彻底搞定 C 指针

（完全版・修订增补版）

著＝姚云飞

修订＝丁正宇

前言

姚云飞先生的大作《彻底搞定 C 指针》是互联网上中文 C/C++界内为数不多的专门阐述 C 指针问题的优秀文献资源之一。

正如书名所示，对于那些学习了 C 基础知识却始终对 C 指针不得要领的读者，或者那些已经长期被 C 指针困扰的读者，作者致力于彻底解决他们在这方面的难题。为了达到这个目的，作者运用了许多生动与亲切的例子，深入浅出地讲透了 C 指针的原理与机制，并辅以编程实践中最常用的惯例和技巧作为示范。

《彻底搞定 C 指针》是互联网上下载次数最多的针对 C 指针问题的中文资源之一。现在，经由修订者的重新修订、编辑与排版，本书的《完全版・修订增补版》全新登场。新版本中的技术用语更加清楚严谨，行文的结构层次更加分明，例子中的程序代码均通过编译以测试其精准性。修订者希望这份新的成果能够令各位读者在 C 编程方面获得更多的益处，同时也期待着读者们宝贵的反馈信息。

再次向姚云飞先生致敬！

目 录

修订说明

A类：规范化

A1. C 程序的代码段，以及行文中的代码的字体，均统一调整为 Courier New，例如：

- 类型说明符“int”、变量名“a”、地址表达式“&a”、函数名“Exchg1”等等均作调整。

A2. 行为中的代码段，按一般行文处理缩进；代码段内部规整缩进。

A3. 规整 C 语句，例如：

- 语句中形如“a=b+c(x,y)”的，将调整为形如“a = b + c(x, y)”的新样式，即在运算符、用来间隔参数的逗号等的旁边补足空白，令语句的可读性更强。
- 补全语句结尾的“;”。

A4. 规整行文语序，令其更加通顺。

A5. 规整术语写法，例如：

- “C、C++”调整为“C/C++”。

B类：更正

B1. 更正术语，例如：

- “申明”调整为“声明”。

B2. 规整 C 技术用语，例如：

- “一个声明一整型指针变量的语句”调整为“一条声明一个指向整型变量的指针的语句”。

B3. 规整 C 程序，例如：

- 补全定义函数时的类型说明符“void”。

- 补全 `main()` 程序段中的“`return(0);`”。

B4. 规整行文，例如：

- “真正有意义上的指针”调整为“具有真正‘指针’意义的变量”。

B5. 更正标点符号，例如：

- 将行文里面中文/英文标点符号（全角/半角）混用、前后抵牾的情况进行更正。
- 将程序里面有编程代码意义的符号（如双引号“""”）中被错误地录入为中文标点符号（全角）的，调整为英文（半角）的。

B6. 更正一些外语行文。

C类：明晰化

C1. 初次介绍（不一定是初次出现）专业术语时，用**黑体字**。

C2. 需要突出重点的地方，用**粗体字**。

C3. 重整程序，例如：

- 原作中某处的例 1 中的函数被定义名为“`Exchg1()`”，例 2 中的函数被定义名为“`Exchg2()`”，那么，将例 3 中的函数名在定义中调整为“`Exchg3()`”，使它们的逻辑关系更为明晰，易于读者阅读和理解。
- 循环体中的“`printf("%d", a[i]);`”调整为“`printf("%d\n", a[i])`”。

C4. 规整行文分段，令其更合乎逻辑。

D类：编译器

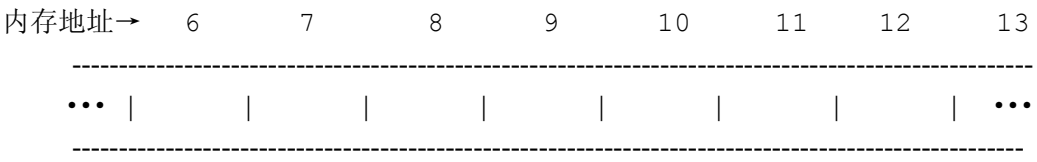
D1. 著者声明相关代码“都是在 VC6.0 上实验”，而修订者则是使用 gcc 3.4.2 编译器测试相关代码。

第壹篇 变量的内存实质

1．先来理解C语言中变量的实质

要理解 C 指针，我认为一定要理解 C 中“变量”的存储实质，所以我就从“变量”这个东西开始讲起吧！

先来理解理解内存空间吧！请看下图：



如上图所示，内存只不过是一个存放数据的空间，就好像我的看电影时的电影院中的座位一样。电影院中的每个座位都要编号，而我们的内存要存放各种各样的数据，当然我们要知道我们的这些数据存放在什么位置吧！所以内存也要象座位一样进行编号了，这就是我们所说的**内存编址**。座位可以是遵循“一个座位对应一个号码”的原则，从“第 1 号”开始编号。而内存则是按一个字节接着一个字节的次序进行编址，如上图所示。每个字节都有个编号，我们称之为**内存地址**。好了，我说了这么多，现在你能理解内存空间这个概念吗？

我们继续看看以下的 C/C++语言变量声明：

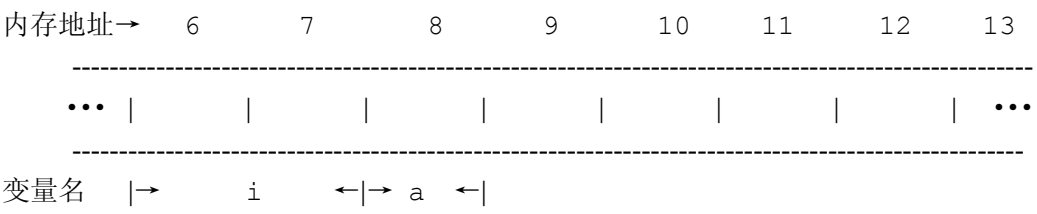
```
int i;
char a;
```

每次我们要使用某变量时都要事先这样**声明**它，它其实是内存中申请了一个名为 i 的整型变量宽度的空间（DOS 下的 16 位编程中其宽度为 2 个字节），和一个名为 a 的字符型变量宽度的空间（占 1 个字节）。

我们又如何来理解变量是如何存在的呢。当我们如下声明变量时：

```
int i;
char a;
```

内存中的映象可能如下图：



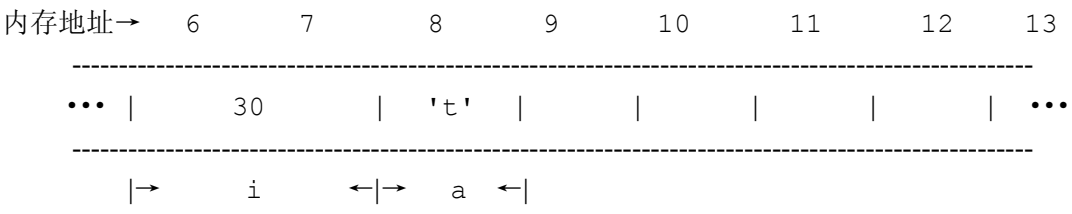
图中可看出，i 在内存起始地址为 6 上申请了两个字节的空间（我这里假设了 int 的宽度为 16 位，不同系统中 int 的宽度可能是不一样的），并命名为 i。a 在内存地址为 8 上申请了一字节的空间，并命名为 a。这样我们就有两个不同类型的变量了。

2. 赋值给变量

再看下面赋值：

```
i = 30;
a = 't';
```

你当然知道个两个语句是将 30 存入 i 变量的内存空间中，将“t”字符存入 a 变量的内存空间中。我们可以利用这样的形象来理解啦：



3．变量在哪里？（即我想知道变量的地址）

好了，接下来我们来看看&i 是什么意思？

是取 i 变量所在的地址编号嘛！我们可以这样读它：返回 i 变量的地址编号。你记住了吗？

我要在屏幕上显示变量的地址值的话，可以写如下代码：

```
printf("%x", &i);
```

以上图的内存映象为例，屏幕上显示的不是 i 值 30，而是显示 i 的内存地址编号 6 了。当然，在你的实际操作中，i 变量的地址值不会是这个数了。

这就是我所认为的作为初学者应该能够想象到的变量存储的实质了。请这样理解吧！

最后总结代码如下：

```
main()
{
    int i = 39;
    printf("%d\n", i);    /*①*/
    printf("%d\n", &i);   /*②*/
    return(0);
}
```

现在你可知道①、②两个 printf 分别在屏幕上输出的是 i 的什么东西啊?

好啦！下面我们就开始真正进入指针的学习了。

第贰篇 指针是什么？

1．指针是什么东西

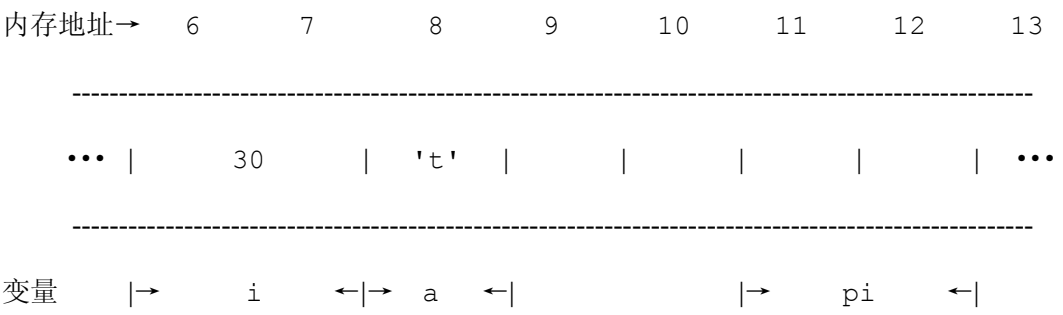
指针，想说弄懂你不容易啊！我们许多初学指针的人都要这样感慨。我常常在思索它，为什么呢？其实生活中处处都有指针，我们也处处在使用它。有了它我们的生活才更加方便了。没有指针，那生活才不方便。不信？你看下面的例子。

这是一个生活中的例子：比如说你要我借给你一本书，我到了你宿舍，但是你人不在宿舍，于是我把书放在你的 2 层 3 号的书架上，并写了一张纸条放在你的桌上。纸条上写着：你要的书在第 2 层 3 号的书架上。当你回来时，看到这张纸条，你就知道了我借与你的书放在哪了。你想想看，这张纸条的作用，纸条本身不是书，它上面也没有放着书。那么你又如何知道书的位置呢？因为纸条上写着书的**位置**嘛！其实这张纸条就是一个指针了。它上面的内容不是书本身，而是书的**地址**，你通过纸条这个指针找到了我借给你的这本书。

那么我们 C/C++中的指针又是什么呢？请继续跟我来吧，下面看一条声明一个指向整型变量的指针的语句：

```
int *pi;
```

pi 是一个指针，当然我们知道啦，但是这样说，你就以为 pi 一定是个多么特别的东西了。其实，它也只过是一个变量而已。与上一篇中说的变量并没有实质的区别。不信你看下面图：

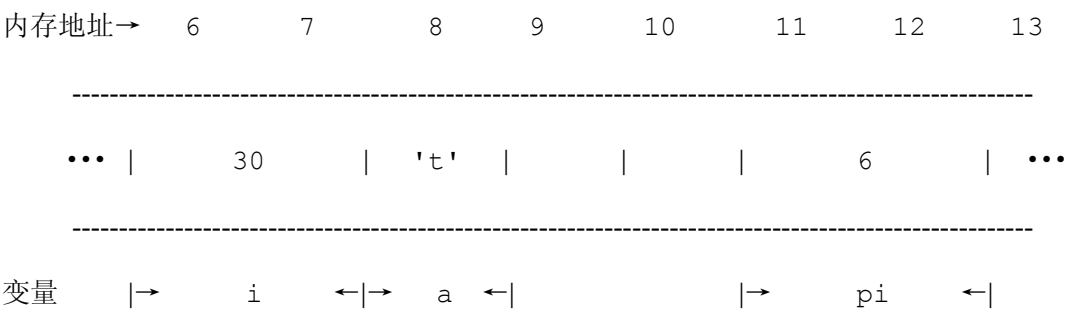


（说明：这里我假设了指针只占 2 个字节宽度，实际上在 32 位系统中，指针的宽度是 4 个字节宽的，即 32 位。）

由图示中可以看出，我们使用“int *pi”声明**指针变量** —— 其实是在内存的某处声明一个一定宽度的内存空间，并把它命名为 pi。你能在图中看出 pi 与前面的 i、a 变量有什么本质区别吗？没有，当然没有！pi 也只不过是一个变量而已嘛！那么它又为什么会被称为“指针”？关键是我们要让这个变量所存储的内容是什么。现在我要让 pi 成为具有真正“指针”意义的变量。请接着看下面语句：

```
pi = &i;
```

你应该知道 &i 是什么意思吧！再次提醒你啦：这是返回 i 变量的地址编号。整句的意思就是把 i 地址的编号赋值给 pi，也就是你在 pi 里面写上 i 的地址编号。结果如下图所示：



你看，执行完 pi=&i 后，在图示中的内存中，pi 的值是 6。这个 6 就是 i 变量的地址编号，这样 pi 就指向了变量 i 了。你看，pi 与那张纸条有什么区

别？pi 不就是那张纸条嘛！上面写着 i 的地址，而 i 就是那个本书。你现在看懂了吗？因此，我们就把 pi 称为指针。所以你要记住，指针变量所存的内容就是内存的地址编号！好了，现在我们就可以通过这个指针 pi 来访问到 i 这个变量了，不是吗？看下面语句：

```
printf("%d", *pi);
```

那么*pi 什么意思呢？你只要这样读它：pi 的内容所指的地址的内容（嘻嘻，看上去好像在绕口令了），就是 pi 这张“纸条”上所写的位置上的那本 “书”——i 。你看，Pi 的内容是 6，也就是说 pi 指向内存编号为 6 的地址。*pi 嘛，就是它所指地址的内容，即地址编号 6 上的内容了，当然就是 30 这个“值”了。所以这条语句会在屏幕上显示 30。也就是说 printf("%d", *pi)等价于 printf("%d", i) ，请结合上图好好体会吧！各位还有什么疑问？

到此为止，你掌握了类似&i、*pi 写法的含义和相关操作吗？总的一句话，我们的纸条就是我们的指针，同样我们的 pi 也就是我们的纸条！剩下的就是我们如何应用这张纸条了。最后我给你一道题：程序如下。

```
char  a,*pa;
a = 10;
pa = &a;
*pa = 20;
printf("%d", a);
```

你能直接看出输出的结果是什么吗？如果你能，我想本篇的目的就达到了。好了，就说到这了。Happy Study! 在下篇中我将谈谈“指针的指针”即对

```
int **ppa;
```

中 ppa 的理解。

第叁篇 指针与数组名

1. 通过数组名访问数组元素

看下面代码：

```
int i, a[] = {3,4,5,6,7,3,7,4,4,6};
for (i = 0; i <= 9; i++)
{
   printf("%d\n", a[i]);
}
```

很显然，它是显示 a 数组的各元素值。

我们还可以这样访问元素，如下：

```
int i, a[] = {3,4,5,6,7,3,7,4,4,6};
for (i = 0; i <= 9; i++)
{
   printf("%d\n", *(a+i));
}
```

它的结果和作用完全一样。

2. 通过指针访问数组元素

```
int i, *pa, a[] = {3,4,5,6,7,3,7,4,4,6};
pa = a; /*请注意数组名 a 直接赋值给指针 pa*/
for (i = 0; i <= 9; i++)
```

```
{
   printf("%d\n", pa[i]);
}
```

很显然，它也是显示 a 数组的各元素值。

另外与数组名一样也可如下：

```
int i, *pa, a[] = {3,4,5,6,7,3,7,4,4,6};
pa = a;
for (i = 0; i <= 9; i++)
{
   printf("%d\n", *(pa+i));
}
```

看 pa = a，即**数组名赋值给指针**，以及通过数组名、指针对元素的访问形式看，它们并没有什么区别，从这里可以看出：数组名其实也就是指针。难道它们没有任何区别？有，请继续。

3．数组名与指针变量的区别

请看下面的代码：

```
int i, *pa, a[] = {3,4,5,6,7,3,7,4,4,6};
pa = a;
for (i = 0; i <= 9; i++)
{
    printf("%d\n", *pa);
    pa++;  /*注意这里，指针值被修改*/
}
```

可以看出，这段代码也是将数组各元素值输出。不过，你把循环体{}中的 pa

改成 a 试试。你会发现程序编译出错，不能成功。看来指针和数组名还是不同的。其实上面的指针是指针变量，而**数组名只是一个指针常量**。这个代码与上面的代码不同的是，指针 pa 在整个循环中，其值是不断递增的，即指针值被修改了。数组名是指针常量，其值是不能修改的，因此不能类似这样操作：a++。

前面 4、5 节中 pa[i]，*(pa+i)处，指针 pa 的值是使终没有改变。所以变量指针 pa 与数组名 a 可以互换。

4. 声明指针常量

再请看下面的代码：

```
int i, a[] = {3,4,5,6,7,3,7,4,4,6};
int *const pa = a; /* 注意 const 的位置：不是 const int *pa */
for (i = 0; i <= 9; i++)
{
   printf("%d\n", *pa);
   pa++ ;  /*注意这里，指针值被修改*/
}
```

这时候的代码能成功编译吗？不能。因为 **pa 指针被定义为常量指针了**。这时与数组名 a 已经没有不同。这更说明了数组名就是常量指针。但是……

```
int *const a = {3,4,5,6,7,3,7,4,4,6}; /*不行*/
int a[]={3,4,5,6,7,3,7,4,4,6}; /*可以，所以初始化数组时必定要这样。*/
```

以上都是在 VC6.0 上实验。

第肆篇const int *pi与int *const pi的区别

1. 从const int i 说起

你知道我们声明一个变量时象这样 int i ；这个 i 是可能在它处重新变赋值的。如下：

```
int i = 0;
/* . . . */
i = 20; /*这里重新赋值了*/
```

不过有一天我的程序可能需要这样一个变量（暂且称它变量），在声明时就赋一个初始值。之后我的程序在其它任何处都不会再去重新对它赋值。那我又应该怎么办呢？用 const 。

```
/* . . . */
const int ic =20;
/* . . . */
ic = 40; /*这样是不可以的，编译时是无法通过，因为我们不能对 const 修饰的 ic 重新赋值的。*/
/*这样我们的程序就会更早更容易发现问题了。*/
/* . . . */
```

有了 const 修饰的 ic 我们不称它为变量，而称**符号常量**，代表着 20 这个数。这就是 const 的作用。ic 是不能在它处重新赋新值了。

认识了 const 作用之后，另外，我们还要知道格式的写法。有两种：

```
const int ic = 20;
```

与

```
int const ic = 20;
```

它们是完全相同的。这一点我们是要清楚。总之，你务必要记住 const 与 int 哪个写前都不影响语义。有了这个概念后，我们来看这两个家伙：

```
const int *pi
```

与

```
int const *pi
```

按你的逻辑看，它们的语义有不同吗？呵呵，你只要记住一点：**int 与 const 哪个放前哪个放后都是一样的**，就好比 const int ic;与 int const ic;一样。也就是说，它们是相同的。

好了，我们现在已经搞定一个“双包胎”的问题。那么

```
int *const pi;
```

与前两个语句又有什么不同呢？我下面就来具体分析它们的格式与语义吧！

2. const int *pi的语义

我先来说说 const int *pi 是什么作用 （当然 int const *pi 也是一样的，前面我们说过，它们实际是一样的）。看下面的例子：

```
/* 代码开始 */

int i1 = 30;

int i2 = 40;

const int *pi = &i1;

pi = &i2;     /* 注意这里，pi 可以在任意时候重新赋值一个新内存地址*/

i2 = 80;     /* 想想看：这里能用*pi = 80 来代替吗？当然不能！ */
```

```
printf("%d\n", *pi);  /* 输出是 80 */

/* 代码结束 */
```

语义分析：

看出来了没有啊，pi 的值是可以被修改的。即它可以重新指向另一个地址的，但是，不能通过*pi 来修改 i2 的值。这个规则符合我们前面所讲的逻辑吗？当然符合了！

首先 const 修饰的是整个*pi（注意，我写的是*pi 而不是 pi）。所以*pi 是常量，是不能被赋值的（虽然 pi 所指的 i2 是变量，不是常量）。

其次，pi 前并没有用 const 修饰，所以 pi 是指针变量，能被赋值重新指向另一内存地址的。你可能会疑问：那我又如何用 const 来修饰 pi 呢？其实，你注意到 int *const pi 中 const 的位置就大概可以明白了。请记住，通过格式看语义。哈哈，你可能已经看出了规律吧？那下面的一节也就没必要看下去了。不过我还得继续我的战斗！

3. 再看int *const pi

确实，int *const pi 与前面的 int const *pi 会很容易给混淆的。注意：前面一句的 const 是写在 pi 前和*号后的，而不是写在*pi 前的。很显然，它是修饰限定 pi 的。我先让你看例子：

```
/* 代码开始 */

int i1 = 30;

int i2 = 40;

int *const pi = &i1;

/* pi = &i2;     注意这里，pi 不能再这样重新赋值了，即不能再指向
```

另一个新地址。（第 4 行的注释）*/

/* 所以我已经注释了它。*/

i1 = 80; /* 想想看：这里能用 *pi = 80; 来代替吗？可以，这里可以通过*pi 修改 i1 的值。（第 5 行的注释）*/

/* 请自行与前面一个例子比较。 */

printf("%d", *pi); /* 输出是 80 */

/* 代码结束 */

语义分析：

看了这段代码，你明白了什么？有没有发现 pi 值是不能重新赋值修改了。**它只能永远指向初始化时的内存地址了**。相反，这次你可以通过*pi 来修改 i1 的值了。与前一个例子对照一下吧！看以下的两点分析：

1）pi 因为有了 const 的修饰，所以只是一个指针常量：也就是说 pi 值是不可修改的（即 pi 不可以重新指向 i2 这个变量了）（请看第 4 行的注释）。

2）整个*pi 的前面没有 const 的修饰。也就是说，*pi 是变量而不是常量，所以我们可以通过*pi 来修改它所指内存 i1 的值（请看第 5 行的注释）。

总之一句话，**这次的 pi 是一个指向 int 变量类型数据的指针常量**。

我最后总结两句：

1） 如果 const 修饰在*pi 前，则不能改的是*pi（即不能类似这样：*pi=50;赋值）而不是指 pi。

2） 如果 const 是直接写在 pi 前，则 pi 不能改（即不能类似这样：pi=&i;赋值）。

请你务必先记住这两点，相信你一定不会再被它们给搞糊了。现在再看这两个声明语句 int const *pi 和 int *const pi 时，呵呵，你会头昏脑胀还是很轻松惬意？它们各自声明的 pi 分别能修改什么，不能修改什么？再问问

自己，把你的理解告诉我吧，可以发帖也可以发到我的邮箱（我的邮箱 yyf977@163.com)！我一定会答复的。

4．补充三种情况

这里，我再补充以下三种情况。其实只要上面的语义搞清楚了，这三种情况也就已经被包含了。不过作为三种具体的形式，我还是简单提一下吧！

情况一：int *pi 指针指向 const int i 常量的情况

```
/* begin */
const int i1 = 40;
int *pi;
pi = &i1;  /* 这样可以吗？不行，VC 下是编译错。*/
/* const int 类型的 i1 的地址是不能赋值给指向 int 类型地址的指针 pi 的。否则 pi 岂不是能修改 i1 的值了吗！*/
pi = (int *) &i1;   /*  这样可以吗？强制类型转换可是 C 所支持的。*/
/* VC 下编译通过，但是仍不能通过 *pi = 80 来修改 i1 的值。去试试吧！看看具体的怎样。*/
/* end */
```

情况二：const int *pi 指针指向 const int i1 的情况

```
/* begin */
const int i1=40;
const int * pi;
```

pi=&i1;/* 两个类型相同，可以这样赋值。很显然，i1 的值无论是通过 pi 还是 i1 都不能修改的。 */

/* end */

情况三：用 const int *const pi 声明的指针

/* begin */

int i;

const int * const pi=&i; /*你能想象 pi 能够作什么操作吗？pi 值不能改，也不能通过 pi 修改 i 的值。因为不管是*pi 还是 pi 都是 const 的。 */

/* end */

第伍篇 函数参数的传递

1．三道考题

开讲之前，我先请你做三道题目。（嘿嘿，得先把你的头脑搞昏才行……唉呀，谁扔我鸡蛋？）

考题一，程序代码如下：

```
void Exchg1(int x, int y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
   printf("x = %d, y = %d\n", x, y);
}
main()
{
   int a = 4,b = 6;
   Exchg1(a, b);
   printf("a = %d, b = %d\n", a, b);
   return(0);
}
```

输出的结果为：

x = ____, y=____.

a = ____, b=____.

问下划线的部分应是什么，请完成。

考题二，程序代码如下：

```
void Exchg2(int *px, int *py)
{
    int tmp = *px;
    *px = *py;
    *py = tmp;
    printf("*px = %d, *py = %d.\n", *px, *py);
}
main()
{
    int a = 4;
    int b = 6;
    Exchg2(&a, &b);
    printf("a = %d, b = %d.\n", a, b);
    return(0);
}
```

输出的结果为为：

*px=____, *py=____.

a=____, b=____.

问下划线的部分应是什么，请完成。

考题三，程序代码如下：

```
void Exchg3(int &x, int &y)
```

```
{
        int tmp = x;
        x = y;
        y = tmp;
        printf("x = %d,y = %d\n", x, y);
}
main()
{
        int a = 4;
        int b = 6;
        Exchg3(a, b);
        printf("a = %d, b = %d\n", a, b);
        return(0);
}
```

输出的结果为：

x=____, y=____.

a=____, b=____.

问下划线的部分应是什么，请完成。

你不在机子上试，能作出来吗？你对你写出的答案有多大的把握？

正确的答案，想知道吗？（呵呵，让我慢慢地告诉你吧!）

好，废话少说，继续我们的探索之旅了。

我们都知道：C 语言中函数参数的传递有：**值传递、地址传递、引用传递**这三种形式。题一为值传递，题二为地址传递，题三为引用传递。不过，正是这几种参数传递的形式，曾把我给搞得晕头转向。我相信也有很多人与我有同感吧？

下面请让我逐个地谈谈这三种传递形式。

2. 函数参数传递方式之一：值传递

（1）值传递的一个错误认识

先看考题一中 Exchg1 函数的定义：

```
void Exchg1(int x, int y)   /* 定义中的x,y变量被称为Exchg1函数的形式参数 */
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
    printf("x = %d, y = %d.\n", x, y);
}
```

问：你认为这个函数是在做什么呀？

答：好像是对参数 x、y 的值对调吧？

请往下看，我想利用这个函数来完成对 a,b 两个变量值的对调，程序如下：

```
main()
{
    int a = 4,b = 6;
    Exchg1(a, b);     /*a,b变量为Exchg1函数的实际参数。*/
    printf("a = %d, b = %d.\n", a, b);
    return(0);
}
```

我问：Exchg1()里头的 printf("x = %d, y = %d.\n", x, y);语

句会输出什么啊？

我再问：Exchg1()后的 printf("a = %d, b = %d.\n", a, b);语句输出的是什么？

程序输出的结果是：

x = 6, y = 4.

a = 4, b = 6.

为什么不是 a = 6,b = 4 呢？

奇怪，明明我把 a、b 分别代入了 x、y 中，并在函数里完成了两个变量值的交换，为什么 a、b 变量值还是没有交换（仍然是 a = 4、b = 6，而不是 a = 6、b = 4）？如果你也会有这个疑问，那是因为你根本就不知实参 a、b 与形参 x、y 的关系了。

（2）一个预备的常识

为了说明这个问题，我先给出一个代码：

```
int a = 4;
int x;
x = a;
x = x + 3;
```

看好了没，现在我问你：最终 a 值是多少，x 值是多少？

（怎么搞的，给我这个小儿科的问题。还不简单，不就是 a = 4、x = 7 嘛！）

在这个代码中，你要明白一个东西：虽然 a 值赋给了 x，但是 a 变量并不是 x 变量哦。我们对 x 任何的修改，都不会改变 a 变量。呵呵！虽然简单，并且一看就理所当然，不过可是一个很重要的认识喔。

（3）理解值传递的形式

看调用 Exch1 函数的代码：

```
main()
{
      int a = 4,b = 6;
      Exchg1(a, b) /* 这里调用了 Exchg1 函数 */
      printf("a = %d, b = %d.\n", a, b);
}
```

Exchg1(a, b)时所完成的操作代码如下所示。

```
int x = a; /* ← */
int y = b; /* ← 注意这里，头两行是调用函数时的隐含操作 */
int tmp;
tmp = x;
x = y;
y = tmp;
```

请注意在调用执行 Exchg1 函数的操作中我人为地加上了头两句：

```
int x = a;
int y = b;
```

这是调用函数时的两个隐含动作。它确实存在，现在我只不过把它**显式地**写了出来而已。问题一下就清晰起来啦。（看到这里，现在你认为函数里面交换操作的是 a、b 变量或者只是 x、y 变量呢？）

原来 ，其实函数在调用时是隐含地把实参 a、b 的值分别赋值给了 x、y，之后在你写的 Exchg1 函数体内再也没有对 a、b 进行任何的操作了。交换的只是 x、y 变量。并不是 a、b。当然 a、b 的值没有改变啦！函数只是把 a、b 的值通过赋值传递给了 x、y，函数里头操作的只是 x、y 的值并不是 a、b 的值。这就是所谓的参数的值传递了。

哈哈，终于明白了，正是因为它隐含了那两个的赋值操作，才让我们产生

了前述的迷惑（以为 a、b 已经代替了 x、y，对 x、y 的操作就是对 a、b 的操作了，这是一个错误的观点啊!）。

3. 函数参数传递方式之二：地址传递

继续！地址传递的问题！

看考题二的代码：

```
void Exchg2(int *px, int *py)
{
      int tmp = *px;
      *px = *py;
      *py = tmp;
      printf("*px = %d, *py = %d.\n", *px, *py);
}
main()
{
      int a = 4;
      int b = 6;
      Exchg2(&a, &b);
      printf("a = %d, b = %d.\n”, a, b);
      return(0);
}
```

它的输出结果是：

```
  *px = 6, *py = 4.
  a = 6, b = 4.
```

看函数的**接口**部分：`Exchg2(int *px, int *py)`，请注意：参数 px、

py 都是指针。

再看调用处：Exchg2(&a, &b);

它将 a 的地址（&a）代入到 px，b 的地址（&b）代入到 py。同上面的值传递一样，函数调用时作了两个隐含的操作：将&a，&b 的值赋值给了 px、py。

```
px = &a;
py = &b;
```

呵呵！我们发现，其实它与值传递并没有什么不同，只不过这里是将 a、b 的地址值传递给了 px、py，而不是传递的 a、b 的内容，而（请好好地在比较比较啦）整个 Exchg2 函数调用是如下执行的：

```
px = &a;   /* ← */
py = &b;   /* ← 请注意这两行，它是调用 Exchg2 的隐含动作。*/
int tmp = *px;
*px = *py;
*py = tmp;
printf("*px =%d, *py = %d.\n", *px, *py);
```

这样，有了头两行的隐含赋值操作。我们现在已经可以看出，指针 px、py 的值已经分别是 a、b 变量的地址值了。接下来，对*px、*py 的操作当然也就是对 a、b 变量本身的操作了。所以函数里头的交换就是对 a、b 值的交换了，这就是所谓的地址传递（传递 a、b 的地址给了 px、py），你现在明白了吗？

4. 函数参数传递方式之三：引用传递

看题三的代码：

```
void Exchg3(int &x, int &y) /* 注意定义处的形式参数的格式与值传递不同 */
{
    int tmp = x;
```

```
        x = y;
        y = tmp;
        printf("x = %d, y = %d.\n", x, y);
}
main()
{
  int a = 4;
  int b = 6;
     Exchg3(a, b);  /*注意：这里调用方式与值传递一样*/
     printf("a = %d, b = %d.\n", a, b);
}
```

输出结果：

```
x = 6, y = 4.
a = 6, b = 4.   /*这个输出结果与值传递不同。*/
```

看到没有，与值传递相比，代码格式上只有一处是不同的，即在定义处：

```
Exchg3(int &x, int &y)
```

但是我们发现 a 与 b 的值发生了对调。这说明了 Exchg3(a, b)里头修改的是 a、b 变量，而不只是修改 x、y 了。

我们先看 Exchg3 函数的定义处 Exchg3(int &x, int &y)。参数 x、y 是 int 的变量，调用时我们可以像值传递（如： Exchg1(a, b); ）一样调用函数（如： Exchg3(a, b);）。但是 x、y 前都有一个取地址符号“&”。有了这个，调用 Exchg3 时函数会将 a、b 分别代替了 x、y 了，我们称：**x、y 分别引用了 a、b 变量。**这样函数里头操作的其实就是实参 a、b 本身了，也就是说函数里是可以直接修改到 a、b 的值了。

最后对值传递与引用传递作一个比较：

1）在函数定义格式上有不同：

值传递在定义处是：`Exchg1(int x, int y);`

引用传递在这义处是：`Exchg3(int &x, int &y);`

2）调用时有相同的格式：

值传递：`Exchg1(a, b);`

引用传递：`Exchg3(a, b);`

3）功能上是不同的：

值传递的函数里操作的不是 a、b 变量本身，只是将 a、b 值赋给了 x、y。函数里操作的只是 x、y 变量而不是 a、b，显示 a、b 的值不会被 Exchg1 函数所修改。

引用传递 Exchg3(a, b)函数里是用 a、b 分别代替了 x、y。函数里操作的就是 a、b 变量的本身，因此 a、b 的值可在函数里被修改的。

第陆篇 指向另一指针的指针

1. 回顾指针概念

早在本书第贰篇中我就对指针的实质进行了阐述。今天我们又要学习一个叫做**“指向另一指针地址”的指针**。让我们先回顾一下指针的概念吧！

当我们程序如下声明变量：

```
short int i;
char a;
short int * pi;
```

程序会在内存某地址空间上为各变量开辟空间，如下图所示：

```
内存地址→ 5      6      7       8        9       10      11     12    13
    ---------------------------------------------------------------------------------------------
   … |      |      |       |       |       |       |       |      |      |
    ---------------------------------------------------------------------------------------------
      |short int i|char a|       |short int * pi |
```

图中所示中可看出：

i 变量在内存地址 5 的位置，占 2 个字节。

a 变量在内存地址 7 的位置，占 1 个字节。

pi 变量在内存地址 9 的位置，占 2 个字节。(注：pi 是指针，我这里指针的宽度只有 2 个字节，32 位系统是 4 个字节)

接下来如下赋值：

```
i = 50;
pi = &i;
```

经过上在两句的赋值，变量的内存映象如下：

```
内存地址→ 5     6     7     8       9     10     11    12   13
     ------------------------------------------------------------------------------------------
    … |    50     |     |     |     5       |     |     |
     ------------------------------------------------------------------------------------------
       |short int i|char a|     |short int * pi|
```

看到没有：短整型指针变量 pi 的值为 5，它就是 i 变量的内存起始地址。所以，这时当我们对*pi 进行读写操作时，其实就是对 i 变量的读写操作。如：

```
*pi=5;   /* 就是等价于 i = 5; */
```

你可以回看本书的第贰篇，那里有更加详细的解说。

2. 指针的地址与指向另一指针地址的指针

在上一节中，我们看到，指针变量本身与其它变量一样也是在某个内存地址中的，如 pi 的内存起始地址是 9。同样的，我们也可能让某个指针指向这个地址。

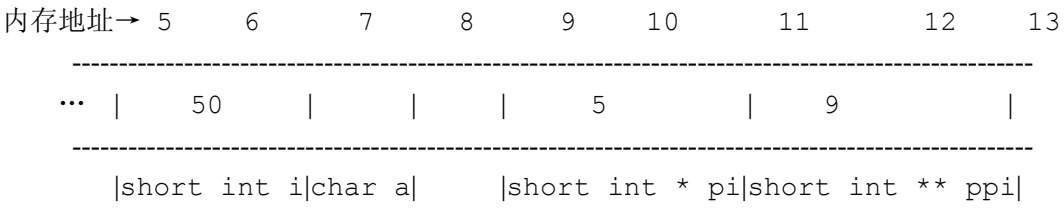
看下面代码：

```
short int **ppi; /* 这是一个指向指针的指针，注意有两个“*”号 */

*ppi = &pi;
```

第一句：short int **ppi; —— **声明了一个指针变量 ppi，这个 ppi 是用来存储（或称指向）一个 short int * 类型指针变量的地址。**

第二句：&pi 那就是取 pi 的地址，**ppi = &pi 就是把 pi 的地址赋给了 ppi。即将地址值 9 赋值给 ppi。如下图：



从图中看出，指针变量 ppi 的内容就是指针变量 pi 的起始地址。于是……

ppi 的值是多少呢？—— 9。

*ppi 的值是多少呢？—— 5，即 pi 的值。

**ppi 的值是多少呢？—50，即 i 的值，也是*pi 的值。

呵呵！不用我说太多了，我相信你应明白这种指针了吧！

3. 一个应用实例

（1）设计一个函数：void find1(char array[], char search, char *pa)

要求：这个函数参数中的数组 array 是以 0 值为结束的字符串，要求在字符串 array 中查找字符是参数 search 里的字符。如果找到，函数通过第三个参数（pa）返回值为 array 字符串中第一个找到的字符的地址。如果没找到，则为 pa 为 0。

设计：依题意，实现代码如下。

```
void find1(char array[], char search, char *pa)
{
      int i;
      for (i = 0; *(array + i) != 0; i++)
      {
         if ( *(array+i) == search)
        {
            pa = array + i;
```

```
            break;
        }
        else if (*(array+i) == 0)
        {
            pa = 0;
            break;
        }
    }
}
```

你觉得这个函数能实现所要求的功能吗？

调试：

我下面调用这个函数试试。

```
main()
{
    char str[] = {"afsdfsdfdf\0"};  /* 待查找的字符串 */
    char a = 'd';   /* 设置要查找的字符 */
    char *p = 0;  /* 如果查找到后指针 p 将指向字符串中查找到
  的第 1 个字符的地址。 */
    find1(str, a, p);  /* 调用函数以实现所要操作。 */
    if (0 == p)
    {
       printf("没找到! \n"); /* 如果没找到则输出此句 */
    }
    else
    {
       printf("找到了, p = %d", p);  /* 如果找到则输出此句
   */
```

```
        }
        return(0);
    }
```

分析：

上面代码，你认为会是输出什么呢？

运行试试。

唉！怎么输出的是：没有找到！

而不是“找到了，……”。

明明 a 值为'd'，而 str 字符串的第四个字符是'd'，应该找得到呀！

再看函数定义处：void find1(char array[], char search, char *pa)

看调用处：find1(str, a, p);

依我在第伍篇的分析方法，函数调用时会对每一个参数进行一个隐含的赋值操作。

整个调用如下：

```
    array = str;
    search = a;
    pa = p;    /* 请注意：以上三句是调用时隐含的动作。*/
    int i;
    for(i =0; *(array+i) != 0; i++)
    {
        if (*(array+i) == search)
        {
           pa = array + i;
           break;
        }
        else if (*(array+i)==0)
        {
```

```
            pa=0;
            break;
        }
    }
```

哦！参数 pa 与参数 search 的传递并没有什么不同，都是值传递嘛（小语：地址传递其实就是地址值传递嘛)！所以对形参变量 pa 值（当然值是一个地址值）的修改并不会改变实参变量 p 值，因此 p 的值并没有改变(即 p 的指向并没有被改变)。

（如果还有疑问，再看一看《第五篇：函数参数的传递》了。）

修正：

```
    void find2(char array[], char search, char **ppa)
    {
        int i;
        for (i=0; *(array + i) != 0; i++)
        {
            if(*(array + i) == search)
            {
                *ppa = array + i;
                break;
            }
            else if(*(array + i) == 0)
            {
                *ppa = 0;
                break;
            }
        }
    }
```

主函数的调用处改如下：

```
find2(str, a, &p);  /*调用函数以实现所要操作。*/
```

再分析：

这样调用函数时的整个操作变成如下：

```
array = str;
search = a;
ppa = &p;    /* 请注意：以上三句是调用时隐含的动作。 */
int i;
for (i = 0; *(array + i) != 0; i++)
{
      if (*(array + i) == search)
      {
         *ppa = array + i
         break;
      }
      else if (*(array+i)==0)
      {
         *ppa=0;
         break;
   }
}
```

看明白了吗？

ppa 指向指针 p 的地址。

对*ppa 的修改就是对 p 值的修改。

你自行去调试。

经过修改后的程序就可以完成所要的功能了。

看懂了这个例子，也就达到了本篇所要求的目的。

第柒篇 函数名与函数指针

1. 通常的函数调用

一个通常的函数调用的例子：

```
/* 自行包含头文件 */
void MyFun(int x); /* 此处的声明也可写成：void MyFun(int) */
int main(int argc, char* argv[])
{
     MyFun(10);     /* 这里是调用 MyFun(10) 函数 */

     return(0);
}
void MyFun(int x)  /* 这里定义一个 MyFun 函数 */
{
     printf("%d\n",x);
}
```

这个 MyFun 函数是一个无返回值的函数，它并不“完成”什么事情。这种调用函数的格式你应该是很熟悉的吧!看主函数中调用 MyFun 函数的书写格式：

```
MyFun(10);
```

我们一开始只是从功能上或者说从数学意义上理解 MyFun 这个函数，知道 MyFun 函数名代表的是一个功能（或是说一段代码）。

直到——学习到函数指针概念时。我才不得不在思考：函数名到底又是什么东西呢？

（不要以为这是没有什么意义的事噢！呵呵，继续往下看你就知道了。）

2．函数指针变量的声明

就象某一数据变量的内存地址可以存储在相应的指针变量中一样，**函数的首地址**也以存储在某个函数指针变量里的。这样，我就可以通过这个函数指针变量来调用所指向的函数了。

在 C 系列语言中，任何一个变量，总是要先声明，之后才能使用的。那么，函数指针变量也应该要先声明吧？那又是如何来声明呢？以上面的例子为例，我来声明一个可以指向 MyFun 函数的函数指针变量 FunP。下面就是声明 FunP 变量的方法：

```
void (*FunP)(int) ;   /* 也可写成 void (*FunP)(int x)*/
```

你看，整个函数指针变量的声明格式如同函数 MyFun 的声明处一样，只不过—我们把 MyFun 改成“(*FunP)”而已，这样就有了一个能指向 MyFun 函数的指针 FunP 了。（当然，这个 FunP 指针变量也可以指向所有其它具有相同参数及返回值的函数了。）

3．通过函数指针变量调用函数

有了 FunP 指针变量后，我们就可以对它赋值指向 MyFun，然后通过 FunP 来调用 MyFun 函数了。看我如何通过 FunP 指针变量来调用 MyFun 函数的：

```
/* 自行包含头文件 */
void MyFun(int x);  /* 这个声明也可写成：void MyFun( int )*/
void (*FunP)(int ); /*也可声明成 void(*FunP)(int x)，但习惯上一般不这样。 */
int main(int argc, char* argv[])
{
```

```
    MyFun(10);      /* 这是直接调用 MyFun 函数 */
    FunP = &MyFun;  /* 将 MyFun 函数的地址赋给 FunP 变量  */
    (*FunP)(20);    /* （★）这是通过函数指针变量 FunP 来调用
 MyFun 函数的。  */
}

void MyFun(int x)  /* 这里定义一个 MyFun 函数 */
{
    printf("%d\n",x);
}
```

请看（★）行的代码及注释。

运行看看。嗯，不错，程序运行得很好。

哦，我的感觉是：MyFun 与 FunP 的类型关系类似于 int 与 int * 的关系。函数 MyFun 好像是一个如 int 的变量（或常量），而 FunP 则像一个如 int * 一样的指针变量。

```
int i,*pi;
pi = &i;    /* 与 FunP = &MyFun 比较。*/
```

（你的感觉呢？）

呵呵，其实不然……

4. 调用函数的其它书写格式

函数指针也可如下使用，来完成同样的事情：

```
/* 自行包含头文件 */
void MyFun(int x);
void (*FunP)(int );/* 声明一个用以指向同样参数，返回值函数的
```

指针变量。 */

```
int main(int argc, char* argv[])
{
     MyFun(10);     /* 这里是调用 MyFun(10)函数 */
     FunP = MyFun;  /* 将 MyFun 函数的地址赋给 FunP 变量 */
     FunP(20); /* （★）这是通过函数指针变量来调用 MyFun 函数
 的。*/
     return 0;
}
void MyFun(int x)  //这里定义一个 MyFun 函数
{
     printf("%d\n",x);
}
```

我改了（★）行（请自行与之前的代码比较一下）。

运行试试，啊！一样地成功。

咦？

```
FunP = MyFun;
```

可以这样将 MyFun 值同赋值给 FunP，**难道 MyFun 与 FunP 是同一数据类型（即如同的 int 与 int 的关系），而不是如同 int 与 int*的关系了？**（有没有一点点的糊涂了？）

看来与之前的代码有点矛盾了，是吧！所以我说嘛！

请容许我暂不给你解释，继续看以下几种情况（这些可都是可以正确运行的代码哟!）：

代码之三：

```
int main(int argc, char* argv[])
{
```

```
        MyFun(10);      /* 这里是调用 MyFun(10)函数 */
        FunP = &MyFun;  /* 将 MyFun 函数的地址赋给 FunP 变量 */
        FunP(20); /* 这是通过函数指针变量来调用 MyFun 函数的。 */
        return 0;
    }
```

代码之四：

```
    int main(int argc, char* argv[])
    {
        MyFun(10);      /* 这里是调用 MyFun(10)函数 */
        FunP = MyFun;  /* 将 MyFun 函数的地址赋给 FunP 变量 */
        (*FunP)(20); /*这是通过函数指针变量来调用 MyFun 函数的。
     */
        return 0;
    }
```

真的是可以这样的噢！

（哇！真是要晕倒了!）

还有呐！看——

```
    int main(int argc, char* argv[])
    {
      (*MyFun)(10);   /*看，函数名 MyFun 也可以有这样的调用格式*/
      return 0;
    }
```

你也许第一次见到吧：函数名调用也可以是这样写的啊！（只不过我们平常没有这样书写罢了。）

那么，这些又说明了什么呢？

呵呵！依据以往的知识和经验来推理本篇的“新发现”，我想就连“福尔摩斯”也必定会由此分析并推断出以下的结论：

1）其实，MyFun 的函数名与 FunP 函数指针都是一样的，即都是函数指针。MyFun 函数名是一个函数指针常量，而 FunP 是一个函数数指针变量，这是它们的关系。

2）但函数名调用如果都得如(*MyFun)(10)这样，那书写与读起来都是不方便和不习惯的。所以 C 语言的设计者们才会设计成又可允许 MyFun(10)这种形式地调用（这样方便多了并与数学中的函数形式一样，不是吗？）。

3）为统一起见，FunP 函数指针变量也可以 FunP(10)的形式来调用。

4）赋值时，即可 FunP = &MyFun 形式，也可 FunP = MyFun。

上述代码的写法，随便你爱怎么着！

请这样理解吧！这可是有助于你对函数指针的应用喽！

最后 ——

补充说明一点，在函数的声明处：

```
void MyFun(int);     /*不能写成 void (*MyFun)(int)。*/
void (*FunP)(int);   /*不能写成 void FunP(int)。*/
```

（请看注释）这一点是要注意的。

5. 定义某一函数的指针类型

就像自定义数据类型一样，我们也可以先定义一个函数指针类型，然后再用这个类型来声明函数指针变量。

我先给你一个自定义数据类型的例子。

```
typedef int* PINT;     /* 为 int* 类型定义了一个 PINT 的别名*/
int main()
{
    int x;
    PINT px = &x;   /* 与“int *px=&x;”是等价的。PINT 类型
```

其实就是 int * 类型 */

```
    *px = 10;       /* px 就是 int*类型的变量  */
    return 0;
}
```

根据注释，应该不难看懂吧！（虽然你可能很少这样定义使用，但以后学习 Win32 编程时会经常见到的。）

下面我们来看一下函数指针类型的定义及使用：（请与上对照！）

```
/* 自行包含头文件 */
void MyFun(int x); /*此处的声明也可写成：void MyFun( int )*/
typedef void (*FunType)(int);  /*（★）这样只是定义一个函数指针类型*/
FunType FunP;  /*然后用 FunType 类型来声明全局 FunP 变量*/
int main(int argc, char* argv[])
{
    FunType FunP;    /*函数指针变量当然也是可以是局部的 ，那就请在这里声明了。 */
    MyFun(10);
    FunP = &MyFun;
    return 0;
}
void MyFun(int x)
{
    printf("%d\n",x);
}
```

看（★）行：

首先，在 void (*FunType)(int)前加了一个 typedef 。这样只是定

义一个名为 FunType 函数指针类型，而不是一个 FunType 变量。

然后，“FunType FunP;”这句就如“PINT px;”一样地声明一个 FunP 变量。

其它相同。整个程序完成了相同的事。

这样做法的好处是：

有了 FunType 类型后，我们就可以同样地、很方便地用 FunType 类型来声明多个同类型的函数指针变量了。如下：

```
FunType FunP2;
FunType FunP3;
/* . . . */
```

6. 函数指针作为某个函数的参数

既然函数指针变量是一个变量，当然也可以作为某个函数的参数来使用的。所以，你还应知道函数指针是如何作为某个函数的参数来传递使用的。

给你一个实例：

要求：我要设计一个 CallMyFun 函数，这个函数可以通过参数中的函数指针值不同来分别调用 MyFun1、MyFun2、MyFun3 这三个函数（注：这三个函数的定义格式应相同）。

实现：代码如下：

```
/* 自行包含头文件 */
void MyFun1(int x);
void MyFun2(int x);
void MyFun3(int x);
typedef void (*FunType)(int ); /* ②. 定义一个函数指针类型 FunType,与①函数类型一致 */
void CallMyFun(FunType fp,int x);
```

```
int main(int argc, char* argv[])
{
    CallMyFun(MyFun1,10);   /* ⑤. 通过 CallMyFun 函数分别调用三个不同的函数 */
    CallMyFun(MyFun2,20);
    CallMyFun(MyFun3,30);
}
void CallMyFun(FunType fp,int x) /* ③. 参数 fp 的类型是 FunType。*/
{
    fp(x);/* ④. 通过 fp 的指针执行传递进来的函数，注意 fp 所指的函数是有一个参数的。 */
}
void MyFun1(int x) /* ①. 这是个有一个参数的函数，以下两个函数也相同。 */
{
     printf("函数 MyFun1 中输出：%d\n",x);
}
void MyFun2(int x)
{
     printf("函数 MyFun2 中输出：%d\n",x);
}
void MyFun3(int x)
{
     printf("函数 MyFun3 中输出：%d\n",x);
}
```

输出结果：略

分析：看我写的注释。你可按我注释的①②③④⑤顺序自行分析。

/* -- 完 -- */