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

指针的值，或者叫指针所指向的内存区或地址。指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在 32 位程序里，所有类型的指针的值都是一个 32 位整数，因为 32 位程序里内存地址全都是 32 位长。指针所指向的内存区就

是从指针的值所代表的那个内存地址开始，长度为 si zeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是 XX，就相当于说该指针指向了以 XX 为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？

➢ 指针本身所占据的内存区

指针本身占了多大的内存？你只要用函数 sizeof(指针的类型)测一下就知道了。在 32 位平台里，指针本身占据了 4 个字节的长度。

指针本身占据的内存这个概念在判断一个指针表达式是否是左值时很有用。

➢ 指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

例二：

```
1、chara[20];

2、int*ptr=a;

...
3、ptr++;
```

在上例中，指针 ptr 的类型是 int*,它指向的类型是 int，它被初始化为指向整形变量 a。接下来的第 3 句中，指针 ptr 被加了 1，编译器是这样处理的：它把指针 ptr 的值加上了 sizeof(int)，在 32 位程序中，是被加上了 4。由于地址是用字节做单位的，故 ptr 所指向的地址由原来的变量 a 的地址向高地址方向增加了 4 个字节。

由于 char 类型的长度是一个字节，所以，原来 ptr 是指向数组 a 的第 0 号单元开始的四个字节，此时指向了数组 a 中从第 4 号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
intarray[20];
int*ptr=array;
...

//此处略去为整型数组赋值的代码。

...

for(i=0;i<20;i++)
{
(*ptr)++;
ptr++;
```

}

这个例子将整型数组中各个单元的值加 1。由于每次循环都将指针 ptr 加 1，所以每次循环都能访问数组的下一个单元。

再看例子：

例四：

1、chara[20];

2、int*ptr=a;

...

3、ptr+=5;

在这个例子中，ptr 被加上了 5，编译器是这样处理的：将指针 ptr 的值加上 5 乘 sizeof(int)，在 32 位程序中就是加上了 5 乘 4=20。由于地址的单位是字节，故现在的 ptr 所指向的地址比起加 5 后的 ptr 所指向的地址来说，向高地址方向移动了 20 个字节。在这个例子中，没加 5 前的 ptr 指向数组 a 的第 0 号单元开始的四个字节，加 5 后，ptr 已经指向了数组 a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，ptr 是被减去 5，那么处理过程大同小异，只不过 ptr 的值是被减去 5 乘 sizeof(int)，新的 ptr 指向的地址将比原来的 ptr 所指向的地址向低地址方向移动了 20 个字节。

总结一下，一个指针 ptrold 加上一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向高地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

一个指针 ptrold 减去一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值减少了 n 乘 sizeof(ptrold 所指向的类型)个字节，就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向低地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

➢ 运算符&和*

这里&是取地址运算符，*书上叫做"间接运算符"。

&a 的运算结果是一个指针，指针的类型是 a 的类型加个*，指针所指向的类型是 a 的类型，指针所指向的地址嘛，那就是 a 的地址。

*p 的运算结果就五花八门了。总之*p 的结果是 p 所指向的东西，这个东西有这些特点：它的类型是 p 指向的类型，它所占用的地址是 p 所指向的地址。

例五：

```
inta=12;
intb;
int*p;
int**ptr;
p=&a;
//&a 的结果是一个指针，类型是 int*，指向的类型是 int，指向的地址是 a 的地址。
*p=24;
//*p 的结果，在这里它的类型是 int，它所占用的地址是 p 所指向的地址，显然，
*p 就是变量 a。
ptr=&p;
```

//&p 的结果是个指针，该指针的类型是 p 的类型加个*，在这里是 int **。该指针所指向的类型是 p 的类型，这里是 int*。该指针所指向的地址就是指针 p 自己的地址。

*ptr=&b;

//*ptr 是个指针，&b 的结果也是个指针，且这两个指针的类型和所指向的类型是一样的，所以用&b 来给*ptr 赋值就是毫无问题的了。

**ptr=34;

//*ptr 的结果是 ptr 所指向的东西，在这里是一个指针，对这个指针再做一次*运算，结果就是一个 int 类型的变量。

➢ 指针表达式

一个表达式的最后结果如果是一个指针，那么这个表达式就叫指针表式。

下面是一些指针表达式的例子：

例六：

```
inta,b;
intarray[10];
int*pa;
pa=&a;//&a 是一个指针表达式。
int**ptr=&pa;//&pa 也是一个指针表达式。
*ptr=&b;//*ptr 和&b 都是指针表达式。
pa=array;
pa++;//这也是指针表达式。
```

例七：

```
char*arr[20];
char**parr=arr;//如果把 arr 看作指针的话，arr 也是指针表达式
char*str;
str=*parr;//*parr 是指针表达式
str=*(parr+1);//*(parr+1)是指针表达式
str=*(parr+2);//*(parr+2)是指针表达式
```

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。

在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针 pa，既然 pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

➢ 指针和数组的关系

数组的数组名其实可以看作一个指针。看下例：

例八：

```
intarray[10]={0,1,2,3,4,5,6,7,8,9},value;
...
value=array[0];//也可写成：value=*array;
value=array[3];//也可写成：value=*(array+3);
```

value=array[4];//也可写成：value=*(array+4);

上例中，一般而言数组名 array 代表数组本身，类型是 int[10]，但如果把 array 看做指针的话，它指向数组的第 0 个单元，类型是 int*，所指向的类型是数组单元的类型即 int。因此*array 等于 0 就一点也不奇怪了。同理，array+3 是一个指向数组第 3 个单元的指针，所以 *(array+3)等于 3。其它依此类推。

例九：

```
char*str[3]={
"Hello,thisisasample!",
"Hi,goodmorning.",
"Helloworld"
};
chars[80];
trcpy(s,str[0]);//也可写成 strcpy(s,*str);
trcpy(s,str[1]);//也可写成 strcpy(s,*(str+1));
trcpy(s,str[2]);//也可写成 strcpy(s,*(str+2));
```

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 str 当作一个指针的话，它指向数组的第 0 号单元，它的类型是 char**，它指向的类型是 char*。

str 也是一个指针，它的类型是 char，它所指向的类型是 char，它指向的地址是字符串"Hello,thisisasample!"的第一个字符的地址，即’H’的地址。 str+1 也是一个指针，它指向数组的第 1 号单元，它的类型是 char**，它指向的类型是 char*。

(str+1)也是一个指针，它的类型是 char，它所指向的类型是 char，它指向"Hi,goodmorning."的第一个字符’H’，等等。

下面总结一下数组的数组名的问题。声明了一个数组 TYPEarray[n]，则数组名称 array 就有了两重含义：第一，它代表整个数组，它的类型是 TYPE[n]；第二，它是一个指针，该指针的类型是 TYPE*，该指针指向的类型是 TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第 0 号单元，该指针自己占有单独的内存区，注意它和数组第 0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 array++的表达式是错误的。

在不同的表达式中数组名 array 可以扮演不同的角色。

在表达式 sizeof(array)中，数组名 array 代表数组本身，故这时 sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第 0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式 array+n（其中 n=0，1，2，....。）中，array 扮演的是指针，故 array+n 的结果是一个指针，它的类型是 TYPE*，它指向的类型是 TYPE，它指向数组第 n 号单元。故 sizeof(array+n)测出的是指针类型的大小。

例十：

```
intarray[10];
int(*ptr)[10];
ptr=&array;：
```

上例中 ptr 是一个指针，它的类型是 int(*)[10]，他指向的类型是 int[10] ，我们用整个数组的首地址来初始化它。在语句 ptr=&array 中，array 代表数组本身。

本节中提到了函数 sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者。

例如：

int(*ptr)[10];

则在 32 位程序中，有：

izeof(int(*)[10])==4

izeof(int[10])==40

izeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

➢ 指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。intfun1(char*,int);

int(*pfun1)(char*,int);

pfun1=fun1;

....

Int a=(*pfun1)("abcdefg",7);//通过函数指针调用函数。

在指针的强制类型转换：ptr1=(TYPE*)ptr2 中，如果 sizeof(ptr2 的类型)大于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是安全的。如果 sizeof(ptr2 的类型)小于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是不安全的。至于为什么，读者结合例十七来想一想，应该会明白的。

摘录的别人的:

C 语言所有复杂的指针声明，都是由各种声明嵌套构成的。如何解读复杂指针声明呢？右左法则是一个既著名又常用的方法。不过，右左法则其实并不是 C 标准里面的内容，它是从 C 标准的声明规定中归纳出来的方法。C 标准的声明规则，是用来解决如何创建声明的，而右左法则是用来解决如何辩识一个声明的，两者可以说是相反的。右左法则的英文原文是这样说的：

The right-left rule: Start reading the declaration from the innermost parentheses, go right, and then go left. When you encounter parentheses, the direction should be reversed. Once everything in the parentheses has been parsed, jump out of it. Continue till the whole declaration has been parsed.

这段英文的翻译如下：

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

笔者要对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，之所以是未定义的标识符，是因为一个声明里面可能有多个标识符，但未定义的标识符只会有一个。

现在通过一些例子来讨论右左法则的应用，先从最简单的开始，逐步加深：

int (*func)(int *p);

首先找到那个未定义的标识符，就是 func，它的外面有一对圆括号，而且左边是一个*号，这说明 func 是一个指针，然后跳出这个圆括号，先看右边，也是一个圆括号，这说明(*func)是一个函数，而 func 是一个指向这类函数的指针，就是一个函数指针，这类函数具有 int*类型的形参，返回值类型是 int。

int (*func)(int *p, int (*f)(int*));

func 被一对括号包含，且左边有一个*号，说明 func 是一个指针，跳出括号，右边也有

个括号，那么 func 是一个指向函数的指针，这类函数具有 int *和 int (*)(int*)这样的形参，返回值为 int 类型。再来看一看 func 的形参 int (*f)(int*)，类似前面的解释，f 也是一个函数指针，指向的函数具有 int*类型的形参，返回值为 int。

int (*func[5])(int *p);

func 右边是一个[]运算符，说明 func 是一个具有 5 个元素的数组，func 的左边有一个*，说明 func 的元素是指针，要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func 先跟[]结合，因此*修饰的是 func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为 int。

int (*(*func)[5])(int *p);

func 被一个圆括号包含，左边又有一个*，那么 func 是一个指针，跳出括号，右边是一个[]运算符号，说明 func 是一个指向数组的指针，现在往左看，左边有一个*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有 int*形参，返回值为 int 类型的函数。

int (*(*func)(int *p))[5];

func 是一个函数指针，这类函数具有 int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有 5 个 int 元素的数组。

要注意有些复杂指针声明是非法的，例如：

int func(void) [5];

func 是一个返回值为具有 5 个 int 元素的数组的函数。但 C 语言的函数返回值不能为数组，这是因为如果允许函数返回值为数组，那么接收这个数组的内容的东西，也必须是一个数组，但 C 语言的数组名是一个右值，它不能作为左值来接收另一个数组，因此函数返回值不能为数组。

int func[5](void);

func 是一个具有 5 个元素的数组，这个数组的元素都是函数。这也是非法的，因为数组的元素除了类型必须一样外，每个元素所占用的内存空间也必须相同，显然函数是无法达到这个要求的，即使函数的类型一样，但函数所占用的空间通常是不相同的。

作为练习，下面列几个复杂指针声明给读者自己来解析，答案放在第十章里。

int (*(*func)[5][6])[7][8];

int (*(*(*func)(int *))[5])(int *);

int (*(*func[7][8][9])(int*))[5];

实际当中，需要声明一个复杂指针时，如果把整个声明写成上面所示的形式，对程序可读性是一大损害。应该用 typedef 来对声明逐层分解，增强可读性，例如对于声明：

int (*(*func)(int *p))[5];

可以这样分解：

typedef int (*PARA)[5];

typedef PARA (*func)(int *);

这样就容易看得多了。

实际当中，需要声明一个复杂指针时，如果把整个声明写成上面所示的形式，对程序可读性是一大损害。应该用 typedef 来对声明逐层分解，增强可读性，例如对于声明：

int (*(*func)(int *p))[5];

可以这样分解：

typedef int (*PARA)[5];

typedef PARA (*func)(int *);

这个比较有实际，其它那么复杂的定义在现实使用中一点意义都没有,属写编译器的人研究的问题 ,如以下

int (*(*func)[5][6])[7][8];

int (*(*(*func)(int *))[5])(int *);

int (*(*func[7][8][9])(int*))[5];

➢ 指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十一：

```
tructMyStruct
{
    Int a;
    Int b;
    Int c;
}
```

MyStructss={20,30,40};

//声明了结构对象 ss，并把 ss 的三个成员初始化为 20，30 和 40。

MyStruct*ptr=&ss;

//声明了一个指向结构对象 ss 的指针。它的类型是 MyStruct*,它指向的类型是 MyStruct。

int*pstr=(int*)&ss;

//声明了一个指向结构对象 ss 的指针。但是它的类型和它指向的类型和 ptr 是不同的。

请问怎样通过指针 ptr 来访问 ss 的三个成员变量？

答案：

ptr->a;

ptr->b;

ptr->c;

又请问怎样通过指针 pstr 来访问 ss 的三个成员变量？

答案：

*pstr；//访问了 ss 的成员 a。

*(pstr+1);//访问了 ss 的成员 b。

*(pstr+2)//访问了 ss 的成员 c。

虽然我在我的 MSVC++6.0 上调式过上述代码，但是要知道，这样使用 pstr 来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元：

例十二：

```
intarray[3]={35,56,37};
int*pa=array;
通过指针 pa 访问数组 array 的三个单元的方法是：
*pa;//访问了第 0 号单元
*(pa+1);//访问了第 1 号单元
*(pa+2);//访问了第 2 号单元
```

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。

所有的 C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干个"填充字节"，这就导致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使*pstr 访问到了结构对象 ss 的第一个成员变量 a，也不能保证*(pstr+1)就一定能访问到结构成员 b。因为成员 a 和成员 b 之间可能会有若干填充字节，说不定*(pstr+1)就正好访问到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，嘿，这倒是个不错的方法。 通过指针访问结构成员的正确方法应该是象例十二中使用指针 ptr 的方法。

➢ 指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
intfun1(char*,int);
int(*pfun1)(char*,int);
pfun1=fun1;
....
inta=(*pfun1)("abcdefg",7);//通过函数指针调用函数。
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为实参。

例十三：

```
intfun(char*);
int a;
charstr[]="abcdefghijklmn";
a=fun(str);
...
intfun(char*s)
{
    int num=0;
    for(int i=0;i<strlen(s);i++)
    {
        num+=*s;s++;
    }
    returnnum;
}
```

这个例子中的函数 fun 统计一个字符串中各个字符的 ASCII 码值之和。前面说了，数组的名字也是一个指针。在函数调用中，当把 str 作为实参传递给形参 s 后，实际是把 str 的值传递给了 s，s 所指向的地址就和 str 所指向的地址一致，但是 str 和 s 各自占用各自的存储空间。在函数体内对 s 进行自加 1 运算，并不意味着同时对 str 进行了自加 1 运算。

➢ 指针类型转换

当我们初始化一个指针或给一个指针赋值时，赋值号的左边是一个指针，赋值号的右边是一个指针表达式。在我们前面所举的例子中，绝大多数情况下，指针的类型和指针表达式的类型是一样的，指针所指向的类型和指针表达式所指向的类型是一样的。

例十四：

1、floatf=12.3;

2、float*fptr=&f;

3、int*p;

在上面的例子中，假如我们想让指针 p 指向实数 f，应该怎么搞？是用下面的语句吗？

```
p=&f;
```

不对。因为指针 p 的类型是 int*，它指向的类型是 int。表达式&f 的结果是一个指针，指针的类型是 float*,它指向的类型是 float。两者不一致，直接赋值的方法是不行的。至少在我的 MSVC++6.0 上，对指针的赋值语句要求赋值号两边的类型一致，所指向的类型也一致，其它的编译器上我没试过，大家可以试试。为了实现我们的目的，需要进行"强制类型转换"：

```
p=(int*)&f;
```

如果有一个指针 p，我们需要把它的类型和所指向的类型改为 TYEP*TYPE， 那么语法格式是：

```
(TYPE*)p;
```

这样强制类型转换的结果是一个新指针，该新指针的类型是 TYPE*，它指向的类型是 TYPE，它指向的地址就是原指针指向的地址。而原来的指针 p 的一切属性都没有被修改。

一个函数如果使用了指针作为形参，那么在函数调用语句的实参和形参的结合过程中，也会发生指针类型的转换。

例十五：

```
voidfun(char*);
inta=125,b;
fun((char*)&a);
...
voidfun(char*s)
{
    charc;
    c=*(s+3);*(s+3)=*(s+0);*(s+0)=c;
    c=*(s+2);*(s+2)=*(s+1);*(s+1)=c;
}
```

注意这是一个 32 位程序，故 int 类型占了四个字节，char 类型占一个字节。函数 fun 的作用是把一个整数的四个字节的顺序来个颠倒。注意到了吗？在函数调用语句中，实参&a 的结果是一个指针，它的类型是 int*，它指向的类型是 int。形参这个指针的类型是 char*，它指向的类型是 char。这样，在实参和形参的结合过程中，我们必须进行一次从 int*类型到 char*类型的转换。结合这个例子，我们可以这样来想象编译器进行转换的过程：编译器先构造一个临时指针 char*temp，然后执行 temp=(char*)&a，最后再把 temp 的值传递给 s。所以最后的结果是：s 的类型是 char*,它指向的类型是 char，它指向的地址就是 a 的首地址。

我们已经知道，指针的值就是指针指向的地址，在 32 位程序中，指针的值其实是一个

32 位整数。那可不可以把一个整数当作指针的值直接赋给指针呢？就象下面的语句：

```
unsignedinta;
TYPE*ptr;//TYPE 是 int，char 或结构类型等等类型。
...
a=20345686;
ptr=20345686;//我们的目的是要使指针 ptr 指向地址 20345686（十进制 ）
ptr=a;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）
```

编译一下吧。结果发现后面两条语句全是错的。那么我们的目的就不能达到了吗？不，还有办法：

```
unsignedinta;
TYPE*ptr;//TYPE 是 int，char 或结构类型等等类型。
...
a=某个数，这个数必须代表一个合法的地址；
ptr=(TYPE*)a；//呵呵，这就可以了。
```

严格说来这里的(TYPE*)和指针类型转换中的(TYPE*)还不一样。这里的(TYPE*)的意思是把无符号整数 a 的值当作一个地址来看待。上面强调了 a 的值必须代表一个合法的地址，否则的话，在你使用 ptr 的时候，就会出现非法操作错误。

想想能不能反过来，把指针指向的地址即指针的值当作一个整数取出来。完 全可以。下面的例子演示了把一个指针的值当作一个整数取出来，然后再把这个整数当作一个地址赋给一个指针：

例十六：

```
inta=123,b;
int*ptr=&a;
char*str;
b=(int)ptr;//把指针 ptr 的值当作一个整数取出来。
tr=(char*)b;//把这个整数的值当作一个地址赋给指针 str。
```

现在我们已经知道了，可以把指针的值当作一个整数取出来，也可以把一个整数值当作地址赋给一个指针。

➢ 指针的安全问题

看下面的例子：

例十七：

```
chars='a';
int*ptr;
ptr=(int*)&s;
*ptr=1298;
```

指针 ptr 是一个 int*类型的指针，它指向的类型是 int。它指向的地址就是 s 的首地址。在 32 位程序中，s 占一个字节，int 类型占四个字节。最后一条语句不但改变了 s 所占的一个字节，还把和 s 相临的高地址方向的三个字节也改变了。这三个字节是干什么的？只有编译程序知道，而写程序的人是不太可能知道的。也许这三个字节里存储了非常重要的数据，也许这三个字节里正好是程序的一条代码，而由于你对指针的马虎应用，这三个字节的值被改变了！这会造成崩溃性的错误。

让我们再来看一例：

例十八：

1、chara;

2、int*ptr=&a;

...

3、ptr++;

4、*ptr=115;

该例子完全可以通过编译，并能执行。但是看到没有？第 3 句对指针 ptr 进行自加 1 运算后，ptr 指向了和整形变量 a 相邻的高地址方向的一块存储区。这块存储区里是什么？我们不知道。有可能它是一个非常重要的数据，甚至可能是一条代码。而第 4 句竟然往这片存储区里写入一个数据！这是严重的错误。所以在使用指针时，程序员心里必须非常清楚：我的指针究竟指向了哪里。在用指针访问数组的时候，也要注意不要超出数组的低端和高端界限，否则也会造成类似的错误。