《现代电子技术》增刊

新编C语言实用技术大全

王小华 编写

陕西电子杂志社

新编C语言实用技术大全

王小华 编写

陕西电子杂志社

银行汇至：西安市工行小寨分理处
户　　名：陕西电子杂志社　　帐　　号：216－144875－68
电　　话：（029）5236743，5244464 电　　挂：8860
　　5236744（带传真），5236745 邮政编码：710061
地　　址：西安市小寨纬二街西段 8 号

前　言

C 语言作为一种优秀的软件开发语言，近年来随着微机的普及，已越来越被广大的软件开发人员所接受。目前，C 语言在操作系统、软件工具、图象处理、汉字技术、数据处理、人工智能、数据库管理及网络技术等许多方面都得到了广泛的应用。

随着 C 语言的普及，更由于 C 语言的许多突出的优点，使得它在众多的微机语言中独树一帜，吸引了越来越多的人们应用 C 语言开发软件，形成了一个 C 语言热潮。

鉴于目前系统介绍 C 语言综合技术方面的书还很不够，很难满足广大 C 语言程序设计人员的需要。作者通过阅读国内外 C 语言方面的大量书籍，结合自己多年的工作实践，编写了这本综合性较强的 C 语言技术大全。该书的内容几乎覆盖了 C 语言在目前应用的所有领域，叙述简捷易懂，程序实例丰富，即是一本帮助初学者向高级程序设计过渡的教科书，又是一本难得的技术参考手册。书中的材料大多取自国内外专家的著作，也有作者近年来辛勤工作的结晶。为的是向广大读者奉献最新与最好的资料。

本书编写过程中，得到了陕西电子编辑部总编张忠智高级工程师等的支持和帮助，得到了西北工业大学部分老师和研究生的协助，在此一并表示谢意。

由于水平有限，恳请读者提批评意见，以便作者及时改进。

编　者

1993 年 4 月

目　录

第一章　指针技术

正确理解和使用指针对于成功地进行 C 语言程序设计是至关重要的，其理由有三个：①指针的使用为函数修改其调用参数提供了方便；②C 语言中动态分配例程需要指针的支持；③使用指针可以提高某些程序的效率。

指针作为 C 语言之最显著特征的同时，又是 C 语言最危险的特征。例如，未初始化的指针或称为无向指针（wild pointer）常常导致系统的崩溃。也许更糟的是，指针极易用错，而这类错误可以产生难以发现的程序故障。

由于指针非常有用又容易被滥用，所以本章详细讨论有关指针的问题。

1.1　指针变量

1.1.1　指针即地址

指针包含的是内存地址。在绝大多数情况下，该地址是内存中另一个变量的存储位置。若一个变量包含的是另一个变量的地址，那么就叫第一个变量指向第二个变量。图 1-1 说明的就是这种情况。

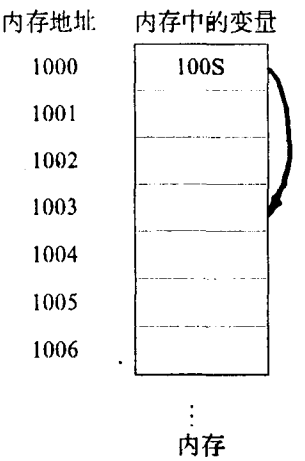

图 1-1　一变量指向另一变量

1.1.2　指针变量

欲使一个变量成为指针，必须对其加以说明。定义指针变量的一般形式是：

```
type * name:
```

这里，类型是 C 语言的任何一种有效类型（也叫基类型），名字是指针变量的名称。

指针的基类型定义了该指针所指向变量的类型。在技术上，任何类型的指针均可指向内存中的任何位置。所有的指针运算都是与它的基类型相关联的，因而正确定义指针非常重要。

1.1.3　指针运算符

有两个特殊的指针运算符：&和 * 。&是一元运算符，功能是返回其操作数的内存地址。（一元运算符仅需一个操作数。）例如：

```
m = &count:
```

上式将变量 count 的内存地址存入 m。这个地址是变量在机器内部的存储单元，与 count 的值毫无关系。记住，指针运算符&返回的是紧跟其后的变量的地址。因此，前面的赋值语句可以解释为“m 接收了 count 的地址”。

为了更好地理解内存地址的赋值，假定变量 count 在 2000 号内存单元存储其值，且 count 的值为 100；那么在前面赋值语句执行后，m 的值为 2000。另一个操作符 * 与&相反，它作为一元运算符，它返回的是存放在紧跟其后的地址中变量的值。例如，若 m 中存放的是变量 count 的内存地址，那么：

```
q = * m;
```

将把 count 的值存入 q，接着上面的例子，q 的值成为 100。因为 100 是存放在 2000 号内存单元中的数，而 2000 即是 m 中的内存地址。记忆 * 号的运算操作时，可以把它理解为"间接寻址"，这样就可以把上面的赋值语句解释为"q 得到存在地址 m 处的值"。

遗憾的是，字符 * 既是乘号又是间接寻址运算符，而字符&既是逻辑与算符又是求地址算符。* 当作指针算符时，尽管看起来与算术运算符是一样的，但它们之间并无关系。&和 * 这两个指针运算符的优先级比所有的算术运算符都要高。但有一个例外，那就是一元减法，其优先级与&和 * 是相等的。

必须保证指针变量所指向的数据的类型是正确的。例如，当把一个指针定义为 int 类型后，编译程序便"假定"它存放的任何地址都指向一个整型变量。因为 C 语言允许将任一地址赋给指针变量，故下面的程序段在编译时不会有任何错误信息（或任何警告信息，这取决于使用的编译程序），但它并不产生期望的结果：

```
main( )                    |    p = &x;
{  float x,y;              |    y =  * p;
   int * p;                |  }
```

x 的值不会赋给 y，因为 p 被定义为整型指针，故只有两个字节的信息传递给 y，而不是通常构成浮点数的若干个字节。

1.2 指针表达式

一般说来，含有指针的表达式应遵循 C 语言的表达式规则。在这一节里，我们将讨论指针表达式的几个特殊方面。

1.2.1 指针赋值

与变量相同，指针也可以出现在赋值语句的右端，将它的值赋给另一指针。例如：

```
main( )
{
    int x;
    int * p1,* p2;
    p1 = * x;
    p2=p1;
    printf("%u",p2);   /  * print the decimal value of the address of x—not x's value! * /
}
```

printf( )得到打印无符号整数的信息后，将 x 的十进制值显示出来（若编译程序符合 ANSI 标准，便可用%p 直接打印出地址）。这个程序段并不意味着无符号整数就是指针，也并非意味着通常都是用无符号整数来存放指针值。

1.2.2 指针运算

对指针来说仅有算术运算符：+、－，++和－－。为了理解指针运算的过程，假定 p1 是一个整型指针、当前值为 2000，同时假定整数是两字节的。那么

```
p1++;
```

运算后，p1 的值变为 2002，而不是 2001！p1 每递增一次，就将指向后一个整数，递减也一样。例如，假定 p1 值为 2000，那么

```
p1—:
```

将使 p1 的值变为 1998。

指针每递增一次，就指向后一个基类型元素的内存单元；指针每递减一次，就指向前一个元素的内存单元。在字符指针的情况下，这看起来就象是"正常"的算术运算。但其它各种指针都必须按照所指数据的类型决定递增或递减的步长。例如，假设有一单字节字符类型和另一个双字节整数类型：当字符指针递增时，增量为 1，而整数指针递增时，增量为 2。其原因是：指针增减时，根据基类型的长度确定增减量，以保证指针总是指向下一个或前一个元素。所有指针运算都与指针的基类型相关联，使得指针永远指向一个恰当的元素。图 1-2 对这个概念做了说明。

```
p1 = p1 = 9:
char * ch = 3000:
int * i = 3000:
```

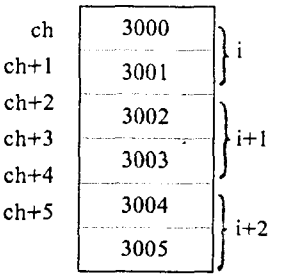

图 1-2 指针运算与基类型相关

指针运算不仅限于递增和递减，我们还可对指针进行加减运算，下式使得 p1 指到向后第 9 个具有 p1 类型的元素。

除了对指针加上或减去一个整数外，其它任何算术运算都是非法的。具体地说，不允许对指针做乘除法运算；不允许两个指针相加或相减；不允许对指针使用位移或位屏蔽操作符；不允许对指针加上或减去一个 float 类型或 double 类型的数据。

1.2.3 指针的比较

在关系表达式中可对两个指针进行比较。例如，给定两个指针 p 和 q，下列语句是完全有效的：

```
if(p<q)printf("p points to lower memory than q\n");
```

通常两个或多个指针同时指向一个目标时才使用指针比较。例如，设想建立一个存放整数的堆栈。堆栈就是一个存取方式为先进后出的线性表，它常被比作桌子上的一叠盘子，最先放上去的最后使用。堆栈在编译程序、解释程序和电子报表等与系统有关的软件中经常使用。要建立一个堆栈，需要有两个例程：push( )和 pop( )。函数 push( )将数据压入栈内，而 pop( )将数据弹出栈外。用于堆栈的内存空间是从堆中分配而来的。变量 tos 存放栈顶的地址，防止堆栈下溢。一旦堆栈被初始化，push( )就作为一个整数堆栈的函数而使用。这两个函数在这里被一个简单的函数 main( )所调用：

```
char * malloc( ); / * declare malloc( ) to be returning a character pointer * /
```

```
int  * pl, * tos;
main( )
{
   int value;
   pl=(int * ) malloc(50  * sizeof(int));
   if(!pl){
   printf("allocation failure\n")
   return;
}
tos=pl;   /  * let tos hold top of stack  * /
   do {
     scanf("%d,&value);
     if(value!=0) push(value);
     else printf("this is it %d\n",pop( ));
     } while(value!=-1);
}
push(i)
int i;
{
pl++;
if(pl= =(tos+50)) {
   printf("stack overflow");
   exit( );
}
 * pl=i;
}
pop( )
{
   if((pl)= =tos) {
     printf("stack underflow");
     exit( );
 }
   pl--;
   return  * (pl+1);
}
```

在 push( )和 pop( )中都对 pl 作了关系判断，以检测是否出现溢出错误。在 push( )中，pl 与栈顶地址（tos 加上栈的长度 50）相比较，而在 pop( )中，pl 与 tos 相比较以确保堆栈不发生下溢。

在 pop( )的 return 语句中，必须使用圆括号，否则该句就变成:

```
return  pl+1;
```

这句中返回的是 pl 指向的值加 1，而不是 pl+1 指向的值。因此，在对指针使用括号时必须十分小心，以保证计算顺序的正确性。

1.3 C 语言的内存动态分配函数

任何 C 语言程序在编译之后所占的机器内存将分为四个区域：程序码、全局数据、栈和堆。

堆是一个自由存储区，由 C 的动态分配函数 malloc( )和 free( )来管理。函数 malloc( )的功能是分配存储空间并返回一个指向分配空间起始单元的字符指针（若是 ANSI 标准的编译程序，则返回空指针），而函数 free( )是将先前分配的存储单元送回堆中以便再用。调用这两个函数的一般形式如下：

```
char * p;
p=malloc(num_bytes);
free(p);
```

这里，num_bytes 是申请的字节数，若没有足够的自由内存空间，malloc( )将返回一个空字符。在调用 free( )时必须带有一个先前已分配的有效指针。这一点很重要，否则，这种调用有可能破坏堆的组织并导致程序的崩溃。

下面的程序段分配 1000 字节的存储单元：

```
char * p;
p=malloc(1000);      / * get 1000 bytes * /
```

赋值后，p 指向这 1000 字节的存储区的起始单元。

若需分配非字符类型的存储空间，则必须在赋值语句中使用显式的特性说明。下面的语句说明如何分配 50 个整型数的内存空间：

```
int * p;
p=(int * ) malloc (50 * sizeof(int));
```

由于堆并不是无限大的，因而无论何时分配内存空间，在使用指针之前都必须检查 malloc( )的返回值是否为空(null)。使用空指针几乎肯定会导致程序崩溃。下面的程序设计给出一个分配内存空间并检测有效指针的合适方法。

```
if(!(p=malloc(10) {
   printf("Out of memory \n");
  exit(1);
}
```

当然，可以用其它错误处理程序代替 exit( )，但目的只有一个，那就是绝不能使用空指针 p。

本书后面将讨论用指针和动态分配的方法建立链接表，稀疏矩阵之类的问题。

1.4 指针和数组

指针和数组是紧密相联的，考虑如下程序段：

```
char str[80], * p1;
p1=str;
```

这里，p1 被置为数组 str 的第 1 个元素的地址，要访问 str 的第 5 个元素，可写成：

```
str[4]  或  *(p1+4)
```

注意，数组的下标从零开始，故要用下标 4 来检索 str 的第 5 个元素。指针 p1 加上 4 同样也可存取第 5 个元素。这是因为 p1 当前指向 str 的第 1 个元素。不带下标的数组名返回的是数组的起始地址，即第 1 个元素的地址。

实际上，C 语言提供了两种存取数组元素的方法：指针运算和数组下标检索。选择哪一种方法很重要，因为前者的运算可能比后者快。由于程序设计中速度是经常考虑的问题，所以在 C 语言程序中用指针存取数组元素是很普遍的。

看下例是如何用指针代替数组下标的。考虑两种方法编写的函数 puts( )：一个用下标检索，另一个用指针。puts( )的功能是将一个字符串写入标准输出设备，下面是该函数的两种写法：

```
push(s) / * with arrays * /
char *s;
{
   register int t;
   for(t=0;s[t];++t)putchar(s[t]);
}
puts(s) / * with pointers * /
char *s;
{
   while(*s) putchar(*s++);
}
```

大多数专业 C 语言程序员都认为第二种方法易读易懂。事实上，C 语言中诸如此类的例程通常都是用指针方法编写的。

然而，初学 C 语言的程序员有时可能错误地认为使用指针的效率总是高得多，因而从不使用数组下标检索。事实这并不全面：如果按照严格的递增或递减顺序访问数组，那么指针用起来就又快又方便；但如果数组是随机访问的，那么用下标就更好一些。使用下标常常和使用复杂指针表达式速度一样快，而且编程和理解都很容易。另外，当使用数组时，编译程序还可以帮助我们做些处理。

1.4.1 字符数组的指针

C 语言中许多字符串操作通常是由数组的指针及指针运算来实现的。因为对字符串来说，一般都是严格的顺序存取方式，故指针是明显的选择。

例如，下面是 C 语言程序库中函数 strcmp() 的一个写法：

```
strcmp(s1,s2) / * with pointers * /
char *s1, *s2;
{  while(*s1)
if(*s1=s2)
   return *s1-*s2;
else {
```

```
    s1++;
    s2++;
  }
  return '\0';  /* equal */
  }
```

切记，C 语言中所有字符串均以空字符 null 结尾，其值为 false。因而只有到达串尾时，下列语句才为 false:

```
while( *s1)
```

这里，当 s1 等于 s2 时，strcmp() 返回值为 0；如果 s1 小于 s2，则返回值小于 0；否则其返回值大于 0。

当考虑循环控制时，多数类似 strcmp() 这样的字符串函数均使用指针。这种情况使用指针速度更快，效率更高，而且比数组更容易理解。

在 strcmp() 中，s1 和 s2 都是局部变量，它们可被修改但对调用参数无副作用。但是我们应用时必须小心。为了解其原因，请仔细考察下列程序。

```
main( ) /  * this program has a bug * /
{
  char s[80];
  char *p1;
  p1=s;
  do {
    gets(s);  /  * read a string * /
    /  * print the decimal equivalent of each character * /
    while(*p1) printf("%d",*p1++);
  } while(! strcmp(s,"done"));
}
```

读者能发现其中的错误吗?

问题出在程序只做了一次将 s 的地址赋给 p1 的操作。在第一次循环时，p1 确实指向 s 的第一个字符，但在第二次循环时，p1 将从当前地址出发继续移动。因为它没有被重新置为 s 的起始地址，下一个字符也许是另一个字符串的一部分，或是另一个变量，甚至是程序代码。该程序的正确写法如下:

```
main( ) /  * this program is correct * /
{
char *p1;
do{
p1=s;
gets(s):  / read a string /
/  * print the decimal equivalent of each character * /
while(*p1) printf("%d",*p1++);
}while(! strcmp(s,"done"));
}
```

这样，每循环一次，P1 都被置为字符串的起始地址。

1.4.2 指针数组

指针可以象任何数据类型那样被说明为数组。长度为 10 的整型指针数组可定义如下:

```
int  * x[10];
```

要将整型变量 var 的地址赋给数组的第 3 个元素，可以写为:

```
x[2] = &var
```

欲求 var 的值，可写为:

```
* x[2]
```

若要将指针数组传递给一个函数，可采用传递其它类型数组的同样方法，调用函数时带上数组名即可，无需带任何下标。例如，接收数组 X 的函数如下:

```
display_array(q)
int  * q[ ];
{   int t;
    for(t = 0.t < 10;t++)
    printf("%d",(q[t]);
}
```

注意: q 不是整型指针，它是整型指针数组的指针。因而在程序中要将参数 q 定义为整型指针的数组，而不能定义成整型指针，因为它本身就不是整型指针。

指针数组常用来存放指向错误信息的指针。正如函数 serror() 表明的那样，可以定义一个函数，它根据错误码编号输出信息:

```
serror(num)
int num;
{  static char  * drr[ ] = {
   "cannot open file\n",  "read error\n",  "write error\n",  "media failure\n"
   };
printf("%s",err[num]);
}
```

正如我们看到的那样，调用 serror() 中 printf() 时用的是字符指针，它指向某一个错误信息，这些信息是通过传递给函数的错误号来检索的。例如，如果 num 传递的值为 2，那么将显示信息"Write error"。

有趣的是，命令行变量 argv 也是字符指针数组。

1.5 多级指针

指针数组与指针的指针是一回事。指针数组的概念容易理解一些，因为下标使其含义非常明确，而指针的指针却很容易造成混乱。

指针的指针是多级间址（multiple indirection）的一种形式，或叫指针链(chain of pointers)，见图 1-3。

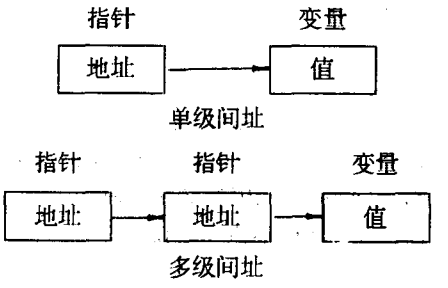

图 1-3 单级间址和多级间址

如图所示，在单级间址中，指针的

值就是存放期望值的变量的地址。在后一种情况下，第一个指针存放第二个指针的地址，而第二个指针指向存放期望值的变量。

多级间址能延伸到所需要的任何一级，但很少有需要超过指针的指针这种情况的。间址的级数过多会使跟踪困难，且易发生概念性错误。(不要混淆多级间址与链接表，后者用在数据库一类的软件中。)

将变量定义成指针的指针时，需要在变量名之前附加一个额外的符号 * 。例如以下声明通知编译程序：newbalance 是一个指向 float 型指针的指针：

```
float * * newbalance;
```

注意：newbalance 不是指向浮点数的指针，它是指向另一指向 float 型指针的指针。

欲访问一个指针的指针所间接指向的目标值，必须连续两次使用运算符 * 。例如：

```
main( )
{
  int x, *p, * *q;
  s=10;
  p=&x;
  q=&p;
  printf("%d",* *q); /  * print the value of x  * /
}
```

这里，p 定义为指向一整数的指针，而 q 是指向另一指向整数之指针的指针，调用 printf() 将在屏幕上显示出"10"。

1.6 指针的初始化

一个指针在定义之后和赋值之前，其值是未知的。如果在对其赋值之前即引用之，则不仅可能破坏程序，甚至可能破坏计算机的操作系统。这是非常严重的错误!

按照惯例，未用指针应当赋以值 null，以表明它未指向任何地方。不过正因为值为 null 的指针并不保险，如果将它放在赋值句的左端，仍然有使程序或操作系统崩溃的危险。

一个空指针被认为是未用的。在应用指针的例程中，利用空指针可以简化程序代码、提高效率。例如，可用空指针作为指针数组的结尾标志。这样，访问数组的例程一旦遇到空值 null，便"知道"已达到数组末尾。下面的函数 search() 说明了这种方法。

```
/  * look up a name  * /
search(p,name)
char  * p[ ],* name;
{
  register int t;
  for(t=0;p[t];++t)
  if(! strcmp(p[t],name))return  t;
  return-1;/  * not found  * /
}
```

仅当产生一个匹配或遇到一个空指针时，search() 中的 for 循环才终止。因为数组的