

第 8 章 指针

指针是 C 语言最重要和最独特的一种变量类型。指针能够模拟引用调用，使函数返回多个值；能够建立和操作动态数据结构，如链表、队列、栈与树等；能像汇编语言一样处理内存地址，编出精炼而高效的程序。

8.1 指针的概念

在讲指针的概念之前，先来看一个例子。如图 8-1 所示，在这里居民的住家有三层意思：

- (1) 房子名称——赵刚的府邸（赵刚的家）
- (2) 房子所在的地址——大连市明星路呼啸山庄 9 栋 1 号
- (3) 房子内所住的人——房主赵刚，其妻王莉，女儿蒙蒙



图 8-1 居民住家

若邮递员送一封挂号信给赵刚，则邮递员送信的过程为：按地址找到房子→确认是赵刚的家→赵刚签字收信。

计算机的内存储器（内存）是存储变量的容器，存储管理将存储器划分成一个个具有一定容量（1 个字节）的存储单元，每个存储单元都有自己的编号，把这个编号称做该存储单元的地址（一般用十六进制数表示）。而单元的内部存储的是变量的值，因此，程序中的变量也具有像住家一样的 3 个层次的要素：

- (1) 变量名
- (2) 存储变量的存储单元地址
- (3) 变量的值（变量的内容）

图 8-2 给出了变量的三个层次的要素。

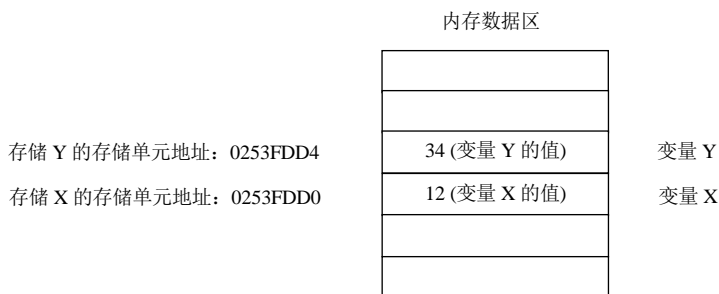


图 8-2 变量的 3 个层次的要素

若程序中的变量参与运算，变量取值的过程为：首先找到存储该变量的存储单元地址，再由该单元内取出变量的值。

通常把存储某个变量的存储单元地址，简称为该变量的地址。换句话说，该地址指明了变量值存储的地方，是指向该变量的，因此把变量的地址又叫做变量的指针。

由第 2 章我们知道，不同类型的变量，其取值的范围不同，在内存中存储时所占的字节数（内存单元个数）也不同。如整型变量占 4 个字节，实型占 4 个字节，字符型占 1 个字节等。通常变量的地址是由编译系统所决定的，程序编译时根据变量定义时的类型分配一定长度的内存单元给它，一般计算机均采用“字节寻址法”，即把内存中的每个字节按顺序编号，而变量的地址即为最开头第一个字节的地址。看例 8.1。

【例 8.1】定义不同类型的变量，显示变量的地址与所占字节数，观察它们之间的关联。

```
#include <stdio.h>
void main()
{ int x;
  float y;
  char z;
  x=2;
  y=2.5;
  z='a';
  printf("x=%3d, sizeof(x)=%d, address=%x\n", x, sizeof(x), &x);
  printf("y=%3.1f, sizeof(y)=%d, address=%x\n", y, sizeof(y), &y);
  printf("z=%3c, sizeof(z)=%d, address=%x\n", z, sizeof(z), &z);
}
```

程序运行结果为：

```
x= 2, sizeof(x)=4, address=12FF7c
y=2.5, sizeof(y)=4, address=12FF78
z= a, sizeof(z)=1, address=12FF74
```

分析：变量的存储顺序与变量的定义顺序相反，为由变量 z 向变量 x 的逆向存储。变量 x 占用 12FF7c、12FF7d、12FF7e 和 12FF7f 四个字节，因此变量 x 的地址为 12FF7c，其他变量地址分配如图 8-3 所示。

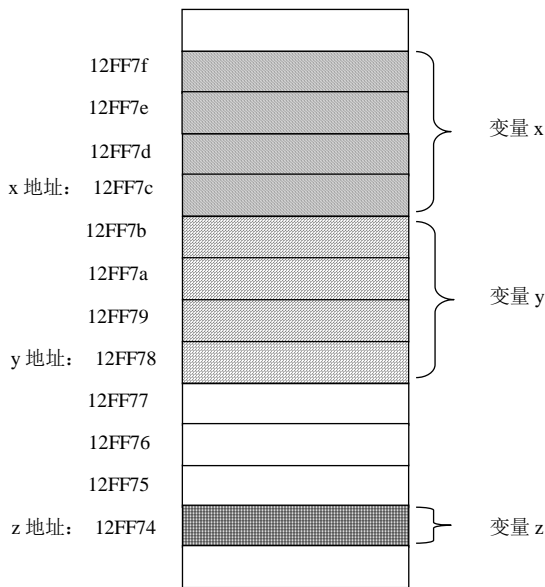


图 8-3 变量地址分配图

注意：函数 `sizeof`（变量名）给出变量所占的字节数，`&`为地址运算符，用于取变量的地址。

8.2 指针变量和指针运算符

本节我们介绍用指针变量来表示变量的地址与对指针进行运算的符号。

8.2.1 指针变量

变量的存取方式分为两种：直接访问方式与间接访问方式，如图 8-4 所示。

按照变量地址存取变量值的方式称为“直接访问方式”。

假设有一个变量 `x`，将变量 `x` 的地址存放在另一个变量 `Px` 中，通过变量 `Px` 找到变量 `x` 的地址，从而进行存取变量 `x` 值的方式称为“间接访问方式”。

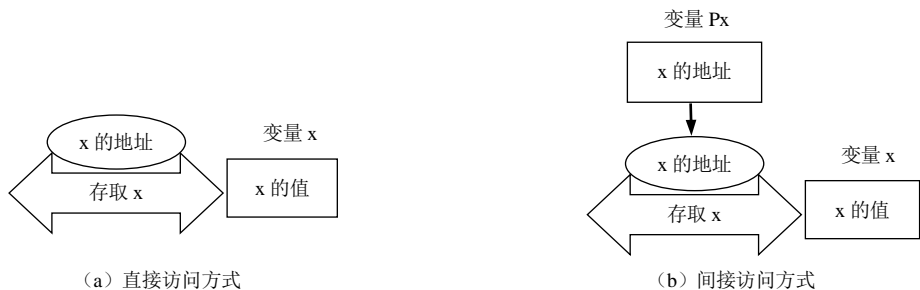


图 8-4 变量的存取方式

把专门用于存储变量指针（地址）的变量称为指针变量。如变量 `Px`，即为指针变量，由

于 Px 中存放的是变量 x 的地址, 因此把变量 Px 称做变量 x 的指针变量。换个角度, 通常叫做指针变量 Px 指向变量 x。

C 语言规定所有的变量必须先定义后使用。指针变量不同于其他变量, 是专门用做存放地址的变量, 应定义为指针类型。

格式:

类型说明符 *指针变量名;

说明:

(1) 类型说明符: 是定义的指针变量所指向变量的类型。

(2) *: 专门用于定义指针变量的, 表示*后边的变量名是一个指针变量, 而不是其他的普通变量。

例如,

```
int *ptr1;           /*定义一个指向整型变量的指针变量 ptr1*/
char *ptr2;          /*定义一个指向字符型变量的指针变量 ptr2*/
float *ptr3;          /*定义一个指向实型变量的指针变量 ptr3*/
```

注意: 指针变量只能指向同一类型的变量。若指针变量的类型与所指向的变量的类型不同, 编译时就会出错。

既然指针变量也是变量, 那么指针变量在内存中也要占用一定的字节数, 在多数计算机内指针变量占用 4 个字节。

【例 8.2】 定义不同类型的变量及其指针变量, 显示变量所占字节数与指针变量所占字节数。

```
#include <stdio.h>
void main()
{ int x;*px;
  float y;*py;
  char z;*pz;
  x=2;px=&x;
  y=2.5;py=&y;
  z='a';pz=&z;
  printf("x=%3d,sizeof(x)=%d, sizeof(px)=%d \n",x,sizeof(x), sizeof(px));
  printf("y=%3.1f,sizeof(y)=%d, sizeof(py)=%d \n",y,sizeof(y), sizeof(py));
  printf("z=%3c, sizeof(z)=%d, sizeof(pz)=%d \n",z,sizeof(z), sizeof(pz));
}
```

程序运行结果为:

```
x= 2,sizeof(x)=2, sizeof(px)=4
y=2.5, sizeof(y)=4, sizeof(py)=4
z= a, sizeof(z)=1, sizeof(pz)=4
```

8.2.2 指针运算符

指针运算符共有两个, 一是取地址运算符&, 一是取内容运算符*, 且都为单目运算符。

1. 取地址运算符&

利用取地址运算符&可以取得变量在内存中的地址，其使用格式为：

```
&变量名
例如，int x,*px;
      x=5;
      px=&x;          /* 取 x 的地址赋给指针变量 px */
```

注意：指针变量未赋值时，可以是任意值，可能是某个变量的地址，也可能是系统某个数据的地址，因此是不能使用的，否则将造成意外错误。

2. 取内容运算符*

利用取内容运算符*可以取得指针变量所指向的变量的内容（即变量的值），取内容的过程是，先取得指针变量内存放的变量地址，再根据变量地址取出存储单元里的内容，其使用格式为：

```
*指针变量名
说明：
```

（1）这个星号和定义指针变量时的星号意义不同，定义指针变量时的星号是告诉编译系统该变量应定义为指针类型，而内容运算符*是取指针变量所指向的变量的内容。

（2）*号后的指针变量必须是指向某一个具体的变量，否则取内容的话会出错。

【例 8.3】指针运算符的应用。

```
#include <stdio.h>
void main()
{ int x,*px;
  float y,*py;
  x=2;px=&x;
  y=2.5;py=&y;
  printf("x=%3d,*px=%3d\n",x,*px);      /* *px 为取出 px 所指向的变量 x 的值 */
  printf("y=%3.1f,*py=%3.1f\n",y,*py);
}
```

运行结果为：

```
x= 2,*px= 2
y=2.5,*py=2.5
```

分析：对指针变量 px 进行取内容运算的过程为：

- （1）按照指针变量 px 里存的 x 的地址，找到存储变量 x 的内存单元。
- （2）取出单元里的内容，如图 8-5 所示。所取得的内容就是 x 的值，*px 代表的值与 x 的值是同一个内存单元里的值（图中实线箭头为指针指向变量，虚线箭头为数据操作，以下同）。

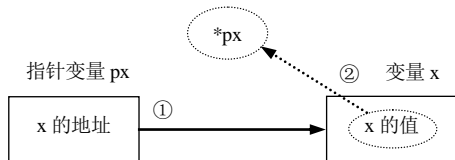


图 8-5 对指针变量 px 进行取内容运算的过程

注意：由于指针变量 `px` 指向变量 `x`，`x` 与 `*px` 等价，则 `*px` 可出现在 `x` 能出现的任何地方，即能参加 `x` 所具有的所有运算。

8.2.3 指针变量的运算

指针变量的运算主要在赋值和加减运算方面。

1. 指针变量的赋值运算

指针变量的赋值运算有以下 3 种形式。

(1) 指针变量的初始化赋值。如：

```
int x, *px=&x;    /* 指定义指针变量的同时给它赋值 */
```

(2) 把一个变量的地址赋给指向相同数据类型的指针变量。如：

```
int x, *px;
px=&x;           /* 指定义指针变量和给指针变量的赋值分为两步进行 */
```

(3) 把一个指针变量的值赋给指向相同数据类型的另一个指针变量。如：

```
int x, *ptr1, *ptr2;
ptr1=&x;
ptr2=ptr1;       /* ptr2 和 ptr1 具有相同的数值，即变量 x 的地址 */
```

【例 8.4】 指针变量的赋值运算应用一。

```
#include <stdio.h>
void main()
{ int x, *ptr1=&x, *ptr2;
  x=2;
  ptr2=ptr1;
  printf("x=%d, *ptr1=%d, &x=%x, ptr1=%x \n", x, *ptr1, &x, ptr1);
  printf("x=%d, *ptr2=%d, &x=%x, ptr2=%x \n", x, *ptr2, &x, ptr2);
}
```

程序的运行结果为：

```
x=2, *ptr1=2, &x=12ff7c, ptr1=12ff7c
x=2, *ptr2=2, &x=12ff7c, ptr2=12ff7c
```

分析：由程序的运行结果可以看出：指针变量 `ptr1` 和 `ptr2` 与 `&x` 具有相同的数值，也就是说，指针变量 `ptr1` 和 `ptr2` 同时指向变量 `x`。

【例 8.5】 指针变量的赋值运算应用二。

```
#include <stdio.h>
void main()
{ int x, *ptr1=&x, *ptr2;
  x=2;
  ptr2=ptr1;
  *ptr2=5;        /* 把 5 赋给 ptr2 所指向的变量 x，相当于语句 x=5 */
  printf("x=%d, *ptr1=%d, &x=%x, ptr1=%x \n", x, *ptr1, &x, ptr1);
  printf("x=%d, *ptr2=%d, &x=%x, ptr2=%x \n", x, *ptr2, &x, ptr2);
}
```

程序的运行结果为：

```
x=5, *ptr1=5, &x=12ff7c, ptr1=12ff7c
```

`x=5, *ptr2=5, &x=12ff7c, ptr2=12ff7c`

分析：由图 8-6 执行语句 `*ptr2=5` 前后对比图可见，由于 `ptr1` 和 `ptr2` 两个指针变量内存放的都是变量 `x` 的地址，对它们取内容运算 `*ptr1`、`*ptr2` 都应是 `x` 的值，则执行语句 `*ptr2=5` 就是把 5 赋给了变量 `x`。对于变量 `x`、`*ptr1`、`*ptr2` 三者，无论改变谁的具体数值，都会使其他两者的值随之而改变，因为它们代表的都是同一个内存单元里面的数值。

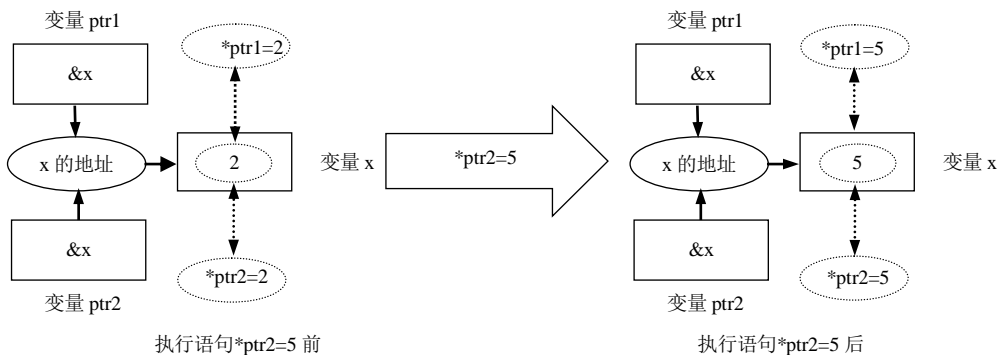


图 8-6 执行语句 `*ptr2=5` 前后对比图

2. 指针变量的加减运算

在算术表达式、赋值表达式和比较表达式中，指针变量是合法的操作数，但并非所有的运算符在与指针变量一起使用时都是合法的。可以对指针进行有限的算术运算，包括自增（++）、自减（--）、加上一个整数（+或+=）、减去一个整数（-或-=）以及减去另一个指针。

指针变量的加法与减法运算指的是对于指针变量内所存的地址进行的运算，执行加法与减法运算时，是针对各个指针变量所指向变量的类型在内存所占的字节数来处理的。

（1）指针变量加减一个整数

指针变量加或减一个整数时，指针变量并非简单地加上或减去该整型数值，而是加上该整数与指针变量所指向变量在内存中占的字节数的乘积。

如图 8-7 所示，假设定义一个整型变量 `x`，并使指针 `px` 指向变量 `x`，当做指针加法运算 `px+1` 时，是将变量 `x` 的地址加上整型类型的长度 4，若变量 `x` 的地址为 1000，则执行加法运算 `px+1` 后，`px` 的值变为 1004，而不是 1001 或 1003 等其他值。当执行加法运算 `px+1` 后，指针变量 `px` 内存的已经不是变量 `x` 的地址了，也就是说，`px` 已经不指向变量 `x` 了。

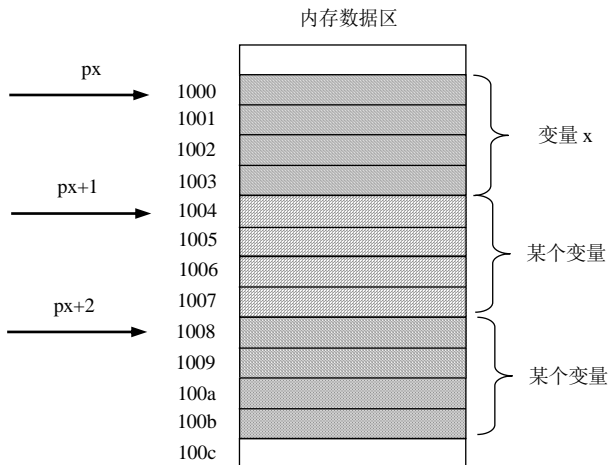


图 8-7 指针变量加法运算示意图

总结, 对于指向整型变量的指针变量, 若加 1, 则指针变量里的值 1000 要加 $1*4=4$ (一个整型变量所占的字节数); 若加 2, 则指针变量里的值 1000 要加 $2*4=8$ (两个整型变量所占的字节数), 其他类型指针变量的加减运算依此类推。

【例 8.6】 指针变量的加减运算应用, 为使大家容易理解, 显示地址用的是无符号十进制数。

```
#include <stdio.h>
void main()
{ int x,*px;
  float y,*py;
  x=2;px=&x;
  y=2.5;py=&y;
  /* 输出指针变量未加减运算前的 x,*px,px,y,*py,py 等项的值 */
  printf("x=%3d,*px=%3d,px=%u\n",x,*px,px);
  printf("y=%3.1f,*py=%3.1f,py=%u\n",y,*py,py);
  printf("*****\n");
  px++;
  py--;
  /* 输出指针变量加减运算后的 x,*px,px,y,*py,py 等项的值 */
  printf("x=%3d,*px=%3d,px=%u\n",x,*px,px);
  printf("y=%3.1f,*py=%3.1f,py=%u\n",y,*py,py);
}
```

程序运行结果为:

```
x= 2,*px= 2,px=1245052
y=2.5,*py=2.5,py=1245044
*****
x= 2,*px= 1245129,px=1245056
y=2.5,*py=0.0,py=1245040
```

分析: 由程序运行结果可以看出, 执行语句 `px++` 前后, `px` 的值相差为 4, 刚好是整型变量在内存所占的字节数; 执行语句 `py--` 前后, `py` 的值相差为 4, 刚好是实型变量在内存所占的字节数。观察两组 `*px` 和 `*py` 的值, 执行语句 `px++`、`py--` 后, `*px` 和 `*py` 的值已经不是 `x` 和 `y` 的值了, 说明 `px` 和 `py` 已经不指向 `x` 和 `y` 了。

在 C 语言中, 允许两个相同类型的指针变量做减法运算, 运算的结果为两个指针之间的距离, 即两个指针相减之差除以指针变量所指向变量在内存中占的字节数的商, 也就是两个指针相差的同类型数据个数。假设有两个整型指针变量 `ptr1` 和 `ptr2`, 两指针变量相减情形如图 8-8 所示。

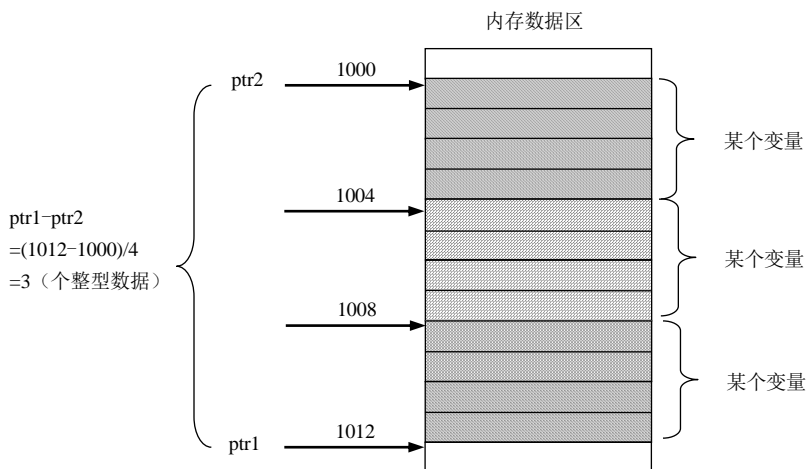


图 8-8 两指针变量相减示意图

【例 8.7】 指针变量的减法运算应用。

```
#include <stdio.h>
void main()
{ int x,y,z;
  int *ptr1,*ptr2;
  ptr1=&x;
  ptr2=&y;
  z=ptr1-ptr2;
  printf("ptr1=%u,ptr2=%u \n",ptr1,ptr2);
  printf("ptr1-ptr2=%d\n",z);
}
```

程序的运行结果为:

```
ptr1=1245052, ptr2=1245048
ptr1-ptr2=1
```

3. 指针变量的使用

指针变量可以出现在表达式中。指针运算符*和&优先级相同，若表达式中含有这两种运算符，则运算方向为从右向左。

如，若“int x,*px=&x;”，则存在以下关系：

$\&*px \rightarrow \&(*px) \rightarrow \&(x) \rightarrow \&x$	/* &*px 与 &x 等价 */
$*\&x \rightarrow *(\&x) \rightarrow *(px) \rightarrow *px \rightarrow x$	/* *&x 与 x 等价 */
$++*px \rightarrow ++(*px) \rightarrow ++x$	/* 先取*px的内容，然后内容加1 */
$(*px)++ \rightarrow x++$	/* 先取*px的内容，然后内容加1 */
$*px++ \rightarrow *(px++)$	/* 先取*px的内容，然后 px+1 */
$++px \rightarrow *(++px)$	/* 先使 px+1，然后取*px的内容 */

【例 8.8】 指针变量的综合应用。

程序 1:

```
#include <stdio.h>
```

```

void main()
{ int x=4,y;
  int *px;
  px=&x;
  y=++*px;

  /* 先取*px 的值为 4, 使 4 加 1 等于 5, x 的值也变为 5, 然后把 5 赋给 y */
  printf("x=%d,y=%d,px=%u \n",x,y,px);
}

```

程序 1 的运行结果为:

x=5,y=5,px=1245052

程序 2:

```

#include <stdio.h>
void main()
{ int x=4,y;
  int *px;
  px=&x;
  y=(*px)++;

  /* 先取*px 的值为 4, 把 4 赋给 y, 然后使 4 加 1 等于 5, x 的值也变为 5 */
  printf("x=%d,y=%d,px=%u \n",x,y,px);
}

```

程序 2 的运行结果为:

x=5,y=4,px=1245052

程序 3:

```

#include <stdio.h>
void main()
{ int x=4,y;
  int *px;
  px=&x;
  y=*px++; /* 先取*px 的值为 4, 把 4 赋给 y, 然后使 px 加 1, x 的值还是 4 */
  printf("x=%d,y=%d,px=%u \n",x,y,px );
}

```

程序 3 的运行结果为:

x=4,y=4,px=1245056

程序 4:

```

#include <stdio.h>
void main()
{ int x=4,y;
  int *px;
  px=&x;
  y=++*px; /* 先使 px 加 1, 然后取*px 的值, 再把值赋给 y, x 的值不变还是 4 */
  printf("x=%d,y=%d,px=%u \n",x,y,px );
}

```

程序 4 的运行结果为:

x=4,y=1245120,px=1245056

【例 8.9】在 3 个数值中找出最大值和最小值。

```

#include<stdio.h>
void main()
{int  a,b,c,*pmax, *pmin;
  scanf("%d %d %d", &a,&b,&c); /* 输入 3 个数值 */
  if(a>b) /* 如果第一个数大于第二个数 */
  { pmax=&a;
    pmin=&b;
  }
  else /* 如果第一个数小于第二个数 */
  { pmax=&b;

```

```

        pmin=&a;
    }
    if(c>*pmax) pmax=&c;          /* 如果第三个数大于当前最大值 */
    if(c<*pmin) pmin=&c;          /* 如果第三个数小于当前最小值 */
    printf("a=%d, b=%d, c=%d \n", a, b, c);
    printf("max=%d, min=%d\n", *pmax, *pmin);
}

```

输入:

6 3 9

运行结果为:

a=6, b=3, c=9
max=9, min=3

8.2.4 指针变量作为函数参数

指针变量作为函数的参数,是指针的典型应用之一。大家知道,函数想返回某个结果给原调函数,可以利用 `return`,但是 `return` 只能返回一个值,当程序需要传递两个以上的值时,就无法利用 `return`。指针变量作为函数的参数,可以帮我们解决函数间传递多个返回值的问题。

【例 8.10】利用 `return` 返回值使实参的值扩大 10 倍。

```

#include<stdio.h>
int s1(int p1)
{ p1=10*p1;
  return p1;
}
void main()
{int a, b;
  scanf("%d %d", &a, &b);          /* 输入 a 和 b 的值 */
  printf("a=%d,b=%d\n", a, b);     /* 输出调用函数前 a 和 b 的值 */
  a=s1(a);
  b=s1(b);
  printf("10*a=%d,10*b=%d\n", a,b); /* 输出调用函数后 a 和 b 的值 */
}

```

程序运行结果为:

4 6✓
a=4,b=6
10*a=40,10*b=60

分析: 由于 `return` 只能返回一个值,用 `return` 解决上述问题必须两次调用函数 `s1`,且在主函数中通过赋值语句“`a=s1(a)`”和“`b=s1(b)`”才改变了 `a` 和 `b` 的值。

下面用指针作参数传递解决这个问题。

【例 8.11】利用指针作参数传递使实参的值扩大 10 倍。

```

#include<stdio.h>
void s1(int *p1,int *p2)
{ *p1=10**p1;
  *p2=10**p2;
}

```

```

void main()
{int a, b;
  scanf("%d %d", &a, &b);          /* 输入 a 和 b 的值 */
  printf("a=%d,b=%d\n", a, b);    /* 输出调用函数前 a 和 b 的值 */
  s1(&a,&b);
  printf("10*a=%d,10*b=%d\n", a,b); /* 输出调用函数后 a 和 b 的值 */
}

```

程序运行结果为:

```

4 6✓
a=4,b=6
10*a=40,10*b=60

```

分析: 由例 8.11 可以看出, 利用指针作参数, 在被调函数中通过赋值语句 “ $*p1=10**p1$ ” 和 “ $*p2=10**p2$ ” 直接改变了 a 和 b 的值。参数的传递及参数的变化过程如图 8-9 所示。

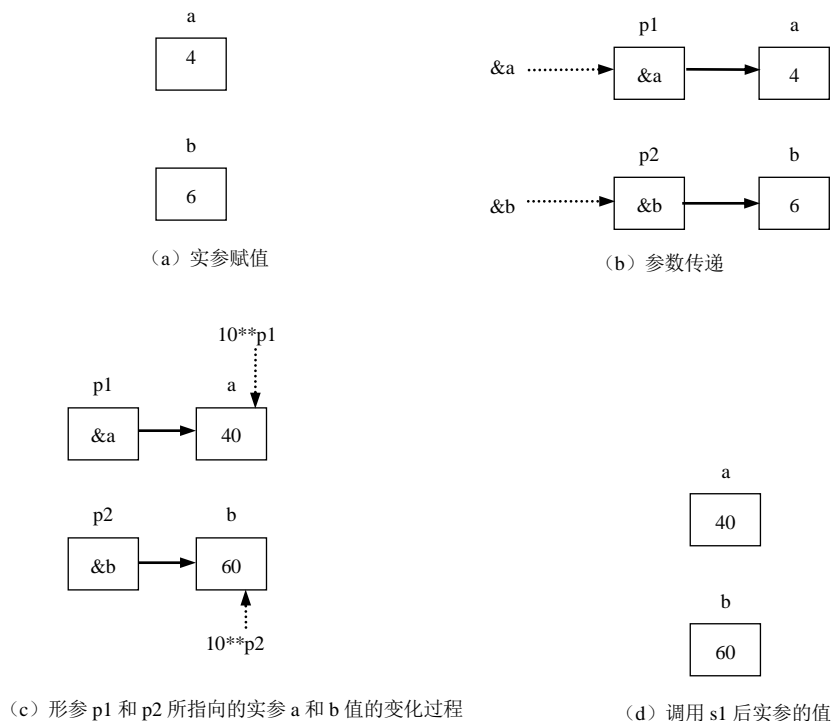


图 8-9 参数的传递及参数的变化过程图

下面用一般变量作为函数的参数及指针变量作为函数的参数两种方法进行比较为例, 来解决 “将 a 和 b 两个数进行交换” 的问题。

【例 8.12】 利用一般变量作为函数的参数, 来解决 “将 a 和 b 两个数进行交换” 的问题。

```

#include<stdio.h>
void swap(int p1, int p2)
{int t;
  t=p1; p1=p2; p2=t;          /* 3 条语句实现 p1 和 p2 的交换 */
}

```

```

}
void main()
{int a, b;
  scanf("%d %d", &a, &b);          /* 输入 a 和 b 的值 */
  printf("a=%d,b=%d\n", a, b);     /* 输出调用 swap 前 a 和 b 的值 */
  swap(a,b);
  printf("a=%d,b=%d\n", a, b);     /* 输出调用 swap 后 a 和 b 的值 */
}

```

程序的运行结果为:

```

2 7✓
a=2,b=7
a=2,b=7

```

分析: 由函数一章知道, C 语言中参数的传递是单向的, 只能由主调函数传给被调函数, 不存在反向传递。上面的例子说明了这一点, 虽然实参 *a* 和 *b* 将数值传给了 *p1* 和 *p2*, 但在函数 *swap* 中形参 *p1* 和 *p2* 的交换并不传回给实参 *a* 和 *b*, 因此 *a* 和 *b* 的值并没有变化。用一般变量作为函数的参数, 无法解决“将 *a* 和 *b* 两个数进行交换”的问题。参数的传递及参数的变化如图 8-10 所示。

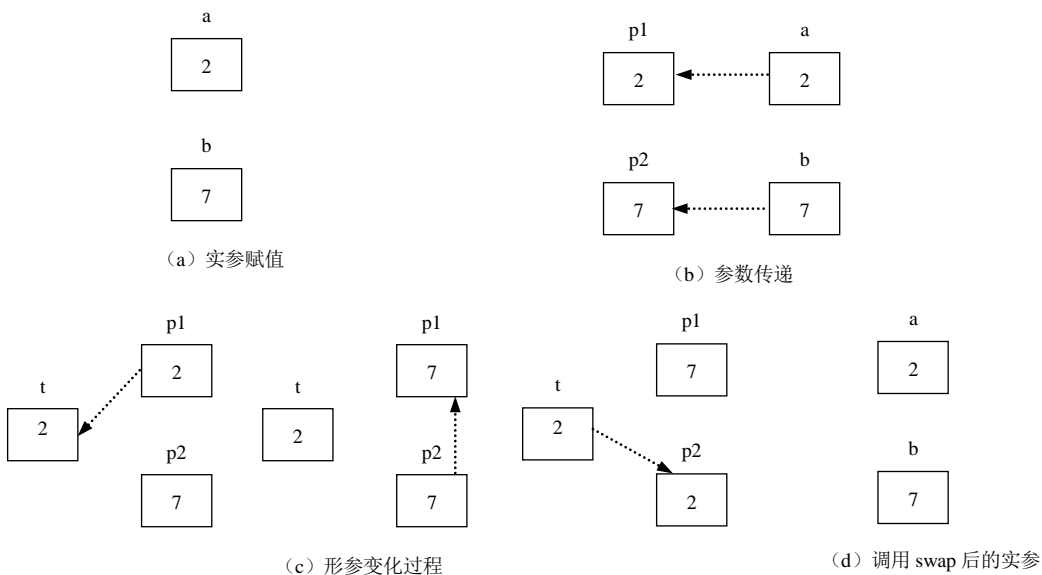


图 8-10 参数的传递及参数的变化过程图

【例 8.13】 利用指针变量作为函数的参数, 来解决“将 *a* 和 *b* 两个数进行交换”的问题。

```

#include<stdio.h>
void swap(int *p1, int *p2)
{int t;
  t=*p1;
  *p1=*p2;          /* 3 条语句实现 *p1 和 *p2 的交换, 即 a 和 b 的交换 */
  *p2=t;
}
void main()

```

```

{int a, b;
  scanf("%d %d", &a, &b);          /* 输入 a 和 b 的值 */
  printf("a=%d,b=%d\n", a, b);     /* 输出调用 swap 前 a 和 b 的值 */
  swap(&a,&b);
  printf("a=%d,b=%d\n", a, b);     /* 输出调用 swap 后 a 和 b 的值 */
}

```

程序的运行结果为:

```

2 7✓
a=2,b=7
a=7,b=2

```

分析: 由运行结果及图 8-11 可知, 虽然 C 语言中参数的传递是单向的, 但可以通过作为形参的指针变量来改变它所指向的实参变量的值。上面的例子说明了这一点, 实参 a 和 b 把自己的地址传给了 p1 和 p2, 在函数 swap 中通过 *p1 和 *p2 的交换, 实际使实参 a 和 b 的值进行了交换。因此用指针变量作为函数的参数, 很好地解决了“将 a 和 b 两个数进行交换”的问题。参数的传递及参数的变化如图 8-11 所示。

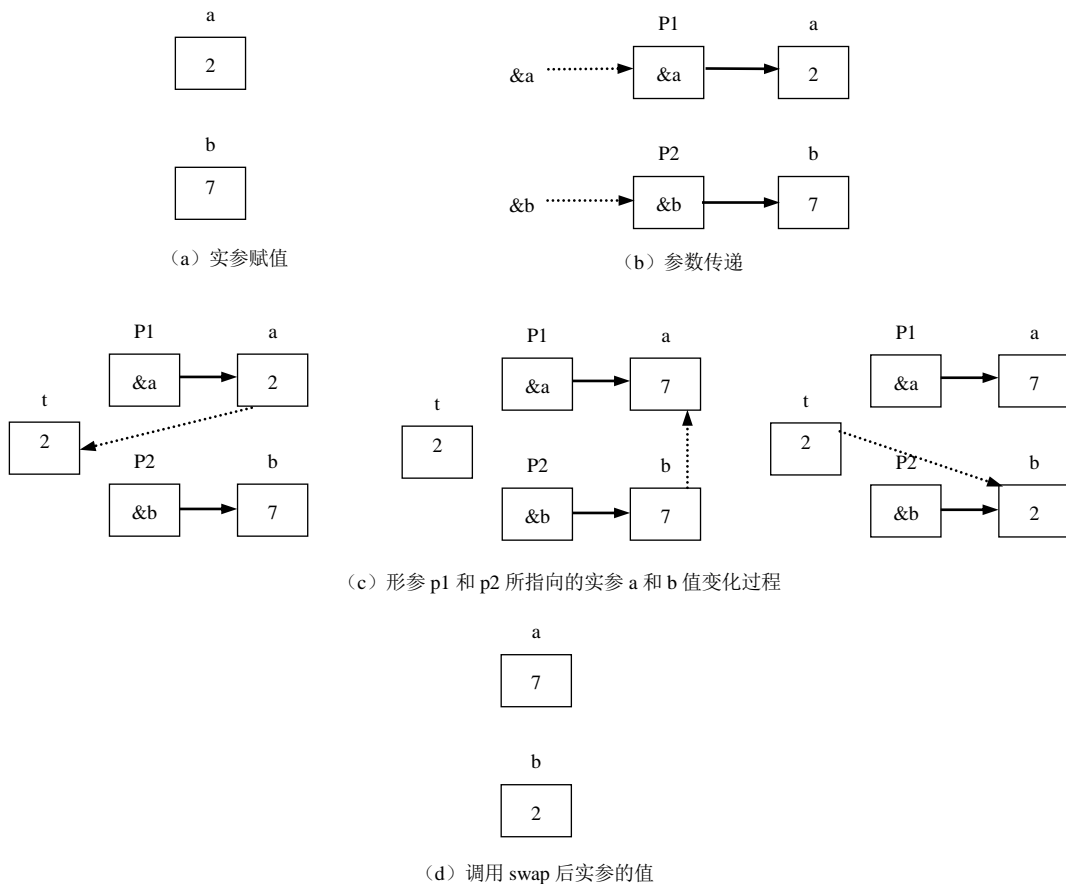


图 8-11 参数的传递及参数的变化过程图

实际上, 函数返回多个值的方法, 就是将实参的地址传递给形参, 使形参指向实参, 在被调函数中改变形参所指向的实参的值, 当然主调函数中实参的值也就发生了变化。

8.3 指针与一维数组

数组是一组数据的集合。在 C 语言中，可以将某个指针变量指向数组的某个元素，通过指针变量的移动可以指向数组的其他元素，然后就可以对数组的元素进行各种操作了。

8.3.1 引用数组元素的数组名法

众所周知变量在内存中占有一定字节数的空间，变量的地址即为最开头第一个字节的地址，把最开头一个字节的地址称为首地址，通过变量的指针（地址）可以对变量进行一系列的操作。数组在内存中占用一定长度的内存块，数组名就代表了数组的首地址，可以把数组名看作一个常量指针，它代表的就是数组中下标为 0 的元素地址。编译时数组的操作要转化为指针表示，即数组名加上数组元素的下标，就是该元素的地址。

【例 8.14】数组名作为常量指针的地址表示法应用。

```
#include<stdio.h>

void main()
{
    int a[3]={1,3,5};
    printf("a=%u, (a+1)=%u , (a+2)=%u \n", a, a+1,a+2);
    printf("&a[0]=%u, &a[1]=%u, &a[2]=%u \n", &a[0], &a[1], &a[2]);
    printf("a[0]=%d,a[1]=%d, a[2]=%d \n", a[0], a[1],a[2]);
    printf("*a=%d,*(a+1)=%d, *(a+2)=%d \n", *a, *(a+1),*(a+2));
}
```

程序运行结果为：

```
a=1245044, (a+1)=1245048, (a+2)=1245052
&a[0]=1245044, &a[1]=1245048, &a[2]=1245052
a[0]=1,a[1]=3,a[2]=5
*a=1,*(a+1)=3,*(a+2)=5
```

分析：由程序的运行结果可以看出，存在表 8-1 所示的等价关系，而且二者可以互用。但需要注意的是，a、a+1、a+2 只是常量地址，不是指针变量，不能参加诸如 a=a+1、a++等变量的运算。把引用数组名作为常量地址的方法称为数组名法。通常将引用数组下标表示数组元素的方法称为下标法。

表 8-1 数组名的操作与数组元素的等价关系

二者等价的关系	二者等价的关系
$\&a[0] \longleftrightarrow a$	$a[0] \longleftrightarrow *a$
$\&a[1] \longleftrightarrow a+1$	$a[1] \longleftrightarrow *(a+1)$
$\&a[2] \longleftrightarrow a+2$	$a[2] \longleftrightarrow *(a+2)$

8.3.2 指向数组元素的指针

一个数组包含若干个元素，每个数组元素都在内存中占用存储单元，它们都有相应的地

址。指针变量既可以指向变量，当然也可以指向数组元素。所谓数组元素的指针就是数组元素的地址。因为数组中的数组元素是普通变量，所以指向数组元素的指针变量就是普通指针变量。

例如，

```
int a[6], *p;
p=a          /* 把数组 a 的首地址赋给 p, 使 p 指向数组元素 a[0] */
```

也可以将 `p=a` 语句换成 `p=&a[0]`，二者是等价的。

可以通过指针变化 `p+1`、`p+2`、`p+3`、`p+4`、`p+5`，访问 `a[1]`、`a[2]`、`a[3]`、`a[4]`、`a[5]` 元素。

【例 8.15】输出数组的全部元素，如图 8-12 所示。

(1) 通过指针引用数组元素。

```
#include<stdio.h>
void main()
{int a[6]={1,3,5,7,9,11};
 int *p,i;
 printf("\n");
 for(p=a,i=0;p<a+6;p++,i++)
 printf("a[%d]=%d\n", i,*p);
}
```

(2) 通过常量指针引用数组元素。

```
#include<stdio.h>
void main()
{int a[6]={1,3,5,7,9,11};
 int i;
 printf("\n");
 for(i=0;i<6;i++)
 printf("a[%d]=%d\n", i,* (a+i));
}
```

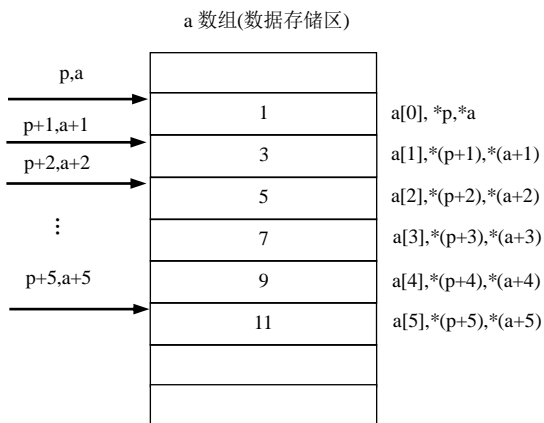


图 8-12 指针与数组元素示意图

两个程序的运行结果都为：

```
a[0]=1
a[1]=3
a[2]=5
a[3]=7
a[4]=9
a[5]=11
```

分析：两种方法可得到相同结果，区别是 `p` 为指针变量，本身可以直接累加，而 `a` 为指针常量，只能通过加常量，实现输出数组元素的值。C 编译系统处理数组就是采用方法（2），由于每次都要计算元素地址，费时较多；方法（1）用指针处理，不用每次重新计算地址，按次序自加移动是比较快的，可以大大提高效率。

在 C 语言中，指向数组的指针变量也可以带下标，如 `p[i]` 与 `*(p+i)` 等价。例 8.15 可以写成下面的形式，其结果是一样的。

```
#include<stdio.h>
void main()
```



```

{ int a[6]={1,3,5,7,9,11};
  int *p,i;
  printf("\n");
  p=a;
  for(i=0;i<6;i++)
    printf("a[%d]=%d\n", i,p[i]);
}

```

8.3.3 数组名作为函数参数

由前面知道, 指针可以作为函数的参数进行传递, 并可以返回多个值。数组名代表数组的首地址, 是个常量指针, 因此数组名也可以作为函数的参数进行传递。数组名作形参时, 接受实参数组的首地址; 数组名作实参时, 将数组的首地址传递给形参数组, 形参数组以此为首地址, 这样形参数组与实参数组就共享同一段内存区, 因此形参数组元素的变化必然影响实参数组元素同时发生变化, 并达到可以返回多个值的目的。如果令一个指针变量指向数组的首地址, 或等于数组名, 此时数组名和指针变量的含义相同, 都表示数组的首地址。所以在函数中地址的传递也可以不用数组名来进行, 而使用指向数组首地址的指针变量, 即实参和形参使用数组名时可以用指针变量替换。

【例 8.16】在一维数组 *a* 中, 要求将数组的第一个元素与最后一个元素交换, 第二个元素与倒数第二个元素交换, 依此类推。

数组中各元素的交换实质上是找下标之间的规律。例如, 解决的方法为, 首先找出数组下标的中间值, 若数组有 *n* 个元素, 则中间值为 $n/2$, 然后通过循环语句, 利用某个中间变量, 将数组的元素交换。

```

#include<stdio.h>
void swap(int x[],int n)
{int i,t;
  for(i=0;i<n/2;i++)
  { t=x[i];
    x[i]=x[n-i-1];
    x[n-i-1]=t;}
}
void main()
{int a[10]={1,3,5,7,9,2,4,6,8,10};
  int i;
  swap(a,10);
  printf("\n");
  for(i=0;i<10;i++)
    printf("a[%d]=%d  ", i,*(a+i));
}

```

运行结果为:

```

a[0]=10  a[1]=8  a[2]=6  a[3]=4  a[4]=2  a[5]=9  a[6]=7  a[7]=5  a[8]=3
a[9]=1

```

分析: 由图 8-13 可以看出, 由于数组 *a* 和数组 *x* 共享一段存储空间, 因此 *x* 数组元素值的变化, 就是 *a* 数组元素值的变化。

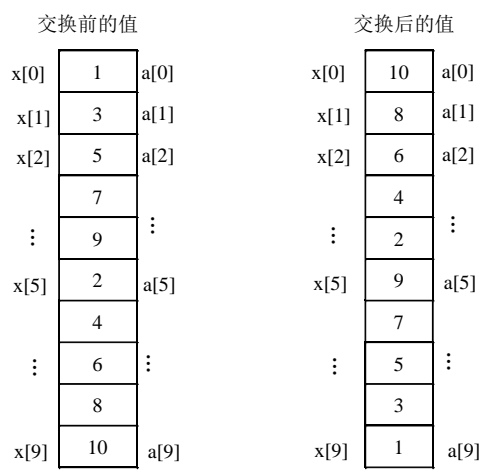


图 8-13 数组 a 和 x 的共享存储区

【例 8.17】已知一个一维数组 x[11]中有 10 个数，求出其中前 n 个数的和并放入 x[10]中。其中 n 由键盘输入。

```
#include<stdio.h>
void sum(int *q,int k)
{int i,s=0;
 int *t;
 t=q;
 /* t 与 q 都指向数组 x 的首地址 */
 for(i=0;i<k;i++,q++)
 s+=*q;
 /*求数组 x 前 n 项的和 */
 *(t+10)=s;
 /* t+10 指向数组 x 的 x[10]元素 */
}
void main()
{int x[11]={1,2,3,4,5,6,7,8,9,10};
 int i,n;
 printf("please input n=?(n<10)\n");
 scanf("%d",&n);
 sum(x,n);
 printf("x[10]=%d ",x[10]);
}
```

运行结果为：

5✓
x[10]=15

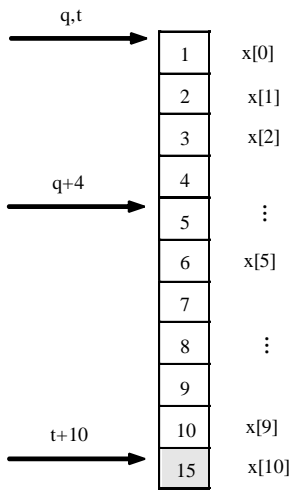


图 8-14 数组 x 的存储区

分析：此例与上例不同，上例函数的形参和实参都是数组名，而此例形参为指针变量。数组有下标的限制，指针变量却没有，因此在 sum 函数中用 i<k 来限定循环的次数。t 中保留了数组 x 的首地址，用*(t+10)=s 来给 x[10]赋值，如图 8-14 所示。

注意：在函数中对于数组地址的传递不止一种方法，可以有以下 4 种，如表 8-2 所示。

表 8-2 数组作为函数参数的几种情况

实 参	形 参
-----	-----

数组名	数组名
数组名	指向数组的指针变量
指向数组的指针变量	指向数组的指针变量
指向数组的指针变量	数组名

【例 8.18】已知一个一维数组 $x[10]$ 中有 10 个数，求出第 m 个数到第 n 个数的和。其中 m 、 n 由键盘输入。

```
#include<stdio.h>
int sum(int *q,int k).
{int i,s=0;
 for(i=0;i<k;i++,q++)
  s+=*q;
 return s;
}
void main()
{int x[10]={1,2,3,4,5,6,7,8,9,10};
 int ss,m,n,*p;
 printf("please input m and n (m<n<10):\n");
 scanf("%d,%d",&m,&n);
 p=x+m-1; /* 第 m 个元素的地址为 x+m-1 */
 ss=sum(p,n-m+1);
 /* 第 m 到第 n 元素的个数应为 n-m+1 */
 printf("%d\n",ss);
}
```

运行结果为：

```
4, 8 ✓
30
```

分析：数组的下标从 0 开始，第 m 个元素和第 n 个元素的下标分别为 $m-1$ 、 $n-1$ ，它们的地址应为下标加数组首地址，分别为 $x+m-1$ 、 $x+n-1$ ；第 m 个到第 n 个元素的个数应为 $n-m+1$ 。在本例中，是用指向数组元素的指针变量作函数的形参和实参，如图 8-15 所示。

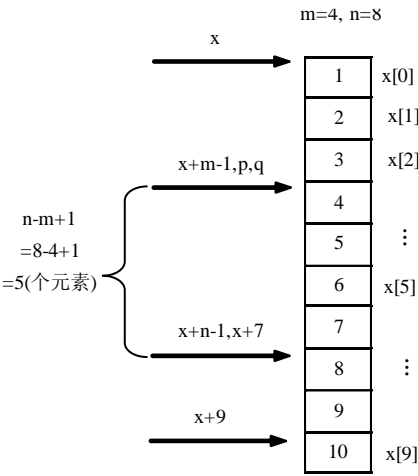


图 8-15 例 8.18 示意图

8.4 指针与字符串

字符串可以看成是字符数组，因此可以像用指针处理数组一样处理字符串。

8.4.1 指向字符串的指针变量

由前面我们知道，字符串是通过一维字符数组来存储的，其在内存中的首地址称为字符串的指针，因此，可以使用指向字符数组的指针变量来实现字符串的操作。下面用一维字符数组和指向字符数组的指针变量来实现字符串的操作。

【例 8.19】用一维字符数组来实现字符串的操作。

```
#include<stdio.h>
void main()
{char str[]="A red bag";
 printf("%s\n",str);
}
```

运行结果为:

A red bag

分析: 本例用的是字符数组, 字符数组与前面介绍的数组属性一样, 数组名 `str` 代表了字符数组的首元素地址 (见图 8-16), 实际上数组元素的操作也是用指针来进行的, `str[4]` 就是 `*(str+4)`, `str+4` 是 `str[4]` 的地址。

【例 8.20】用指向字符数组的指针变量来实现字符串的操作。

```
#include<stdio.h>
void main()
{char *str1="A red bag";
 printf("%s\n",str1);
}
```

运行结果为:

A red bag

分析: 虽然没有定义字符数组, 但实际上还是在内存开辟了一个连续的区域存放字符串常量 “A red bag” (见图 8-17), 在字符串尾加字符 “\0”, 作为字符串的结束标志, 然后将该区域的首地址赋给字符指针变量 `str1`, 就可对字符串进行操作了。

语句 `char *str1="A red bag";` 可以写成下列形式:

```
char *str1;
str1="A red bag";
```

字符指针变量 `str1` 中仅存字符串的首地址, 而字符串本身存储在内存中。一个字符指针变量只能指向一个字符数据, 不能同时指向多个字符数据, 更不能把字符串全部放到指针变量中。因此, 方法 2 中的定义语句写成如下形式是错误的。

```
char *str1;
*str1="A red bag"; /*错误的是将字符串全部放到指针变量中, 而指针变量只能存放地址*/
在用字符指针变量输出字符串时, 要用
printf("%s\n",str1);
```

字符串输出过程为, 首先输出字符指针 `str1` 所指向的字符数据 “A”, 然后自动使 `str1` 加 1, 使之指向下一个字符, 然后再输出 `str1` 所指向的字符数据空格 “ ” ……如此直到遇到字符串的结束标志 “\0” 为止。

通过指向字符串的指针, 可以对字符串进行整体的输入和输出。

对字符串中字符的存取可以用下标法、字符数组名法, 也可以用指针方法。

【例 8.21】比较三种方法输出。

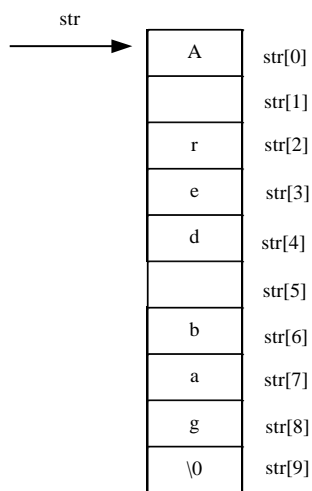


图 8-16 内存中字符串变量示例

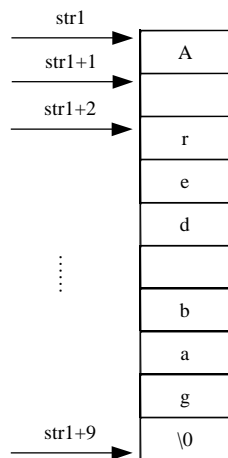


图 8-17 内存中字符串变量示例

```
#include<stdio.h>
void main()
{ char *string="I love China!";
  char str[]="I love China!";
  printf("%s\n", string);      /*输出整个字符串—指针法*/
  printf("%s\n", str);        /*输出整个字符串—数组名法*/
  printf("%c\n", string[0]);   /*输出单个字符—下标法*/
  printf("%c\n", str[0]);      /*输出单个字符—下标法*/
  printf("%c\n", *(string+3)); /*输出单个字符—指针法*/
  printf("%c\n", *(str+3));    /*输出单个字符—数组名法*/
}
```

运行结果:

```
I love China!
I love China!
I
I
o
o
```

由上例可看出, 字符数组名法和指针法引用的结果是相同的, 无论对数组名还是指针都可以引用下标, 其结果也是相同的。

【例 8.22】用指针法来实现将字符串 b 复制到字符串 a 的后边 (见图 8-18)。

```
#include<stdio.h>
void main()
{char a[15]="I love ",b[10]="China!";
  char *s1,*s2;
  for(s1=a;*s1!='\0';s1++); /*①在字符数组 a 中查找结束符'\0'的地址*/
  for(s2=b;*s2!='\0';)      /*将字符串 b 连接到字符串 a 的后边*/
  *s1++=*s2++;               /*②将字符串 b 的元素依次赋给字符数组 a 中对应的位置*/
  *s1='\0';                  /*③加字符串结束符*/
  printf("%s\n",a);
}
```

运行结果为:

```
I love China!
```

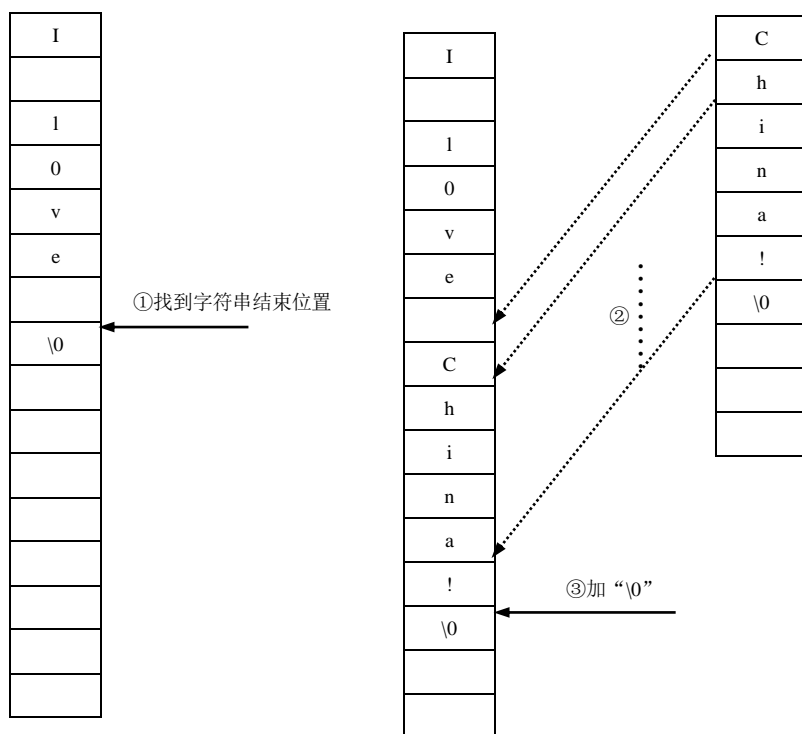


图 8-18 字符串合并示例

分析：

(1) 第一个 for 语句的作用是在字符数组 a 中找到结束符'\0'的地址。先将指针 s1 指向数组 a 的第一个元素，然后通过 s1++使指针逐个指向数组中后面的元素，直到遇到'\0'为止，此时 s1 就是字符串结束符'\0'的地址。

(2) 第二个 for 语句用来复制字符。根据运算符的优先级和结合性，*s1++和*s2++分别相当于*(s1++)和*(s2++)，也就是先令 s1 指向的变量等于 s2 指向的变量的值，然后 s1 和 s2 再进行自增。因此复制语句等价于：

```
s1=s2;
s1++;s2++;
```

(3) 最后的赋值语句是将字符串结束符'\0'放在连接后字符串的末尾。

思考：使用*(a+i)和*(b+i)的字符数组名法如何替换*s1++和*s2++完成上述程序？

8.4.2 字符串指针作为函数参数

字符数组（字符串）与数值数组一样，数组名都代表了数组的首地址，指向字符串的指针表示的也是字符串的地址，机器中字符数组的处理也是用指针计算的方法。因此，在 C 语言中，将一个字符串从一个函数传到另一个函数，可以用字符数组名和字符串指针作为函数的参数。在被调函数中可以改变字符串的内容，在主调函数中可以得到改变了的字符串。

【例 8.23】用字符数组名作为函数的参数来实现字符串的复制。

```
#include<stdio.h>
```

```

void copy1(char s1[],char s2[])
{ int i;
  for(i=0;s1[i]!='\0';i++)
    s2[i]=s1[i];          /*通过 i 值的变化,实现字符串的复制*/
    s2[i]='\0';           /*添加字符串结束标志*/
}
void main()
{char a[20]="I love China!",b[20]="daliang in China!";
  copy1(a,b);
  printf("%s----%s\n",a,b);
}

```

运行结果为:

I love China! --- I love China!

【例 8.24】用指针法来实现字符串的比较（见图 8-19）。

```

#include "stdio.h"
strcmp(char *sp1,char *sp2)
{
  while(*sp1==*sp2)      /*依次比较 sp1 和 sp2 所指字符 */
    if(*sp1=='\0') return 0; /*判断 sp1 所指字符串是否结束,若结束则返回 0*/
    sp1++;sp2++;
}
return *sp1-*sp2;        /*返回 sp1 和 sp2 所指字符的差值*/
}
main()
{int k;
  char *str1="nineteen";
  char *str2="ninety";
  k=strcmp(str1,str2);
  if(k==0)
    printf("str1==str2");
  else if(k>0)
    printf("str1>str2");
  else
    printf("str1<str2");
}

```

运行结果为:

str1<str2

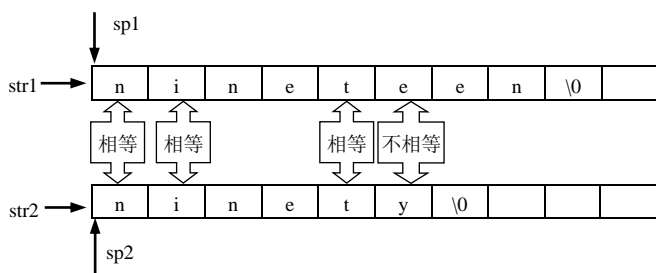


图 8-19 字符串比较示例

分析:

(1) 在函数 `strcmp` 中, 语句 `while(*sp1==*sp2)` 中的语句 `*sp1==*sp2` 用于比较 `sp1` 和 `sp2` 所指字符是否相等, 相等继续循环, 否则中止循环。

(2) `if` 语句的表达式 `*sp1=='\0'`, 用取 `sp1` 位置上的字符是否为 `'\0'` 来判断字符串是否结束, 若循环条件 `*sp1==*sp2` 成立, 且 `if` 条件 `sp1=='\0'` 成立, 可以知道两个字符串相等, 且都结束了, 函数返回 0 值。

(3) 若循环条件 `*sp1==*sp2` 不成立, 则跳出循环, 运行语句 `return *sp1-*sp2`, 返回所比较的两个字符的差值。

思考: 若将 `if` 条件 `*sp1=='\0'` 改成 `*sp2=='\0'`, 是否可以。

8.5 指针与多维数组

本章以二维数组为例介绍多维数组的应用。

8.5.1 多维数组的地址

由数组一章可知, 一个二维数组可以分解为若干个一维数组, 一维数组又由若干个元素组成。可以把二维数组看成是“数组的数组”。

设有如下二维矩阵, 它有 3 行、3 列数据:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

定义为:

```
int a[3][3]={ {1,2,3}, {4,5,6}, {7,8,9} }
```

二维数组在内存中是以行的形式按顺序存储的, 设数组 `a` 的首地址为 1000, 则其在内存中的存储排列示意图如图 8-20 所示。

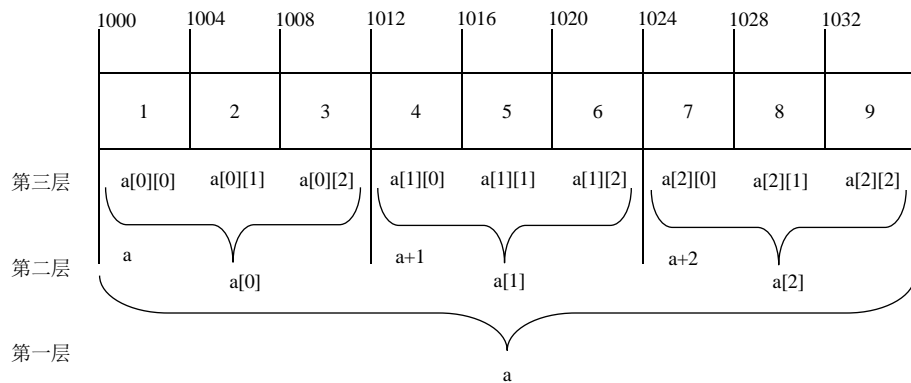


图 8-20 数组 `a` 在内存中的存储排列示意图

由图 8-20 可以看出,从二维数组的整体角度来看,可以把数组 a 分为三层,第一层为整体的二维数组 a ;第二层组成数组 a 的 3 个一维数组 $a[0]$ 、 $a[1]$ 、 $a[2]$;第三层组成一维数组 $a[0]$ 、 $a[1]$ 、 $a[2]$ 的各元素。

二维数组地址的编排也是按层组织的, a 是数组名,代表整个二维数组的首地址,也是一维数组 $a[0]$ 的首地址,等于 1000。那么 $a+1$ 就代表一维数组 $a[1]$ 的首地址 1006, $a+2$ 就代表一维数组 $a[2]$ 的首地址 1012。数组及数组元素的地址表示如表 8-3~表 8-6 所示。

表 8-3 数组 a 及组成数组 a 的各一维数组地址表示

地址 数组	首地址	第 0 行 一维数组 $a[0]$ 地址	第 1 行 一维数组 $a[1]$ 地址	第 2 行 一维数组 $a[2]$ 地址
二维数组 a	a	$a+0$	$a+1$	$a+2$

表 8-4 一维数组 $a[0]$ 及组成数组 $a[0]$ 的各元素地址表示

地址 数组	首地址	第 0 列 $a[0][0]$ 地址	第 1 列 $a[0][1]$ 地址	第 2 列 $a[0][2]$ 地址
一维数组 $a[0]$	$a+0$ $a[0]$ & $a[0]$	$*(a+0)$, $*a$ $a[0]$ & $a[0][0]$	$*(a+0)+1$ $a[0]+1$ & $a[0][1]$	$*(a+0)+2$ $a[0]+2$ & $a[0][2]$

表 8-5 一维数组 $a[1]$ 及组成数组 $a[1]$ 的各元素地址表示

地址 数组	首地址	第 0 列 $a[1][0]$ 地址	第 1 列 $a[1][1]$ 地址	第 2 列 $a[1][2]$ 地址
一维数组 $a[1]$	$a+1$ $a[1]$ & $a[1]$	$*(a+1)$ $a[1]$ & $a[1][0]$	$*(a+1)+1$ $a[1]+1$ & $a[1][1]$	$*(a+1)+2$ $a[1]+2$ & $a[1][2]$

表 8-6 一维数组 $a[2]$ 及组成数组 $a[2]$ 的各元素地址表示

地址 数组	首地址	第 0 列 $a[2][0]$ 地址	第 1 列 $a[2][1]$ 地址	第 2 列 $a[2][2]$ 地址
一维数组 $a[2]$	$a+2$ $a[2]$ & $a[2]$	$*(a+2)$ $a[2]$ & $a[2][0]$	$*(a+2)+1$ $a[2]+1$ & $a[2][1]$	$*(a+2)+2$ $a[2]+2$ & $a[2][2]$

说明:

(1) $a+0$ 、 $a+1$ 、 $a+2$ 与 $*(a+0)$ 、 $*(a+1)$ 、 $*(a+2)$ 都表示二维数组 a 第 0 列元素的地址,但两种表示方法的含义不同。前者从二维数组的整体角度出发, a 是二维数组名,由 3 个一维数组组成,因此 $a+0$ 、 $a+1$ 、 $a+2$ 代表了各一维数组的起始地址。后者从二维数组的层次角度出发,数组名 a 是数组的一层地址(称二级指针常量),一维数组的地址是二层地址(称一级指针常量),这是两级地址,若想得到二层地址,可以取一层地址的内容(见多级指针),即 $*(a+0)$ 、 $*(a+1)$ 、 $*(a+2)$ 。

(2) $a[0]$ 、 $a[1]$ 、 $a[2]$ 与 $\&a[0]$ 、 $\&a[1]$ 、 $\&a[2]$ 都表示一维数组的首地址,但两种表示方法

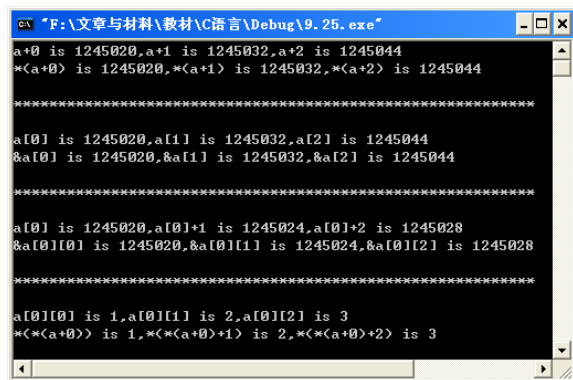
的含义不同。 $a[0]$ 、 $a[1]$ 、 $a[2]$ 是一维数组的数组名，也即是首地址。二维数组 a 由三个一维数组 $a[0]$ 、 $a[1]$ 、 $a[2]$ 组成，取 3 个一维数组的地址即为 $\&a[0]$ 、 $\&a[1]$ 、 $\&a[2]$ 。

(3) $a[0]$ 、 $a[0]+1$ 、 $a[0]+2$ 表示一维数组 $a[0]$ 各元素的地址，一维数组里各元素的存储是挨着存储的， $a[0]$ 是一维数组名，又是一维数组 $a[0]$ 的首地址，因此 $a[0]$ 、 $a[0]+1$ 、 $a[0]+2$ 代表元素 $a[0][0]$ 、 $a[0][1]$ 、 $a[0][2]$ 的地址。

【例 8.25】 验证二维数组的地址表示方式。

```
#include <stdio.h>
main()
{ int a[3][3]={1,2,3},{4,5,6},{7,8,9}};
  printf("a+0 is %u,a+1 is %u,a+2 is %u\n",a+0,a+1,a+2);
  printf("(*(a+0) is %u,*(a+1) is %u,*(a+2) is %u\n\n",*(a+0),*(a+1),*(a+2));
  printf("*****\n");
  printf("\na[0] is %u,a[1] is %u,a[2] is %u\n",a[0],a[1],a[2]);
  printf("&a[0] is %u,&a[1] is %u,&a[2] is %u\n\n",&a[0],&a[1],&a[2]);
  printf("*****\n");
  printf("\na[0] is %u,a[0]+1 is %u,a[0]+2 is %u\n",a[0],a[0]+1,a[0]+2);
  printf("&a[0][0] is %u,&a[0][1] is %u,&a[0][2] is %u\n\n",&a[0][0],&a[0][1],
    &a[0][2]);
  printf("*****\n");
  printf("\na[0][0] is %u,a[0][1] is %u,a[0][2] is %u\n",a[0][0],a[0][1],a[0][2]);
  printf("(*(a+0)) is %u,(*(a+0)+1) is %u,(*(a+0)+2) is %u\n\n",**a,*(a+1),
    *(a+2));
}
```

运行结果如图 8-21 所示。



```

F:\文章与材料\教材\C语言\Debug\9.25.exe
a+0 is 1245020,a+1 is 1245032,a+2 is 1245044
*(a+0) is 1245020,*(a+1) is 1245032,*(a+2) is 1245044
*****
a[0] is 1245020,a[1] is 1245032,a[2] is 1245044
&a[0] is 1245020,&a[1] is 1245032,&a[2] is 1245044
*****
a[0] is 1245020,a[0]+1 is 1245024,a[0]+2 is 1245028
&a[0][0] is 1245020,&a[0][1] is 1245024,&a[0][2] is 1245028
*****
a[0][0] is 1,a[0][1] is 2,a[0][2] is 3
*(a+0) is 1,*(a+0)+1 is 2,*(a+0)+2 is 3

```

图 8-21 例 8.25 的运行结果

8.5.2 多维数组元素的引用方法

多维数组与一维数组相似，都可以通过下标法、数组名法以及指针法引用数组元素。

1. 下标法

利用数组元素的下标来引用数组元素，例如： $a[0][0]$ 、 $a[2][1]$ 等。

2. 数组名法

数组名为常量指针。二维数组是由若干个一维数组组成的，对于二维数组而言，数组名相当于二级指针常量，如例 8.25 中的 `a`、`a+1` 和 `a+i` 等。而组成二维数组的一维数组名相当于一级指针常量，如例 8.25 中的 `a[0]`、`a[0]+1`、`a[1]` 和 `a[i]+j` 等。

通过一级指针常量引用二维数组元素为 `*(a[0])`、`*(a[0]+1)`、`*(a[1])` 和 `*(a[i]+j)` 等。通过二级指针常量可得到一级指针常量，如 `*(a+0)`、`*(a+1)`、`*(a+i)` 和 `*(a+i)+j` 等。通过二级指针常量引用二维数组元素为 `*(*(a+0)+0)`、`*(*(a+1)+0)`、`*(*(a+i)+0)` 和 `*(*(a+i)+j)` 等。

3. 指针法

用指针法引用二维数组元素分为以下两种方式。

(1) 用简单指针变量引用二维数组元素

可以用简单指针变量指向组成二维数组的一维数组（即获得一级指针），从而实现引用二维数组元素的目的。

【例 8.26】用简单指针变量引用二维数组元素。

```
#include <stdio.h>
main()
{
    int a[2][4]={{1,2,3,4},{5,6,7,8}};

    int i,*p=a[0];
    for(i=1;p<a[0]+8;p++,i++)
    {
        printf("%4d",*p);
        if(i%4==0)
            printf("\n");
    }
}
```

运行结果为：

1 2 3 4

5 6 7 8

指针 `p` 的变化过程如图 8-22 所示。

思考：假如将语句 `*p=a[0]` 换成 `*p=a` 是否可以？

(2) 用指向二维数组一整行的指针变量引用二维数组元素

由于二维数组可分解为若干个一维数组，且存储方式又为一个一维数组一行，因此可利用指针指向二维数组的一整行，即二级指针来引用二维数组元素。指向二维数组一整行的指针变量也可称做二维数组的指针变量。二维数组的指针变化以行为单位。

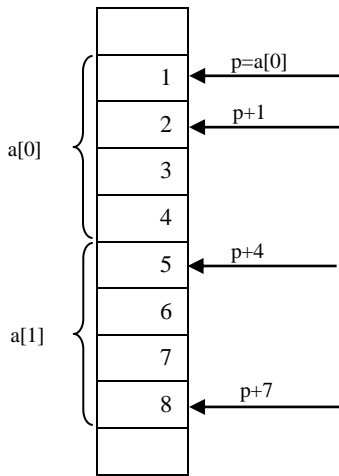


图 8-22 指针 `p` 的变化过程

二维数组指针变量说明的一般形式如下。

格式：

类型说明符 (*指针变量名) [长度]

说明：

- ① “类型说明符”为所指数组的数据类型。
- ② “(*指针变量名)”表示“*”后的变量是指针类型，且两边的括号不可少。
- ③ “长度”表示二维数组分解为多个一维数组时，一维数组的长度也就是二维数组的列数。

例如：设 p 为指向二维数组的指针变量。可定义为：

```
int a[4][4];
int (*p)[4];
p=a;
```

它表示 p 是一个指针变量，它指向包含 4 个整型元素的一维数组。也可以说，它指向二维数组 a 或指向第一个一维数组 $a[0]$ ，其值等于 a 。而 $p+1$ 指向一维数组 $a[1]$ ， $p+i$ 指向一维数组 $a[i]$ 。 p 的变化如图 8-23 所示。

指针变量 p 就相当于数组名 a ，只不过 p 是指针变量，其本身可以变化。而数组名 a 是指针常量，其本身不能变化。二者对于数组元素的引用方式也相同。例如， $*(a+i)+j$ 和 $*(p+i)+j$ 都是二维数组 i 行 j 列的元素地址，而 $*(p+i)+j$ 和 $*(a+i)+j$ 则是 i 行 j 列元素的值。

【例 8.27】以行为单位输出二维数组的元素。

```
#include <stdio.h>
main()
{
    int a[4][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int (*p)[4];
    int i;
    for(p=a;p<a+4;p++)
        for(i=0;i<4;i++)
        {
            printf("%6d",*(p+i));
            if((i+1)%4==0)
                printf("\n");
        }
}
```

程序运行结果为：

```
1      2      3      4
5      6      7      8
9      10     11     12
13     14     15     16
```

分析：由于 p 是指向一行的指针，它的加 1 操作是以行为单位的，因此外循环以 p 为循环变量，若 $p++$ ，则 p 向下移动一行。内循环是输出一行内的 4 个元素，由于 p 是二级指针且初值为 a ，因此取一维数组的第 1 个元素的地址（一级指针）为 $*p$ ，第 2 个元素的地址为 $*p+1$ ，第 i 个元素的地址为 $*p+i$ 。由地址取存储内容为在地址前加一个*即可，如 $*(p+i)$ 。行与列指针的变化如图 8-23 所示。

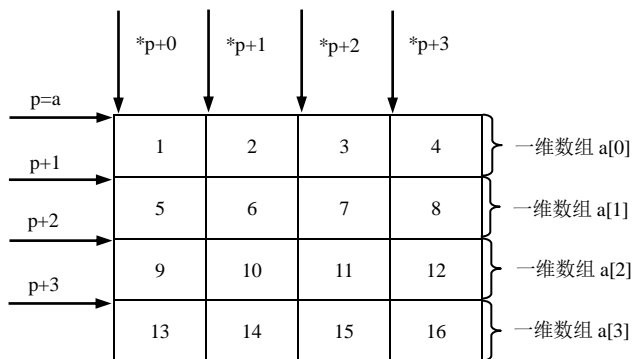


图 8-23 二维数组 a 中行与列指针的变化

对于语句 `int(*p)[4]`, 也可以理解为 `p` 是指向一行整体的指针, `*p` 由 4 个指针元素组成, `(*p)[0]` 是指向一行中第 1 个元素的指针, `(*p)[1]` 是指向一行中第 2 个元素的指针, `(*p)[2]` 是指向一行中第 3 个元素的指针, `(*p)[3]` 是指向一行中第 4 个元素的指针, 如图 8-24 所示。

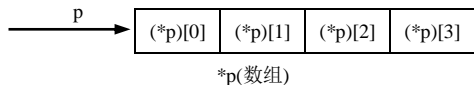


图 8-24

8.6 指针数组与多级指针

本节将介绍指针数组与多级指针, 指针数组与多级指针一般用于二维数组及字符串数组的应用。指针数组也可以作为 `main` 函数的参数。

8.6.1 指针数组

当数组的元素类型为某指针类型时, 该数组即为指针数组。其定义形式如下。

格式:

类型说明符 *数组名 [数组长度]

例如:

```
int *p[3];
```

表示数组 `p` 有 3 个元素, 即 `p[0]`、`p[1]`、`p[2]`, 且 3 个元素都是指向整型变量的指针。

一般用指针数组处理二维数组或字符串数组。例如, 用指针数组处理例 8.27。

【例 8.28】 以行为单位输出二维数组的元素。

```
#include <stdio.h>
main()
{
    int a[4][4]={1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int *p[4]={a[0],a[1],a[2],a[3]};
    int i,j;
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
        {
```

```
        printf("%6d",*(p[i]+j));
        if((j+1)%4==0)
            printf("\n");
    }
}
```

程序运行结果为:

```
1      2      3      4
5      6      7      8
9      10     11     12
13     14     15     16
```

分析: **p** 数组有 4 个元素, 分别为 **p[0]~ p[3]**, 取值依次为 **a[0]、a[1]、a[2]、a[3]**, 分别是 4 个一维数组的首地址 (第 1 个元素的地址), 第 **i** 行的首地址为 **p[i]**, 第 **i** 行第 **j** 列元素的地址为 **p[i]+j**。由地址取存储内容为 ***(p[i]+j)**。

用指针数组处理字符串数组时, 使指针数组的每一个元素分别代表字符串数组中各字符串的首地址, 便于字符串的操作。

【例 8.29】用指针数组输出 5 个城市名。

```
#include <stdio.h>
main()
{
    char *s[5]={"Dalian","Beijing","Shanghai","Tianjin","Chongqing"};
    int j;
    for(j=0;j<5;j++)
        printf("%s ",s[j]);
}
```

运行结果为:

```
Dalian Beijing Shanghai Tianjin Chongqing
```

看语句 **char *s[5]={"Dalian","Beijing","Shanghai","Tianjin","Chongqing"};**似乎是把 5 个城市名赋给了指针数组 **s** 的各元素, 实质上数组 **s** 中只存储了指针, 各指针指向每个城市名字符串的首地址, 如图 8-25 所示。

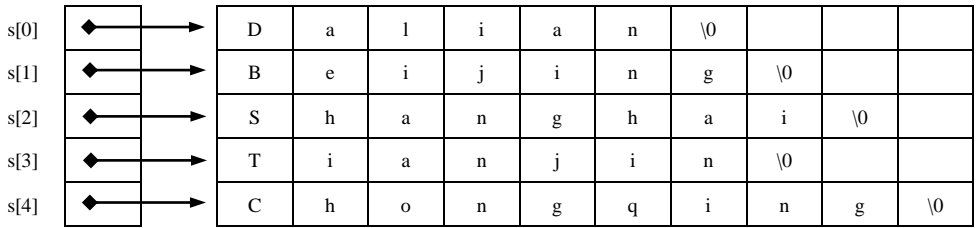


图 8-25 指针数组 **s** 指向字符串的示意图

【例 8.30】输入 5 个城市名并按字母顺序输出。

```
#include <stdio.h>
#include <string.h>
void sort(char *s[],int n)
{char *p;
 int i,j,k;
```

```

for(i=0;i<n-1;i++)
{ k=i;
  for(j=i+1;j<n;j++)
    if(strcmp(s[k],s[j])>0)
      k=j;
  if(k!=i)
    {p=s[i];s[i]=s[k];s[k]=p;}
}
}
main()
{ char *s[]={ "Dalian","Beijing","Shanghai","Tianjin","Chongqing"};
  int j,n=5;
  sort(s,n);
  for(j=0;j<5;j++)
    printf("%s ",s[j]);
}

```

运行结果为:

Beijing Chongqing Dalian Shanghai Tianjin

分析: 本例用函数 `sort` 实现字符串的排序。`sort` 函数中排序选用的是选择法, 与本书前面介绍的一样, 只不过本例用于字符串的排序。调用 `sort` 函数时, 将实参指针数组的首地址传递给形参, 形参指针数组 `s` 与实参指针数组 `s` 共享相同的内存单元。

8.6.2 多级指针

如果一个指针变量存放的是另一个指针变量的地址, 则称这个指针变量为指向指针的指针变量。可以将前面介绍的存放简单变量地址的指针称为一级指针, 将指向指针的指针变量称为二级指针。指针与所指向的变量的关系如图 8-26 所示。

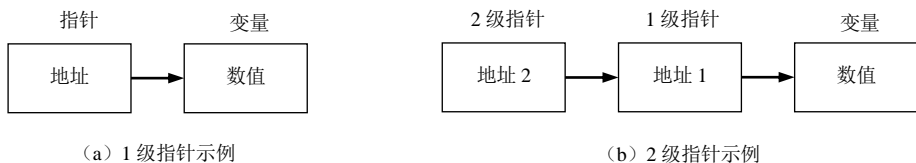


图 8-26 指针与所指向的变量的关系

指向指针的指针变量的定义形式如下。

格式:

类型说明符 **变量名

例如:

```
int **pp;
```

下面通过例子来介绍变量、1 级指针和 2 级指针三者的关系。

【例 8.31】 变量、1 级指针和 2 级指针三者关系示例。

```

#include <stdio.h>
main()
{ int x=5,*p,**pp;
  p=&x;
  pp=&p;
}

```

```

printf("**pp=%u , *pp=%u ,pp=%8u\n",**pp,*pp,pp);
printf("**p=%u , p=%9u ,&p=%8u\n",*p,p,&p);
printf("x=%u , &x=%8u\n\n",x,&x);
}

```

运行结果如图 8-27 所示

分析：由运行结果可见， x 、 $*p$ 、 $**pp$ 三者的值相等，都是 5； $\&x$ 、 p 、 $*pp$ 三者的值相等，都是 x 的地址； $\&p$ 、 pp 二者的值相等，都是 p 的地址。

变量、1 级指针和 2 级指针三者之间存在以下关系。

(1) 1 级指针 p 存的是变量 x 的地址；2 级指针 pp 存的是 1 级指针 p 的地址。

(2) 对 1 级指针 p 取内容操作（即 $*p$ ），结果为 x 的值；对 2 级指针 pp 取内容操作（即 $*pp$ ），是 1 级指针 p 的值（即 $\&x$ ）。

(3) 对 2 级指针 pp 取两次内容操作（即 $**pp=(*pp)=*p=x$ ， $*$ 的结合性是由右到左），相当于对 1 级指针 p 取内容操作（即 $*p$ ），为 x 的值。

三者关系如图 8-28 所示。

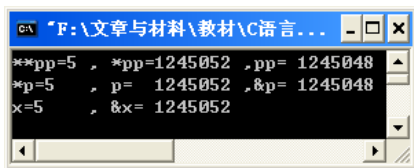


图 8-27 例 8.30 的运行结果

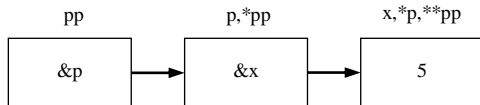


图 8-28 x 、 p 和 pp 三者的关系

一般指向指针的指针多用于二维数组与字符串数组的应用。

【例 8.32】改进例 8.28，以行为单位输出二维数组的元素。

```

#include <stdio.h>
main()
{
    int a[4][4]={1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int *p[4]={a[0],a[1],a[2],a[3]};
    int j,**q;
    for(q=p;q<p+4;q++)
        for(j=0;j<4;j++)
        {
            printf("%6d",*(q+j));
            if((j+1)%4==0)
                printf("\n");
        }
}

```

运行结果为：

```

1      2      3      4
5      6      7      8
9      10     11     12
13     14     15     16

```

分析：把指针数组 p 的首地址赋给 q ， q 首先指向 p 数组的第一个元素 $p[0]$ ， $q+i$ 指向于 p 数组的第 i 个元素 $p[i]$ ， $*q$ 是 q 所指向数组 p 某个元素 $p[i]$ 的值，即一维数组 $a[i]$ 的首地址，而 $*q+j$ 是一维数组 $a[i]$ 第 j 列元素的地址， $*(q+j)$ 是一维数组 $a[i]$ 第 j 列元素的数值。指针数组 p 与双重指针 q 的关系如图 8-29 所示。

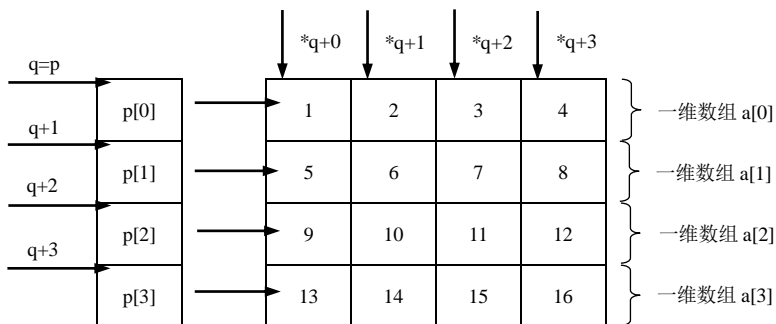


图 8-29 指针数组与双重指针的关系

【例 8.33】改进例 8.29，用指向指针的指针输出 5 个城市名。

```
#include <stdio.h>
main()
{   char *s[5]={"Dalian","Beijing","Shanghai","Tianjin","Chongqing"};
    int j;
    char **p;
    for(j=0,p=s;j<5;j++,p++)
        printf("%s ",*p);
}
```

运行结果为：

Dalian Beijing Shanghai Tianjin Chongqing

分析：第一次循环，把指针数组 s 的首地址赋给 p ， p 首先指向 s 数组的第一个字符串 $s[0]$ ， $*p$ 是 $s[0]$ 的值，即第一个字符串。第二次循环时 $p++$ ，此时 p 指向 s 数组的第二个字符串 $s[1]$ ， $*p$ 是 $s[1]$ 的值，即第二个字符串，其他依此类推。

指向指针的指针可以有多重，但重数越多，越难理解，因此一般应用不超过两重。

8.6.3 指针数组作为 main 函数的形参

在前面介绍的程序中，主函数 $\text{main}()$ 的圆括号中都是不带参数的，其实圆括号中是可以带参数的，而且参数可以是指针数组。例如，

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

说明： argc 和 argv 即为主函数 main 的形参。 argc 代表命令行参数的个数， argv 为指向字符串的指针数组。

当 C 程序生成“.exe”的可执行文件时，DOS 系统就可以以命令行的形式运行该可执行文件。由于 C 程序的运行是由主函数 main 开始的，因此可以在运行该可执行文件时传递参数给主函数。 argc 和 argv 两个参数的类型是固定的，不能改的。就像是 DOS 系统的其他命令一样。Dos 系统命令行的一般形式如下。

格式：

盘符:\>命令名 参数 1 参数 2 ...参数 n

说明：命令名与各参数之间用空格分开。命令名是主函数 main 所在的文件名，文件名后

各参数是需要传给主函数 `main` 的数据。

【例 8.34】改进例 8.30，输入任意个城市名并按字母顺序输出。

```
/* file1.c */
#include <stdio.h>
#include <string.h>
void main(int argc, char *argv[])
{ char *p;
  int i, j, k;
  for(i=1; i<argc-1; i++)
  { k=i;
    for(j=i+1; j<argc; j++)
      if(strcmp(argv[k], argv[j])>0)
        k=j;
    if(k!=i)
    {p= argv[i]; argv[i]=argv[k]; argv[k]=p;}
  }
  for(j=1; j<argc; j++)
    printf("%s ", argv[j]);
}
```

首先将程序 `file1.c` 进行编译，然后运行，即可生成可执行文件 `file1.exe`。然后在命令行窗口或 DOS 窗口中输入以下命令：

盘符路径>`file1.exe Haerbin Dalian`✓

其中，盘符路径为 `file1.exe` 所在的盘符和路径，例如，`c:\>file1.exe Haerbin Dalian`。文件 `file1` 可加 `.exe`，也可不加。

运行结果为：

Dalian Haerbin

在本例中，实质上是将例 8.30 中的 `sort` 函数改成了主函数 `main`，`main` 函数的参数由命令行的参数传送。`argc` 代表命令行中参数的个数，本例中命令行有 3 个参数，分别是 `file1.exe`、`Haerbin`、`Dalian`，因此 `argc` 的值是 3。`argv` 为指向字符串的指针数组，指针数组中存放的是 `file1.exe`、`Haerbin` 和 `Dalian` 这 3 个参数的首地址，如图 8-30 所示。

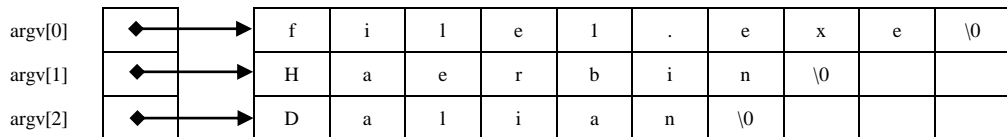


图 8-30 指针数组 `argv` 指向字符串的示意图

由于主函数的参数 `argc` 和 `argv` 由命令行的内容及个数决定，因此运行文件时，输入的参数除了文件名之外，有多少个城市名就可对多少个城市名进行排序输出。参数 `argc` 和 `argv` 的名字不是唯一的，可以任意变换，但一般约定用 `argc` 和 `argv`。

8.7 指针与函数

指针与函数的关系，归纳起来主要有以下 3 种：

- 指针作为函数参数（本章前面已经介绍，在此不再赘述）。
- 指向函数的指针。

- 返回指针的函数。

8.7.1 指向函数的指针

在C语言中，一个函数总是占用一段连续的内存单元，而函数名就是该函数所占连续的内存单元的首地址。可以把函数的这个首地址（或称入口地址）赋予一个指针变量，使该指针变量指向该函数。然后通过该指针变量就可以找到并调用这个函数。把这种指向函数的指针变量称为“指向函数的指针”或“函数指针变量”。

1. 函数指针变量的定义

格式：

类型说明符 (*指针变量名) ([形参类型表]);

说明：

- (1) “类型说明符”表示指针所指向函数的返回值类型。
- (2) “(* 指针变量名)”表示“*”后面的变量是定义的指针变量。
- (3) 形参类型表与指针所指向函数的形参表中的形参个数、类型及顺序应一致。各形参类型之间用“,”分隔。

例如：int (*pf)(int,int);

表示 pf 是一个指向函数入口的指针变量，该函数的返回值（函数值）是整型，且有两个整型参数要传递。

2. 用函数指针调用函数

若要用函数指针实现对函数的调用，首先应将函数的入口地址赋给函数指针，就是使函数指针指向被调函数，然后才可以调用函数。用函数指针调用函数的形式如下。

格式：

(*指针变量名) ([实参表]);

例如：z = (*pf)(a,b);

看下面通过用函数指针实现对函数调用的例子。

【例 8.35】找出 a、b 两个数中的最大值，计算 a!+b!的和。

```
#include <stdio.h>
int max(int a,int b)
{ return a>b?a:b;}
int fct(int a,int b)
{ int i,j,s1=1,s2=1;
  for(i=1;i<=a;i++) s1=s1*i;
  for(j=1;j<=b;j++) s2=s2*j;
  return s1+s2;
}
main()
{ long (*pf)(int,int);      /*定义函数指针 pf */
  int a,b;
  long z1,z2;
  pf=max;                  /*将 max 函数的入口地址赋给函数指针 pf*/
  printf("input two numbers:\n");
  scanf("%d%d",&a,&b);
```

```

    z1=(*pf)(a,b);          /*用函数指针 pf 调用函数 max*/
    printf("max=%d\n",z1);
    pf=fct;                 /*将 fct 函数的入口地址赋给函数指针 pf*/
    z2=(*pf)(a,b);          /*用函数指针 pf 调用函数 fct*/
    printf("a!+b!=%ld\n",z2);
}

```

运行过程与结果为:

```

input two numbers
3 5 ✓
max=5
a!+b!=126

```

使用函数指针变量应注意以下两点:

(1) 函数指针变量不能进行算术运算,这与其他指针变量不同。其他指针变量加减一个整数可使指针移动指向后面或前面的数据,而函数指针的移动是毫无意义的。

(2) 在函数调用中“(*指针变量名)”的两边的括号不可少,其中的*不应该理解为求值运算,在此处它只是一种表示符号。

3. 函数指针的应用

在例 8.35 中调用最大值与阶乘和函数,需要分别将两个函数的入口地址赋给函数指针变量,然后再分别调用这两个函数。其实若将函数指针作为函数的参数就可以解决多次将函数的入口地址赋给函数指针变量的问题。函数指针作为函数的参数是 C 语言的深入的应用,在此做简单的介绍。看下面的例子。

【例 8.36】改进例 8.35,找出 a、b 两个数中的最大值、计算 a!+b!的和,计算 a 的 b 次幂。

本例用函数来解决。max()用于找最大值,fct()用于计算 a!+b!的和,power()用于计算 a 的 b 次幂,pfun()的功能是将主函数对 max()、fct()和 power() 这 3 个函数的调用转化为主函数对 pfun()的调用。

```

#include <stdio.h>
long max(int a,int b)
{ return a>b?a:b;}
long fct(int a,int b)
{ int i,j,s1=1,s2=1;
  for(i=1;i<=a;i++) s1=s1*i;
  for(j=1;j<=b;j++) s2=s2*j;
  return s1+s2;
}
long power(int a,int b)
{ int i;
  long s1=1;
  for(i=1;i<=b;i++) s1=s1*a;
  return s1;
}
long pfun(int a,int b,long (*pf)(int ,int )) /*定义函数 pfun, 函数指针 pf 作为
它的参数*/
{ return (*pf)(a ,b ); }
main()

```

```

{ int a,b;
  printf("input a=? :\n");
  scanf("%d",&a);
  printf("input b=? :\n");
  scanf("%d",&b);
  printf("max=%ld\n",pfun(a,b,max));
  printf("a!+b!=%ld\n",pfun(a,b,fct));
  printf("power=%ld\n",pfun(a,b,power));
}

```

运行过程与结果为:

```

input a=?
3✓
input b=?
5✓
max=5
a!+b!=126
power=243

```

主函数中 3 次调用了函数 pfun。调用 pfun(a,b,max) 时, 其参数传递的过程如图 8-31 所示。

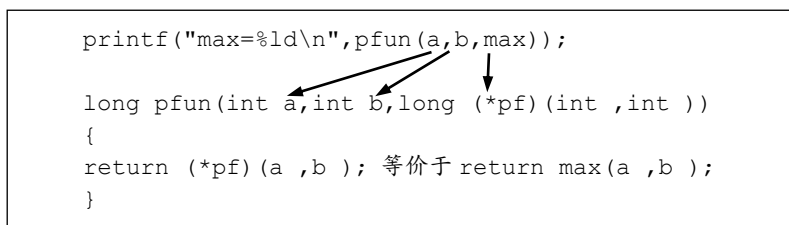


图 8-31 参数传递的过程示意图

从上面的例子可以看出, 不论调用 maxt()、fct()或 power(), 函数 pfun 一点都没有改动, 只是在调用函数 pfun 时将实参函数名改变了而已, 这就增加了函数使用的灵活性。

8.7.2 返回指针值的函数

在 C 语言中, 允许一个函数的返回值是一个指针 (即地址), 这种返回指针值的函数称为指针型函数。定义指针型函数的一般形式为:

格式:

```

类型说明符 *函数名 (形参表)
{ 函数体 }

```

其中函数名之前加了 “*” 号表明函数的返回值是指针类型, 这个函数就是指针型函数。类型说明符表示了返回的指针值所指向的数据类型。例如:

```

int *fp(int x,int y)
{ }

```

表示 fp 是一个返回指针值的指针型函数, 它返回的指针指向一个整型变量。

【例 8.37】 利用指针型函数编程, 输入一个 1~7 之间的整数, 输出对应的星期名。

```

#include <stdio.h>

```

```

main()
{ int i;
  char *day(int n);
  printf("please input i (0<i<8) :\n");
  scanf("%d",&i);
  printf("Week No: %2d is %s\n",i,day(i));
}
char *day(int n)
{ char *name[]={ "Illegal day", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday", "Sunday"};
  return (n<1||n>7)?name[0]:name[n];
}

```

运行过程与结果为:

```

please input i (0<i<8) :
5✓
Week No: 5 is Friday

```

本例中定义了一个指针型函数 `day`，它的返回值指向一个字符串。该函数中定义了一个指针数组 `name`。`name` 数组初始化赋值为 8 个字符串，分别表示各个星期名及出错提示。形参 `n` 表示与星期名所对应的整数。在主函数中，把输入的整数 `i` 作为实参，在 `printf` 语句中调用 `day` 函数并把 `i` 值传送给形参 `n`。`day` 函数中的 `return` 语句包含一个条件表达式，`n` 值若大于 7 或小于 1，则把 `name[0]` 指针返回主函数，输出出错提示字符串 “Illegal day”；否则返回主函数，输出对应的星期名。

注意：应区分开函数指针变量和指针型函数在写法和用法上的不同。

习 题

一、选择题

- 若有说明：`int n=2,*p=&n,*q=p;`，则以下非法的赋值语句是：
 - `p=q;`
 - `*p=*q;`
 - `=*q;`
 - `p=n;`
- 若有说明语句：`int a,b,c,*d=&c;`，则能正确从键盘读入 3 个整数分别赋给变量 `a`、`b`、`c` 的语句是：
 - `scanf("%d%d%d",&a,&b,d);`
 - `scanf("%d%d%d",&a,&b,&d);`
 - `scanf("%d%d%d",a,b,d);`
 - `scanf("%d%d%d",a,b,*d);`
- 若定义：`int a=511,*b=&a;`，则 “`printf("%d\n",*b);`” 的输出结果为
 - 无确定值
 - `a` 的地址
 - 512
 - 511
- 若有说明：`int i,j=2,*p=&j;`，则能完成 `i=j` 赋值功能的语句是
 - `i=*p;`
 - `p*=&j;`
 - `i=&j;`
 - `i==*p;`
- 下列程序的输出结果是

```

main()
{ char a[10]={9,8,7,6,5,4,3,2,1,0},*p=a+5;
  printf("%d",*--p);
}

```

- A. 非法 B. a[4]的地址 C. 5 D. 3

二、填空题

1. 以下函数的功能是,把两个整数指针所指的存储单元中的内容进行交换,将语句补充完整。

```
exchange(int *x, int *y)
{ int t;
  t=*y; *y= _____; *x=_____;
```

2. 下面程序是把数组元素中的最大值放入 a[0]中,则在 if 语句中的条件表达式应该是什么?

```
#include <stdio.h>
main( )
{ int a[10]={6,7,2,9,1,10,5,8,4,3},*p=a,i;
  for(i=0;i<10;i++,p++)
    if(_____) *a=*p;
  printf("%d\n",*a);
}
```

3. 以下函数用来求出两整数之和,并通过形参将结果传回,将语句补充完整。

```
void func(int x,int y, _____z)
{ *z=x+y; }
```

4. 若有以下定义,则不移动指针 p,且通过指针 p 引用值为 98 的数组元素的表达式是_____。

```
int w[10]={23,54,10,33,47,98,72,80,61}, *p=w;
```

5. 设有定义: int n,*k=&n;, 以下语句将利用指针变量 k 读写变量 n 中的内容,将语句补充完整。

```
scanf("%d" , _____ );
printf("%d\n",_____);
```

三、读程题。给出以下程序的输出结果。

- 1.

```
void fun(char *c,int d)
{ *c=*c+1;d=d+1;
  printf("%c,%c,",*c,d);
}
main()
{ char a='A',b='a';
  fun(&b,a); printf("%c,%c\n",a,b);
}
```

- 2.

```
void ss(char *s,char t)
{ while(*s)
{ if(*s==t) *s=t-'a'+'A';
  s++;}}
main()
{ char str1[100]="abcddefefdbd",c='d';
  ss(str1,c); printf("%s\n",str1);
}
```

- 3.

```
char cchar(char ch)
```

```

{ if (ch>='A' && ch<='Z') ch=ch-'A'+'a';
return ch;}
main()
{ char s[]="ABC+abc=defDEF", *p=s;
while (*p)
{ *p=cchar(*p);
p++;}
printf("%s\n", s);
}

```

4.

```

#include <stdio.h>
#include <string.h>
main()
{ char b1[8]="abcdefg", b2[8], *pb=b1+3;
while (--pb>=b1) strcpy(b2, pb);
printf("%d\n", strlen(b2));
}

```

5.

```

main()
{ char *p="abcdefgh", *r;
long *q;
q=(long*)p;
q++;
r=(char*)q;
printf("%s\n", r);
}

```

四、编程题

1. 用函数与指针来完成：由键盘输入 3 个数，按由小到大的顺序排序并显示出来。
2. 用函数与指针来完成：由键盘输入 3 个字符串，按由小到大的顺序排序并显示出来。
3. 用函数与指针来完成：把 b 字符串连接到 a 字符串的后面，并返回 a 中新字符串的长度。
4. 用函数与指针来完成：通过调用函数返回一维数组中的最大值。
5. 用函数与指针来完成：对由 10 个数组成的一维数组中任意个连续的元素按由大到小的顺序排序，并在主调函数中输出。