目录

[前言:复杂类型说明 2](#_Toc216961364)

[1、细说指针 5](#_Toc216961365)

[1.指针的类型 5](#_Toc216961366)

[2.指针所指向的类型 6](#_Toc216961367)

[3.指针的值----或者叫指针所指向的内存区或地址 7](#_Toc216961368)

[4 指针本身所占据的内存区 8](#_Toc216961369)

[2、指针的算术运算 8](#_Toc216961370)

[3、运算符&和\* 13](#_Toc216961371)

[4、指针表达式 14](#_Toc216961372)

[5、数组和指针的关系 16](#_Toc216961373)

[6、指针和结构类型的关系 19](#_Toc216961374)

[7、指针和函数的关系 22](#_Toc216961375)

[8、指针类型转换 23](#_Toc216961376)

[9、指针的安全问题 28](#_Toc216961377)

[10、结束语 30](#_Toc216961378)

让你不再害怕指针

# 前言:复杂类型说明

要了解指针,多多少少会出现一些比较复杂的类型,所以我先介绍

一下如何完全理解一个复杂类型,要理解复杂类型其实很简单,一

个类型里会出现很多运算符,他们也像普通的表达式一样,有优先

级,其优先级和运算优先级一样,所以我总结了一下其原则:

从变量名处起,根据运算符优先级结合,一步一步分析.

下面让我们先从简单的类型开始慢慢分析吧:

int p; //这是一个普通的整型变量

int \*p; //首先从P 处开始,先与\*结合,所以说明P 是一

//个指针,然后再与int 结合,说明指针所指向

//的内容的类型为int 型.所以P 是一个返回整

//型数据的指针

int p[3]; //首先从P 处开始,先与[]结合,说明P 是一个数

//组,然后与int 结合,说明数组里的元素是整

//型的,所以P 是一个由整型数据组成的数组

int \*p[3]; //首先从P 处开始,先与[]结合,因为其优先级

//比\*高,所以P 是一个数组,然后再与\*结合,说明

//数组里的元素是指针类型,然后再与int 结合,

//说明指针所指向的内容的类型是整型的,所以

//P 是一个由返回整型数据的指针所组成的数组

int (\*p)[3]; //首先从P 处开始,先与\*结合,说明P 是一个指针

//然后再与[]结合(与"()"这步可以忽略,只是为

//了改变优先级),说明指针所指向的内容是一个

//数组,然后再与int 结合,说明数组里的元素是

//整型的.所以P 是一个指向由整型数据组成的数

//组的指针

int \*\*p; //首先从P 开始,先与\*结合,说是P 是一个指针,然

//后再与\*结合,说明指针所指向的元素是指针,然

//后再与int 结合,说明该指针所指向的元素是整

//型数据.由于二级指针以及更高级的指针极少用

//在复杂的类型中,所以后面更复杂的类型我们就

//不考虑多级指针了,最多只考虑一级指针.

int p(int); //从P 处起,先与()结合,说明P 是一个函数,然后进入

//()里分析,说明该函数有一个整型变量的参数

//然后再与外面的int 结合,说明函数的返回值是

//一个整型数据

Int (\*p)(int); //从P 处开始,先与指针结合,说明P 是一个指针,然后与

//()结合,说明指针指向的是一个函数,然后再与()里的

//int 结合,说明函数有一个int 型的参数,再与最外层的

//int 结合,说明函数的返回类型是整型,所以P 是一个指

//向有一个整型参数且返回类型为整型的函数的指针

int \*(\*p(int))[3]; //可以先跳过,不看这个类型,过于复杂

//从P 开始,先与()结合,说明P 是一个函数,然后进

//入()里面,与int 结合,说明函数有一个整型变量

//参数,然后再与外面的\*结合,说明函数返回的是

//一个指针,,然后到最外面一层,先与[]结合,说明

//返回的指针指向的是一个数组,然后再与\*结合,说

//明数组里的元素是指针,然后再与int 结合,说明指

//针指向的内容是整型数据.所以P 是一个参数为一个

//整数据且返回一个指向由整型指针变量组成的数组

//的指针变量的函数.

说到这里也就差不多了,我们的任务也就这么多,理解了这几个类型,其它

的类型对我们来说也是小菜了,不过我们一般不会用太复杂的类型,那样会

大大减小程序的可读性,请慎用,这上面的几种类型已经足够我们用了.

# 1、细说指针

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、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];

## 1.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？

## 2.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。

## 3.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？（重点注意）

## 4 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（后面会解释）是

否是左值时很有用。

# 2、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，以单元为单位。例如：

例二：

char a[20];

int \*ptr=(int \*)a; //强制类型转换并不会改变a 的类型

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

为指向整型变量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，因为在32 位程序中，int 占4 个字节。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：

int array[20]={0};

int \*ptr=array;

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 个单元，所以每次循环都能访问数组的下一个单元。

再看例子：

例四：

char a[20]="You\_are\_a\_girl";

int \*ptr=(int \*)a;

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。

下面请允许我再举一个例子:(一个误区)

例五:

#include<stdio.h>

int main()

{

char a[20]=" You\_are\_a\_girl";

char \*p=a;

char \*\*ptr=&p;

//printf("p=%d\n",p);

//printf("ptr=%d\n",ptr);

//printf("\*ptr=%d\n",\*ptr);

printf("\*\*ptr=%c\n",\*\*ptr);

ptr++;

//printf("ptr=%d\n",ptr);

//printf("\*ptr=%d\n",\*ptr);

printf("\*\*ptr=%c\n",\*\*ptr);

}

误区一、输出答案为Y 和o

误解:ptr 是一个char 的二级指针,当执行ptr++;时,会使指针加一个

sizeof(char),所以输出如上结果,这个可能只是少部分人的结果.

误区二、输出答案为Y 和a

误解:ptr 指向的是一个char \*类型,当执行ptr++;时,会使指针加一个

sizeof(char \*)(有可能会有人认为这个值为1,那就会得到误区一的答

案,这个值应该是4,参考前面内容), 即&p+4; 那进行一次取值运算不

就指向数组中的第五个元素了吗?那输出的结果不就是数组中第五个元

素了吗?答案是否定的.

正解: ptr 的类型是char \*\*,指向的类型是一个char \*类型,该指向的

地址就是p的地址(&p),当执行ptr++;时,会使指针加一个sizeof(char

\*),即&p+4;那\*(&p+4)指向哪呢,这个你去问上帝吧,或者他会告诉你在

哪?所以最后的输出会是一个随机的值,或许是一个非法操作.

总结一下:

一个指针ptrold 加(减)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(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 所指向的内存区向高(低)地址方向移动了n 乘

sizeof(ptrold 所指向的类型)个字节。

指针和指针进行加减：

两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的

结果指向一个不知所向的地方，而且毫无意义。两个指针可以进行减法

操作，但必须类型相同，一般用在数组方面，不多说了。

# 3、运算符&和\*

这里&是取地址运算符，\*是间接运算符。

&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。

例六：

int a=12; int b; 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，结果是一个int 类型的变量。

# 4、指针表达式

一个表达式的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：

例七：

int a,b;

int array[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。

例八：

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。

# 5、数组和指针的关系

数组的数组名其实可以看作一个指针。看下例：

例九：

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。

例十：

char \*str[3]={

"Hello,thisisasample!",

"Hi,goodmorning.",

"Helloworld"

};

chars[80]；

strcpy(s,str[0]); //也可写成strcpy(s,\*str);

strcpy(s,str[1]); //也可写成strcpy(s,\*(str+1));

strcpy(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。注意:字符串相当于是一个数组,在内存中以数组的形式储

存,只不过字符串是一个数组常量,内容不可改变,且只能是右值.如果

看成指针的话,他即是常量指针,也是指针常量.

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'

下面总结一下数组的数组名(数组中储存的也是数组)的问题:

声明了一个数组TYPE array[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，它是一个常量

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。在32 位程序中结果是4

例十一:

int array[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：

sizeof(int(\*)[10])==4

sizeof(int[10])==40

sizeof(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。

# 6、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。

例十二：

struct MyStruct

{

int a;

int b;

int c;

};

struct MyStruct ss={20,30,40};

//声明了结构对象ss，并把ss 的成员初始化为20，30 和40。

struct 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pstr 和

//它被指向的类型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; //指向运算符，或者可以这们(\*ptr).a,建议使用前者

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: (将结构体换成数组)

例十三：

int array[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。

不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。

# 7、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。

int fun1(char \*,int);

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

pfun1=fun1;

int a=(\*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。

例十四：

int fun(char \*);

inta;

char str[]="abcdefghijklmn";

a=fun(str);

int fun(char \*s)

{

int num=0;

for(int i=0;;)

{

num+=\*s;s++;

}

return num;

}

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。

# 8、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。

例十五：

float f=12.3;

float \*fptr=&f;

int \*p;

在上面的例子中，假如我们想让指针p 指向实数f，应该怎么办？

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。（切记）

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，必须保证类型一致，否则需要强制转换

例十六：

void fun(char\*);

int a=125,b;

fun((char\*)&a);

void fun(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：

unsigned int a;

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

ptr=a; //我们的目的是要使指针ptr 指向地址20345686

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：

unsigned int a;

TYPE \*ptr; //TYPE 是int，char 或结构类型等等类型。

a=N //N 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：

例十七：

int a=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。

str=(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。

# 9、指针的安全问题

看下面的例子：

例十八：

char s='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：

例十九：

char a;

int \*ptr=&a;

ptr++;

\*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。

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，读者结合例十八来想一想，应该会明

白的。

# 10、结束语

现在你是否已经觉得指针再也不是你所想的那么害怕了，如果你的回

答是：对，我不怕了！哈哈，恭喜你，你已经掌握C 语言的精华了，C

中唯一的难点就是指针，指针搞定其它小菜而已，重要的是实践，好

吧，让我们先暂停C 的旅程吧，开始我们的C++编程，C 是对底层操作

非常方便的语言，但开发大型程序本人觉得还是没有C++方便，至少维

护方面不太好做。而且C++是面向对象的语言，现在基本已经是面向对

象的天下了，所以建议学C++。C++是一门难学易用的语言，要真正掌

握C++可不是那么容易的，将基本的学完后，就学数据结构吧，算法才

是永恒的，程序设计语言层出不穷，永远学不完。学完之后就认真啃

下STL 这根骨头吧，推荐书籍--------范型编程与STL 和STL 源码剖

析。如果你达到了这样要求，再一次恭喜你，你已经是个程序高手了，

甚至可以说是个算法高手，因为STL 里有大量的精华而高效的算法。

唉，已经该说再见的时候了，让我们一起用我们的语言来谱写我们的

人生吧，最后笑个，哈哈，睡觉了。好累，都2：00 了\_\_