第6章指针

学习目标

- 掌握指针与指针变量的概念
- 了解指针与数组的关系，掌握如何使用指针引用数组中的数据
- 了解指针与函数的关系，掌握如何通过指针调用函数
- 掌握指针与二级指针的关系
- 掌握内存的申请方法与操作方式

指针是 C 语言中一种特殊的变量类型，与其它类型的变量不同，指针变量存储的不是变量，而是变量的地址。正确地使用指针，可以使程序更为简洁紧凑，高效灵活。指针是 C 语言的精髓，同时也是 C 语言中最难掌握的一部分。

【案例 1】爸爸在哪儿

案例描述

晚餐时间，妈妈做好了美味的晚餐，走上楼去叫宝宝和爸爸吃饭。到了卧室，发现只有宝宝一个人，妈妈想：“爸爸在哪儿？”。妈妈先让宝宝下楼去餐桌旁，然后走到了书房，在书房找到了正在看书的爸爸。

如果将宝宝和爸爸比作内存中的两个变量，请编程求出他们在内存中的地址。

案例分析

在计算机中，每一个变量都是有地址的，根据地址就能找到某个变量。如在本案例中，宝宝在卧室，则宝宝的地址就是卧室；爸爸在书房，则爸爸的地址就是书房。

根据案例描述，妈妈首先在卧室中找到了宝宝，之后在书房中找到了爸爸。寻找宝宝和寻找爸爸的步骤分别如图 6-1 所示。

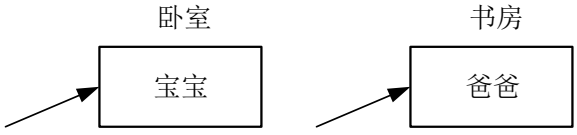


图6-1　步骤示意图

在这个寻找的过程中涉及到了指针与指针变量的相关知识，下面对这些知识逐一讲解。

必备知识

1. 指针与指针变量

（1）指针的概念

如果在程序中定义一个 int 型的变量 a：

```
int a=10;
```

那么编译器会根据变量 a 的类型 int，为其分配 4 个字节地址连续的存储空间。若这块连续空间的首地址为 0x0037FBCC，那么这个变量占据 0x0037FBCC~0x0037FBD0 这四个字节的空间，0x0037FBCC 就是这个变量的地址。因为通过变量的地址可以找到该变量所在的存储空间，所以说该变量的地址指向该变量所在的存储空间，该地址是指向该变量的指针。内存单元和地址的关系示例如图 6-2 所示。

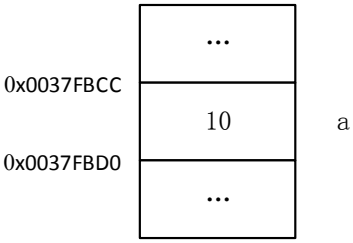


图6-2 内存单元和地址

若将存储空间视为酒店，那么存储单元好比酒店中的房间，地址好比酒店中房间的编号，而存储空间中存储的数据就相当于房间中的旅客。

（2）指针变量的定义

指针指示某个变量所在的存储空间，相应地，指针变量存储这个指针。定义指针变量的语法格式如下：

```
变量类型* 变量名
```

上述语法格式中，变量类型指定定义的指针指向数据的类型，变量名前的符号“*”表示该变量是一个指针变量。举例说明：

```
int* p;                    //定义一个 int*型的指针变量 p
```

其中“*”表明 p 是一个指针变量，int 表明该指针变量指向一个 int 型数据所在的地址。

（3）指针变量初始化

指针变量的赋值有两种方法，一种是接收变量的地址为其赋值，如下所示：

```
int a=10;                  //定义一个 int 型的变量 a
int* p;                    //定义一个 int*型的指针变量 p
p=&a;                      //使 int*型的指针变量 p 指向 int 型变量 a 所在的存储空间
```

另外一种是与其它指针变量指向同一块存储空间：

```
int* q;                    //定义一个 int*型的指针变量 q
q=p;                       //使 int*型的指针变量 q 与 p 指向同一块存储空间
```

在第一种方法中出现的“&”是取址运算符，作用是获取变量 a 的地址。该符号在输入函数 scanf()中也有出现，这是因为，数据只能由实参传递给形参，而不能反向传递，所以只能通过获取变量的地址来对该变量进行操作。

也可以在定义的同时为指针变量赋值，其形式如下所示：

```
int a=10;                //定义一个 int 型的变量 a 并初始化为 10
int *p=&a;               //定义一个 int*型的变量 p 并初始化为变量 a 的地址
```

2. 指针变量的引用

所谓指针变量的引用，就是根据指针变量中存放的地址，访问该地址对应的变量。访问指针变量中指针所指变量的方式非常简单，只需在指针变量名之前添加一个取值 “*” 运算符即可，其语法格式如下所示：

```
*指针变量名
```

具体示例如下：

```
int a=10;
int* p=&a;
printf("%d\n",*p);       //输出指针变量指向的地址中存储的数据
```

该示例中*p 表示指针变量指向的地址中存储的数据，当前指针 p 指向 int 型变量 a 的地址，所以*p 表示的即为 a 的值。这个过程也是对变量 a 的访问，这种通过变量的地址来访问变量的方法称为间接访问。

与之相对还有直接访问，直接访问是直接对变量访问，如：

```
printf("%d\n",a);
```

就是直接对 int 型变量 a 进行访问。

虽然间接访问较直接访问麻烦，但是在如下场合，只能使用间接访问：

（1）用户申请一块内存空间时。因为该内存空间没有对应的变量名，所以只能通过首地址对其进行操作（关于内存空间的申请和回收等操作将在案例 3 中讲解）；

（2）通过被调函数改变主调函数变量的值时。由于值只能由实参向形参单向传递，所以被调函数无法通过改变形参的值去改变主调函数中变量的值，只能通过间接访问指针指向的内存空间来改变主调函数中变量的值。scanf()函数就是一个很好的例子。

案例实现

1. 案例设计

假设将案例描述中的宝宝和爸爸视为变量，书房和卧室视为存储空间，那么在实现时，卧室和书房都应设置为指针，卧室和书房指向宝宝和爸爸的地址。

2. 完整代码

```
3.  #include <stdio.h>
4.
5.  int main()
6.  {
7.   int father = 1;                          //定义爸爸变量
8.       int baby = 2;                        //定义宝宝变量
9.   int *sturoom,*bedroom;                  //定义指针
10.  int *dd = &father;                      //获取爸爸的地址
```

```
11.  sturoom = dd;                          //使用爸爸变量地址为指针 sturoom 赋值
12.  bedroom = &baby;                       //取宝宝变量的地址赋给卧室指针
13.  //输出地址
14.  printf("爸爸所在的地址为：%x\n", sturoom);
15.  printf("宝宝所在的地址为：%x\n", bedroom);
16.  //输出变量存储的数值
17.  printf("爸爸：%d\n", *sturoom);         //通过指针间接访问
18.  printf("宝宝：%d\n", *bedroom);
19.  return 0;
20. }
```

运行结果如图 6-3 所示。

图6-3 【案例 1】程序运行结果

多学一招：空指针、无类型指针、野指针

空指针：空指针即没有指向任一存储单元的指针。有时可能需要用到指针，但是不确定指针在何时何处使用，因此先使定义好的指针指向空。具体示例如下：

```
int* p1=0;                          //0 是唯一不必转换就可以赋值给指针的数据
int* p2=NULL;                       //NULL 是一个宏定义，其作用与 0 相同
                                    //在 ASCII 码中，编号为 0 的字符就是空
```

一般在编程时，先将指针初始化为空，再对其进行赋值操作：

```
int x=10;
int* p=NULL;                        //使指针指向空
P=&x;
```

无类型指针：之前讲述的指针都有确定的类型，如 int*型、char*型等，但有时指针无法被给出明确的类型定义，此时就用到了无类型指针。无类型指针使用 void*修饰，这种指针指向一块内存，但因其类型不定，程序无法根据这种定义确定为该指针指向的变量分配多少存储空间。所以若要使用该指针为其它基类指针赋值，必须先转换成其它类型的指针。使用该指针接收其它指针时不需要强转。具体示例如下：

```
void *p=NULL,*q;                    //定义一个无类型的指针变量
int* m=(int*)p;                     //将无类型的指针变量 p 强制转换为 int*型再赋值
int a=10;
q=&a;                               //接收其它类型的指针时不必强转
```

野指针：指向不可用区域的指针。对野指针进行操作可能会发生不可预知的错误。野指针的形成原因有以下两种：

1、指针变量没有被初始化。定义的指针变量若没有被初始化，则可能指向系统中任意一块存储空间，若指向的存储空间正在使用，当发生调用并执行某种操作时，就可能造成系统崩溃。所以在定义指针时应使其指向合法空间。

2、若两个指针指向同一块存储空间，指针与内存使用完毕之后，调用相应函数释放了一个指针与其指向的内存，却未改变另一个指针的指向，将其置空。此时未被释放的指针就变为野指针。

在编程时，可以通过“if(p==NULL){}”来判断指针是否指向空，但是无法检测该指针是否为野指针，所以要避免野指针的出现。

【案例 2】猜宝游戏

案例描述

学生时代的生活虽然单一，但也有许多小游戏贯穿其中，给平淡的校园生活带来一丝欢乐，猜硬币就是这些游戏之一。某个课间，甲和乙一起玩猜硬币的游戏：初始时，甲的左手握着一枚硬币，游戏开始后，甲进行有限次或真或假的交换，最后由乙来猜测这两只手中是否有硬币。

本案例要求编写程序，实现游戏过程。

案例分析

由于该案例比较主观，并且甲的手法和乙的眼力都能影响游戏的结果，因此本案例的目的在于模拟游戏过程。

因为游戏要执行有限次，所以需要首先确定交换进行的次数，通过循环执行每次交换；又因为每次交换是真是假并不确定，所以至少需要实现两个交换函数，一个函数真正地实现两个手中硬币的交换，另一个只需表面完成交换。而每次是否真正地交换硬币也是随机的，因此使用随机数发生器来决定每次选择执行的函数。

本案例中将涉及到指针的相关使用方式，下面先来学习这些知识。

必备知识

1．指针作为函数参数

在 C 语言中，实参和形参之间的数据传递是单向的值传递，即只能由实参传递给形参，而不能由形参传递给实参。这与 C 语言中内存的分配方式有关。当发生函数调用时，系统会使用形参对应的实参为形参赋值，此时的形参以及该函数中的变量都存放在函数调用过程中系统在栈区开辟的空间里，栈区随着函数的调用而被分配，随着函数的结束而被释放，在此过程中，栈区对主调函数不可见，因此主调函数并不能读取栈中形参的数据。若要将栈中的数据传递给朱调函数，只能用关键字“return”来实现。

但并非所有从主调函数传入被调函数的数据都是不需要改变的。在第四章学习函数时曾讲到过返回值，利用返回值可以将在被调函数中修改的数据返回给主调函数，但是 C 语言中返回值只能返回一个数据，往往不能达到要求；函数中也曾学到过全局变量，但是这种方式违背模块化程序设计的原则，与函数的思想背道而驰。

本节将学习一种新的方法，即使用指针变量作为函数的形参，通过传递地址的方式，使形参和实参都指向主调函数中数据所在地址，从而使被调函数可以对主调函数中的数据进行操作。

2．指针的交换

根据指针可以获得变量的地址，也可以得到变量的信息，所以指针交换包含两个方面，一是指针指向交换，二是指针所指地址中存储数据的改变。

（1）指针指向交换

若要交换指针的指向，首先需要申请一个指针变量，记录其中一个指针原来的指向，再使该指针指向另外一个指针，使另外一个指针指向该指针原来的指向。假设 p 和 q 都是 int* 型的指针，则其指向交换示意图如图 6-4 所示。

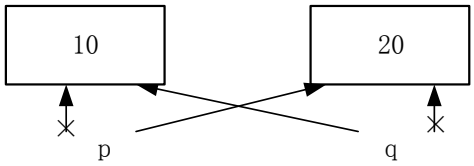


图6-4　指针交换指向

具体的实现方法如下：

```
int *tmp=NULL;                    //创建辅助变量指针
tmp=p;                            //使用辅助指针记录指针 p 的指向
p=q;                              //使指针 p 记录指针 q 的指向
q=tmp;                            //使指针 q 指向 p 原来指向的地址
```

（2）数据的交换

若要交换指针指向的空间中的数据，首先需要获取数据，获取数据的方法在案例一中已经学习，即使用“*”运算符取值。假设 p 和 q 都是 int*型的指针，则数据交换示意图如图 6-5 所示。

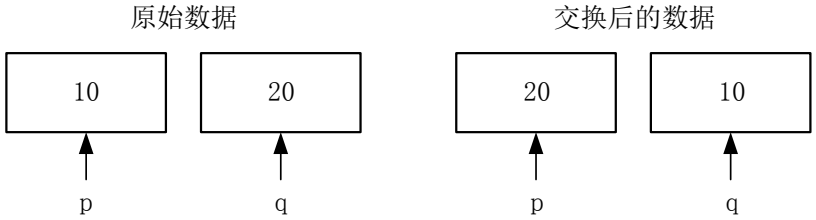


图6-5　数据交换

具体的实现方法如下：

```
int tmp=0;                        //创建辅助变量
tmp=*p;                           //使用辅助变量记录指针 p 指向地址中的
数据
*p=*q;                            //将 q 指向地址中的数据放到 p 所指地址
中
*q=tmp;                           //将 p 中原来的数据放到 q 所指地址中
```

案例实现

1．案例设计

（1）使用基类型的变量作为形参，构造交换函数；
（2）使用指针变量作为形参，在函数体中交换指针的指向；
（3）使用指针变量作为形参，在函数体中交换指针变量所指内存中存储的数据；
（4）使用随机数生成器确定交换发生的次数，选择每轮要执行的交换方法；
（5）使用 while 循环语句控制交换进行的轮数；
（6）使用 switch 语句根据产生的随机数选择本轮执行的交换方法。

2．完整代码

```
#include <stdio.h>
#include <stdlib.h>
//函数声明
void exc1(int l, int r);
void exc2(int* l, int* r);
void exc3(int* l, int* r);

//游戏模拟
//使用随机函数获取交换的次数，和每次交换所选择的函数
int main()
{
    int a = 0, i = 0, j;
    int l = 1, r = 0;
    srand((unsigned int)time(NULL));
    i = 5 + (int)(rand() % 5);                    //随机设置交换次数
    j = i;
    printf("a: %d,i: %d\n",a,i);
    printf("原始状态: \n");
    printf("l=%d,r=%d\n\n", l, r);
    while (i>0)
    {
        i--;
        a = 1 + (int)(rand() % 3);
        switch (a)
        {
        case 1:
            exc1(l, r);
            printf("exc1-第%d 次交换后的状态\n", j - i);
            printf("l=%d,r=%d\n\n", l, r);
            break;
```

```
31          case 2:
32              exc2(&l, &r);
33              printf("exc2-第%d次交换后的状态\n", j - i);
34              printf("l=%d,r=%d\n\n", l, r);
35              break;
36          case 3:
37              exc3(&l, &r);
38              printf("exc3-第%d次交换后的状态\n", j - i);
39              printf("l=%d,r=%d\n\n", l, r);
40              break;
41          default:
42              break;
43          }
44      }
45      return 0;
46  }
47  //函数定义
48  void exc1(int l, int r)
49  {
50      int tmp;
51      tmp = l;        //交换形参的值
52      l = r;
53      r = tmp;
54  }
55  void exc2(int* l, int* r)
56  {
57      int* tmp;
58      tmp = l;        //交换形参的值
59      l = r;
60      r = tmp;
61  }
62  void exc3(int* l, int* r)
63  {
64      int tmp;
65      tmp = *l;       //交换形参变量指向内容的值;
66      *l = *r;
67      *r = tmp;
68  }
```

运行结果如图 6-6 所示。

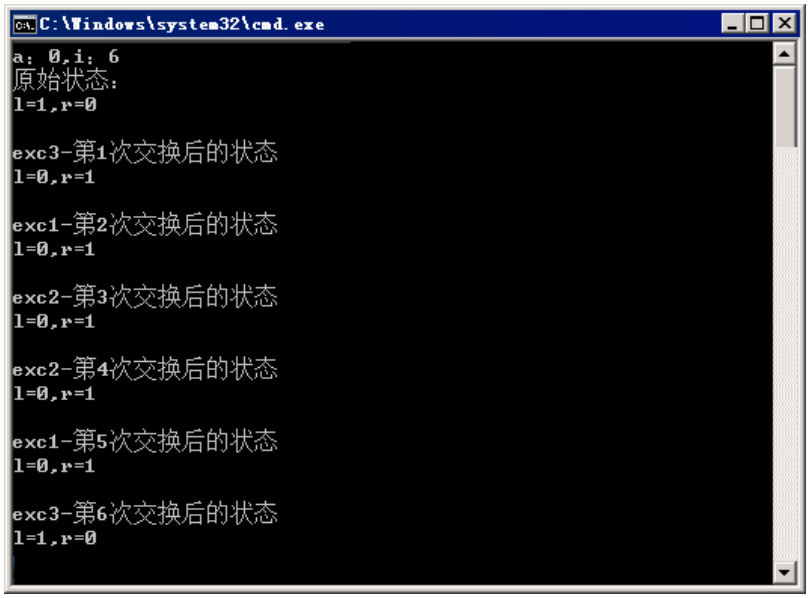


图6-6　【案例 2】运行结果

3．代码详解

代码 4~6 行给出了三个交换函数的声明，代码 10~46 行为主函数，代码 48~68 行为三个交换函数的定义；在主函数中，代码 20~44 行为 while 循环的循环体；代码 24 ~43 行为 switch 语句，根据由 a 记录的 rand()函数产生的随机数选择使用的交换函数。

关于本程序中的三个交换函数，第一个函数传入的是基变量，函数中的数据交换随着函数的结束与栈的回收而失效，并不能对主函数产生影响，所以是一个假交换；第二个函数传入的为指针变量，将两个基变量的地址作为参数传入函数，但是函数中修改的只是这两个形参指针的指向，并未修改原地址中的数据，所以同样是假交换；第三个函数传入指针变量，通过指针变量操作了地址中对应的变量，所以是真交换。

【案例 3】幻方

案例描述

不知大家是否还记得第五章案例 3 中讲解的魔方阵？将从 1 至 n^2 的自然数排列成纵横各有 n 个数的矩阵，使每行、每列、每条主对角线上的 n 个数之和都相等。这样的矩阵就是魔方阵，也称作幻方。本案例要求编写程序，实现奇数阶的幻方。如图 6-7 所示，为一个 3 阶幻方。

8	1	6
3	5	7
4	9	2

图6-7　3 阶幻方

案例分析

观察图 6-7 中的 3 阶幻方，其中的每一行之和分别为：8+1+6=15，1+5+7=15，4+9+2=15；每一列之和分别为：8+3+4=15，1+5+9=15，6+7+2=15；对角线之和分别为：8+5+2=15，6+5+4=15。其行、列、对角线之和全部相等。其和 sum=n×(n²+1)/2=3×(3²+1)/2=15。幻方的构造规则在第五章中已经讲解，这里不再赘述。

在设计案例之前，先来学习案例实现时将会涉及的知识。

必备知识

1．指针和一维数组

一个普通的变量有地址，一个数组包含若干个变量，数组中的每个元素都在内存中占据存储单元，所以每个元素都有各自的地址。指针可以通过变量的地址访问相应的变量，当然也可以根据指针的指向来访问数组中的元素。

以 int 型数组为例，假设有一个 int 型的数组，其定义如下：

```
int a[5]={1,2,3,4,5};
```

若要使用指针指向数组中的元素，则其方法如下：

```
int *p=NULL;                                    //定义一个指针
p=&a[0];                                        //使指针指向数组中的元素 a[0]
```

也可以使指针直接指向数组 a[]。通过对之前章节的学习已经知道，数组名实质上是一个指向数组首地址的指针，也就是指向数组中第一个元素的指针，但这个指针不同于普通的元素指针，它的值不能被修改。所以若要通过指针访问数组中的其它元素，必须先定义一个指向该数组的指针，该指针的定义方式如下：

```
int* p=NULL;                                    //定义一个指针
p=a;                                            //使指针指向数组的首地址
```

实质上本条定义语句与之前的赋值语句“p=&a[0]”等价，都是将数组中首元素的地址赋给指针变量。另外需要注意的是，数组名是一个地址，在为指针赋值时不可再对其进行取址操作。本条赋值语句将数组的数组名赋给了指针 p，此时 p 与数组名等价，所以可以像使用数组名一样，使用下标取值法对数组中的元素进行取值。其表示如下：

```
p[下标]                                         //下标取值法
```

指针的实质就是地址，其实对地址的加减运算并无意义，地址的值也不允许随意修改，但是当指针指向数组元素时，对指针进行加减运算能大大提高指针的效率。

若数组指针与一个整数结合，执行加法操作，例如对以上定义的，指向数组 a[]的指针 p，使 p=p+1，则指针 p 将会指向数组中当前位置的下一个元素，即指向数组 a[]中的元素 a[1]。这是因为针对数组中的元素执行 p+1 操作时并非将地址的值进行简单的加 1，而是根据数组元素的类型，加上一个元素所占的字节数。在本次 p=p+1 时，指针实际上加了 4 个字节（一个 int 型数据所占的字节），若指针 p 存储的地址原本为 0x2016，则运算后的指针存储的地址变为 0x2020。其图形表示如图 6-8 所示。

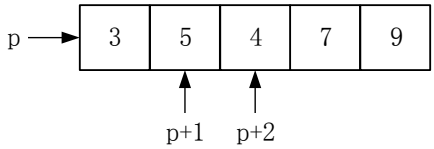


图6-8　数组元素与指针

同理，若执行 p=p+2，则指针 p 将会指向数组元素 a[2]，其地址由原本的 0x2016 变为 0x2024；

若再执行 p=p-1，指针会向后回退一个存储单元，从指向 a[2]变为指向 a[1]。

在案例 1 中已经学习过如何通过指针间接取值，使用指针间接获取数组元素的方式与之类似，都是使用“*”运算符来取值。假设此时指针 p 指向数组元素 a[0]，若要使用指针获取数组元素 a[2]的值，可以使用如下两种方式。

（1）移动指针，使指针指向 a[2]，获取指针指向元素的值：

```
p=p+2;
printf(" %d",*p);
```

（2）不改变指针指向，通过数组元素指针间的关系运算指针并取值：

```
printf(" %d",*(p+2));
```

这是使用指针获取数组中元素的另一种方法。

假设要获取数组 a 中的元素 a[3]，则使用下标法和指针法取值的方式分别如下：

```
p[3]                                    //下标取值法
*(p+3)                                  //指针取值法
```

当指针指向数组元素时，还可以进行减法操作。此时指针类型相同，因此相减之后的结果必为数组元素类型字节长度的倍数，根据这个数值，可以计算出两个元素之间相隔元素的个数。

比如此时指针 p1 指向数组元素 a[1]，指针 p2 指向数组元素 a[3]，则执行以下操作：

```
(p2-p1)/sizeof(int)
```

得到的结果为 2，表示 p1 和 p2 所指的元素之间相隔两个元素，如此一来，不需要具体的知道两个指针所对应的数据，就可以知道它们的相对距离。

需要注意的是，两个指针（地址）相加没有意义。

2．内存分配

在程序执行的过程中，为保证程序能顺利执行，系统会为程序以及程序中的数据分配一定的存储空间。但是有些时候，系统分配的空间无法满足要求，此时需要编程人员手动申请堆上的内存空来存储数据间。

C 语言中申请空间常用的函数其：malloc()函数、calloc()函数和 realloc()函数，这三个函数包含在头文件“stdlib.h”中，都能申请堆上的空间。

（1）malloc()函数

malloc()函数用于申请指定大小的存储空间，其函数原型如下：

```
void* malloc(unsigned int size);
```

在该原型中，参数 size 为所需空间大小。该函数的返回值类型为 void*，使用该函数申请空间时，需要将空间类型强转为目标类型。假设要申请一个大小为 16 字节、用于存储整型数据的空间，则公式如下：

```
int* s=(int*)malloc(16);
```

当为一组变量申请空间时，常用到 sizeof()函数，该函数的作用是求传入参数的字节数。

使用该函数，则可以在已知数据类型和数据数量的前提下方便地传入需要开辟空间的大小。假设为一个包含 8 个 int 型数据的数组申请存储空间，其方法如下所示：

```
int *arr=(int*)malloc(sizeof(int)*8);
```

该语句的作用是，为整型数组 arr 开辟了 8 个 int 类型的存储单元。

（2）calloc()函数

calloc()函数与 malloc()函数基本相同，执行完毕后都会返回一个 void*型的指针，只是在传值的时候需要多传入一个数据。其函数原型如下：

```
void* calloc(unsigned int count,unsigned int size);
```

calloc()函数的作用比 malloc()函数更为全面。经 calloc()函数申请得到的空间会被该函数初始化之后才返回，其数据全为 0，而 malloc()函数申请的空间未被初始化，存储单元中存储的数据不可知。另外 calloc()在申请数组空间时非常方便，它可以将 size 设置为数组元素的空间大小，将 count 设置为数组的容量。

（3）realloc()函数

realloc()函数的函数原型如下：

```
void* realloc(void* memory,unsigned int newSize);
```

realloc()函数的参数列表包含两个参数，参数 memory 为指向堆空间的指针，参数 newSize 为新内存空间的大小。realloc()函数的实质是使指针 memory 指向存储空间的大小变为 newSize。如果 memory 原本指向的空间大小小于 newSize，则系统将试图合并 memory 与其后的空间，若能满足需求，则指针指向不变；如果不能满足，则系统重新为 memory 分配一块大小为 newSize 的空间。如果 memory 原本指向的空间大小大于或等于 newSize，将会造成数据丢失。

3．内存回收

需要注意的是，使用 malloc()函数、calloc()函数、realloc()函数申请到的空间都为堆空间，程序结束之后，系统不会将其自动释放，需要由程序员自主管理。

C 语言提供了 free()函数来释放由以上几种方式申请的内存，free()函数的使用方法如下：

```
int* p=(int*)malloc(sizeof(int)*n);                    //申请
free(p);                                               //释放
```

一个程序结束时，必须保证从堆区申请的所有空间都已被安全释放，否则会导致内存泄露。内存泄漏也称为“内存渗漏”，使用动态存储分配函数开辟的空间，在使用完毕后若未释放，将会一直占据该存储单元，直到程序结束。

若发生内存泄漏，则某个进程可能会逐渐占用系统可提供给进程的存储空间，该进程运行时间越长，占用的存储空间就越多，直到最后耗尽全部存储空间，导致系统崩溃。

内存泄漏是从操作系统的角度考虑的，这里的存储空间并非指物理内存，而是指虚拟内存大小，这个虚拟内存大小取决于磁盘交换区设定的大小。由程序申请的一块内存，如果没有指针指向它，那么就说这块内存泄漏了。

除了这几个函数，还有其它常用的内存操作函数，具体请参见附录Ⅳ。

案例实现

1．案例设计

（1）矩阵的行数、列数、矩阵中元素的数量都由 n 确定，在程序中设置 scanf()函数，由用户手动控制幻方的规模。因为本案例针对奇数阶的幻方，因此如果输入的数据不是奇数，则使用 goto 语句回到输入函数之前；

（2）因为本案例中元素的数量不确定，因此使用 malloc()函数动态申请存储空间；

（3）幻方中的数据按行序优先存储在 malloc()函数开辟的空间中，在输出时，每输出 n 个数据，进行一次换行；

（4）将所有的操作封装在一个函数中，在主函数中调用该函数。在函数结束之前，使用 free()函数释放函数中申请的堆空间。

2．完整代码

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void array();
5  int main ()
6  {
7      array();                              //调用 array 函数
8      return 0;
9  }
10 void array()
11 {
12     int n, i, j, idx, num, MAX;
13     int* M;                               //定义一个一维数组指针，按行优先存储矩阵
中的元素
14     printf("请输入 n: ");
15     input:
16     scanf("%d", &n);
17     if (n % 2 == 0)                       //n 是偶数，则重新输入
18     {
19         printf("n 不为奇数，请重新输入: ");
20         goto input;
21     }
22     MAX = n*n;                            //MAX 为幻方中的最大值，也是元素个数
23     M = (int*)malloc(sizeof(int)*MAX); //分配存储空间
24     M[n / 2] = 1;                         //获取数值 1 的列标
25     i = 0;
26     j = n / 2;
27     //从 2 开始确定每个数的存放位置
```

```
28      for (num = 2; num <= MAX; num++)
29      {
30          i = i - 1;
31          j = j + 1;
32          if ((num - 1) % n == 0)          //当前数是 n 的倍数
33          {
34              i = i + 2;
35              j = j - 1;
36          }
37          if (i < 0)                       //当前数在第 0 行
38              i = n - 1;
39          if (j>n - 1)                     //当前数在最后一列，即 n-1 列
40              j = 0;
41          idx = i*n + j;                   //根据二维数组下标与元素的对应关系
42                                           //找到当前数在数组中的存放位置
43          M[idx] = num;
44      }
45      //打印生成的幻方
46      printf("生成的%d 阶幻方：", n);
47      idx = 0;
48      for (i = 0; i < n; i++)
49      {
50          printf("\n");                    //每 n 个数据为一行
51          for (j = 0; j < n; j++)
52          {
53              printf("%3d", M[idx]);
54              idx++;
55          }
56      }
57      printf("\n");
58      free(M);                             //手动释放堆空间
59  }
```

运行结果如图 6-9 所示。

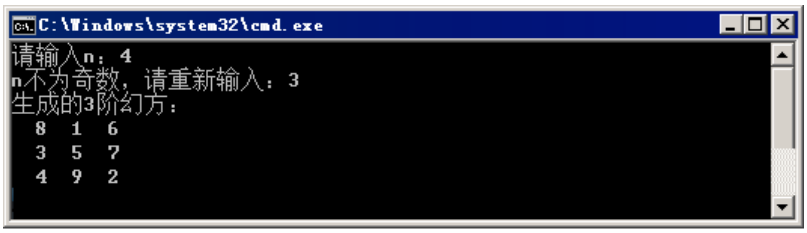


图6-9　【案例 3】运行结果

3．代码详解

本案例的代码实现中包含一个主函数 main()和一个功能函数 array()，其中 main()函数主要作为程序的入口，array()函数实现程序的功能。

在 array()函数中，首先定义了一个一维数组指针，用于存储矩阵中的元素，之后使用 scanf()函数获取矩阵的大小。因为本案例要求生成奇数阶的方阵，所以需要对输入的数据进行判断。代码 17~21 行为判断输入的数据是否为奇数，若不是，则使用 goto 语句回到 scanf()语句之前，重新输入。

代码 23 行为之前定义的一维数组分配存储空间；代码 28~44 行确定数组中每个数据在方阵中的位置；代码 48~56 行输出之前生成的方阵。

代码 58 行为手动释放之前为一维数组申请的堆空间。

根据图 6-9 中的运行结果可以看出，程序实现了案例 3 要求的功能。

【案例 4】快速排序

案例描述

快速排序由 C•A•R•Hoare 在 1962 年提出，是对冒泡排序的改进。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的都小；然后再按此方法，对这两部分数据分别进行快速排序，整个排序过程可以递归进行，直到整个数据变成有序序列为止。相比于冒泡排序，快速排序在时间性能上有大大的提升。本案例要求使用指针实现快速排序算法，并将排序结果逐个输出。

案例分析

设要排序的数组是 S[0]…S[N-1]，首先任意选取一个数据（通常选用数组的第一个数）作为关键数据，然后将所有比键值小的数都放到键值之前，所有比键值大的数都放到键值之后，这个过程称为一趟快速排序。一趟快速排序的算法步骤如下：

（1）设置两个变量 low、high，排序开始的时候：low=0，high=N-1；

（2）以第一个数组元素作为关键数据，赋值给 key，即 key=S[0]；

（3）从 high 开始向前搜索，即从后向前搜索(high--)，找到第一个小于 key 的值 S[high]，将 S[high]和 S[low]互换；

（4）从 low 开始向后搜索，即从前向后搜索（low++），找到第一个大于 key 的值 S[low]，将 S[low]和 S[high]互换；

（5）重复步骤（3）、（4），直到 low>=high 为止；

需要特别注意的是，若在第（3）、（4）步中，没找到符合条件的值，即（3）中 S[high]不小于 key，（4）中 S[low]不大于 key 时，改变 high、low 的值，使得 high=high-1，low=low+1，直至找到为止。找到符合条件的值，进行交换时，low， high 指针位置不变。

案例实现

1．案例设计

根据快速排序的思想可知，快速排序是一个递归的算法，在实现的过程中将会发生对其自身的调用。递归是将一个大型复杂的问题层层转化为一个与原问题相似的，规模较小的问题来求解，在快速排序中，小规模的问题包括两个操作，一是分割数组，二是进行比较。在

最小规模的问题中，分割后的数组只有两个数据，在本层排序中只需对这两个数据进行比较排序即可。

2．完整代码

```
#include <stdio.h>

//快速排序
void QuickSort(int *arr, int left, int right)
{
    //如果数组左边的索引大于或等于右边索引，说明该序列整理完毕
    if (left >= right)
        return;
    int i = left;
    int j = right;
    int key = *(arr + i);           //使用 key 来保存作为键值的数据
    //本轮排序开始，当 i=j 时本轮排序结束，将值赋给 arr[i]
    while (i < j)
    {
        while ((i < j) && (key <= arr[j]))
            j--;                    //不符合条件，继续向前寻找
        *(arr + i) = *(arr + j);
        //从前往后找一个大于当前键值的数据
        while ((i < j) && (key >= arr[i]))
            i++;                    //不符合条件，继续向后寻找
        //直到 i<j 不成立时 while 循环结束，进行赋值
        *(arr + j) = *(arr + i);
    }
    *(arr + i) = key;
    QuickSort(arr, left, i - 1);
    QuickSort(arr, i + 1, right);
}
//输出数组
void print(int *arr, int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", *(arr + i));
}
int main()
{
    int arr[10] = { 3, 5, 6, 7, 2, 8, 9, 1, 0, 4 };
    printf("原数组：\n");
    print(arr, 10);
    QuickSort(arr, 0, 9);           //排序算法
    printf("\n 排序后的数组：\n");
```

```
41      print(arr, 10);                 //输出数组
42      return 0;
43  }
```

运行结果如图 6-10 所示。

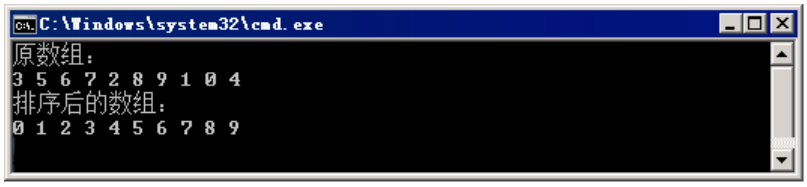


图6-10　【案例 6】运行结果

3．代码详解

代码分为三部分：主函数，快速排序算法函数和输出函数。

代码 4~27 行为排序函数的实现部分，其中参数列表包括一个数组指针和两个下标索引。在算法实现中，首先判断参数列表的两个索引情况，若数组左边的索引大于或等于右边索引，说明数组序列调整完毕，否则使用变量 i、j 分别记录索引，使用变量 key 记录本轮排序的键值。代码 13~23 行为比较赋值部分，其主要功能是调整数组中的数据，使其分别有序。代码 24 行将记录的键值放到合适的位置，然后对被键值分割出的两部分分别进行快速排序。

代码 29~33 行为输出函数的实现，其作用是根据数组指针，逐个输出数组中的数据。

代码 34~43 行为主函数部分，在主函数中首先定义了一个整型数据，之后调用输出函数将该数组输出，然后调用排序函数对其进行排序，最后再将排序后的数据输出。

【案例 5】数据表

案例描述

工作生活中常常需要处理一些数据，小到个人的日常开支，大到公司的整体运营，为了使数据处理的效率更高，操作更加方便，常常使用各式各样的数据表来存储这些数据。例如使用一张表格记录全班学生成绩，针对该表格，可以执行基于行的操作，求出某个学生的总成绩，也可以执行基于列的操作，求得某个科目的成绩，进而得出本班学生某科目的平均分。

如图 6-11 为一个简单的数据表。编程实现一个数据表，用户可以向系统中动态地输入一批正整数，并能完成基于行或列的求和运算。

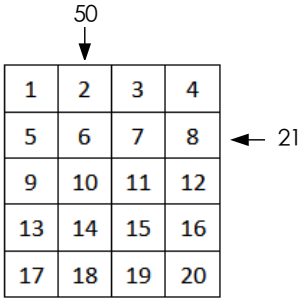


图6-11　数据表

案例分析

图 6-11 中是一张用于存储正整数的数据表，程序应逐行或者逐列地存储表中的每一个数据，并能逐个获取表中的数据，按照行或者列对表中的数据进行运算。该表的形式类似二维数组的逻辑存储结构，所以在案例实现时很容易想到使用二维数组来存储该表，但本章节要讲解的知识都与指针有关，所以本案例的实现也借助指针完成。

本案例要讲解的主要知识有两个：一是函数指针，该知识将在选择求和函数时使用；二是指针与二维数组的联系，本案例中数据的存储基于二维数组，数据的获取利用数组指针。下面分别讲解这两个知识点。

必备知识

1．指针与二维数组

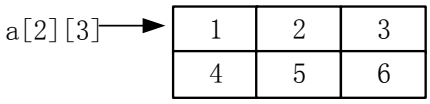
（1）使用指针引用二维数组

在之前的案例中，我们学习了如何用指针引用一维数组。二维数组与多维数组同样有地址，也可以使用指针引用，只是因为其逻辑结构较一维数组复杂，所以操作也较为复杂。程序中使用较多的通常是一维数组与二维数组，这里我们来介绍指针与二维数组的关系。

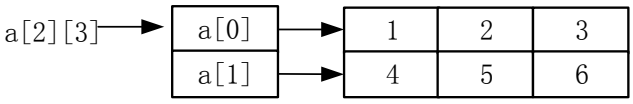
假设要定义一个二行三列的二维数组，其示例如下：

int a[2][3]={{1,2,3},{4,5,6}};

其中 a 是二维数组的数组名，该数组中包含两行数据，分别为{1,2,3}和{4,5,6}。从其形式上可以看出，这两行数据又分别为一个一维数组，所以二维数组又视为数组元素为一维数组的一维数组。其逻辑示意分别如图 6-12 所示。



(a) 二维数组逻辑结构示意



(b) 视为一维数组的二维数组

图6-12 二维数组逻辑示意图

根据图 6-12(b)中的逻辑结构示意图可以看出，与一维数组一样，二维数组的数组指针同样指向数组中第一个元素的地址，只是二维数组中的元素不是单独的数据，而是由多个数据组成的一维数组。

在一维数组中，指向数组的指针每加 1，指针移动步长等于一个数组元素的大小，而在二维数组中，指针每加 1，指针将移动一行，以数组 a 为例，若定义了指向数组的指针 p，则 p 初始时指向数组中的第一行元素，若使 p+1，则 p 将指向数组中的第二行元素。

综上，假设数组中的数据类型为 int，每行有 n 个元素，则数组指针每加 1，指针实际移动的步长为：n*sizeof(int)。

另外，一般用数组名与行号表示一行数据。以上文定义的数组 a[][]为例，a[0]就表示第一行数据，a[1]表示第二行数据。a[0]、a[1]相当于二维数组中一维数组的数组名，指向二维

数组对应行的第一个元素，a[0]=&a[0][0]，a[1]=&a[1][0]。

已经得到二维数组中每一行元素的首地址，那么该如何获取二维数组中单个的元素呢？此时仍将二维数组视为数组元素为一维数组的一维数组，将一个一维数组视为一个元素，再单独获取一维数组中的元素。已知一维数组的首地址为 a[i]，此时的 a[i]相当于一维数组的数组名，类比一维数组中使用指针的基本原则，使 a[i]+j，则可以得到第 i 行中第 j 个元素的地址，对其使用“*”操作符，则*(a[i]+j)表示二维数组中的元素 a[i][j]。若类比取值原则对行地址 a[i]进行转化，则 a[i]可表示为 a+i。

在此需要注意一个问题，即 a+i 与*(a+i)的意义。通过之前一维数组的学习我们都知道，“*”表示取指针指向的地址存储的数据。但在二维数组中，a+i 虽然指向的是该行元素的首地址，但是它代表的是整行数据元素，只是一个地址，并不表示某一元素的值。*(a+i)仍然表示一个地址，与 a[i]等价。*(a+i)+j 表示二维数组元素 a[i][j]的地址，等价于&a[i][j]，也等价于 a[i]+j。

下面给出二维数组中指针与数据的多种表示方法及意义。仍以数组 a[][]为例，具体如表 6-1 所示。

表6-1　二维数组中相关指针与数据的表示形式

表示形式	含义
a	二维数组名，指向一维数组 a[0]，为 0 行元素首地址，也是 a[0][0]的地址
a[i],*(a+i)	一维数组名，表示二维数组第 i 行元素首地址，值为&a[i][0]
*(a+i)+j	二维数组元素地址，二维数组中最小数据单元地址，等价于&a[i][j]
((a+i)+j)	二维数组元素，表示第 i 行第 j 列数据的值，等价于 a[i][j]

（2）作为函数参数的二维数组

一维数组的数组名就是一个指针，若要将一维数组传入函数，只需传入数组名，或指向该数组首地址的指针即可。假设要将一维数组 a[5]传入 func()函数中，函数声明如下：

```
func(int a[]);
```

函数调用时的形式如下：

```
func(a);
```

若在程序中定义一个指向该一维数组的指针：

```
int *p=a;
```

则也可以将该指针传入函数，其形式如下：

```
func(p);
```

若使用一维数组指针传值的方式类比二维数组，很容易将其传入的参数声明为“int **arr”，但这样写是不对的，因为“int **arr”是一个二级指针，其声明的是一个指向整型指针的指针，而非指向整型数组的指针。

若将二维数组传入函数，形式相对略为复杂。一维数组可以不关心数组中数据的个数，但二维数组既有行，又有列，在定义时行值可以缺省，列值不能缺省，所以将二维数组的指针传递到函数中时必须确定数组的列值。定义一个数组指针的形式如下：

```
数据类型 (*数组指针名)[列号];
```

假设现在要将数组 a[4][5]传入函数 func()，则其实现如下：

```
int (*P)[5]=a;
func(p);
```

在这里要注意指针数组与数组指针的区别。指针数组表示数组元素都为指针的一个数组，数组指针表示指向数组的指针，其定义形式的区别在于“*”和“[]”与变量名结合时的优先顺序，切记在定义数组指针时，“()”不可丢失，因为“[]”的优先级高于“*”，所以若没有小括号，该变量就会被编译为指针数组。

2. 函数指针

（1）函数指针的定义

若在程序中定义了一个函数，在编译时，编译器会为函数代码分配一段存储空间，这段空间的起始地址（又称入口地址）称为这个函数的指针。

与普通变量相同，同样可以定义一个指针指向存放函数代码的存储空间的起始地址，这样的指针叫做函数指针。函数指针的定义格式如下：

```
返回值类型 (*变量名)(参数列表)
```

其中返回值类型表示指针指向的函数其返回值类型，“*”表示这是一个指针变量，参数列表表示该指针所指函数的形参列表。

假设定义一个参数列表为两个 int 型变量，返回值类型为 int 的函数指针，则其格式如下：

```
int (*p)(int,int);
```

需要注意的是，因为“*”的优先级较高，所以要将“*变量名”用小括号括起来。

函数指针的类型应与函数返回值指针类型相同，假设有一函数声明为：

```
int func(int a,int b);
```

则可以使用以上定义的函数指针指向该函数，即使用该函数的地址为函数指针赋值，其形式如下：

```
p=func;
```

由此也可以看出，函数名类似于数组名，也是一个指针，指向函数所在存储空间的首地址。

（2）函数指针的应用

函数指针主要有两个用途，一是调用函数，使用函数指针调用对应函数，方法与使用函数名调用函数类似，只是将函数名替换为“*指针名”即可。假设要调用指针 p 指向的函数，其形式如下：

```
(*p)(3,5);
```

另一用途是将函数的地址作为函数参数传入其它函数。将函数的地址传入其它参数，就可以在被调函数中使用实参函数。函数指针作为函数参数的示例如下：

```
void func(int (*p)(int,int),int b,int c);
```

案例实现

1. 案例设计

（1）创建一个二维数组，使用循环语句为其赋值；
（2）在循环结构中使用指针读取数组中的数据并输出；
（3）根据案例要求，在程序使用两个函数分别实现不同方式的求和计算；
（4）同时在主函数中创建函数指针，当用户做出选择之后，根据选择结果调用函数。

2. 完整代码

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

```
2
3  //函数声明
4  void sumbyrow(int(*arr)[4], int row, int *sum);
5  void sumbycol(int(*arr)[4], int col, int *sum);
6
7  int main()
8  {
9      int dataTable[5][4] = { 0 };                //定义数据表
10     int i, j;
11     printf("录入数据中...\n");
12     for (i = 0; i < 5; i++)
13     {
14         for (j = 0; j < 4; j++)
15             dataTable[i][j] = i * 4 + j;
16     }
17     printf("录入完毕\n");
18     int(*p)[4] = dataTable;                     //定义数组指针
19     printf("输出数据: \n");
20     for (i = 0; i < 5; i++)
21     {
22         for (j = 0; j < 4; j++)
23             printf("\t%d", *(*(p + i) + j));
24         printf("\n");
25     }
26     int select, pos, sum;
27     void(*q)();                                 //定义函数指针
28     //求和计算
29     printf("请输入求和方式（行:0/列:1）: ");
30     scanf("%d", &select);
31     printf("选择行/列: ");
32     scanf("%d", &pos);
33     if (select == 0)
34     {
35         printf("按行求和，第%d 行数据",pos);
36         q = sumbyrow;
37     }
38     else if (select == 1)
39     {
40         printf("按列求和，第%d 列数据", pos);
41         q = sumbycol;
42     }
43     (*q)(dataTable, pos, &sum);
44     printf("求和结果为:%d\n", sum);
45     return 0;
46 }
```

```
47 //按行求和
48 void sumbyrow(int (*arr)[4], int row, int *sum)
49 {
50     int i = 0;
51     *sum = 0;
52     for (i = 0; i < 4; i++)
53         *sum += *(*(arr + row-1) + i);
54 }
55 //按列求和
56 void sumbycol(int(*arr)[4], int col, int *sum)
57 {
58     int i = 0;
59     *sum = 0;
60     for (i = 0; i < 5; i++)
61         *sum += *(*(arr + i) + col-1);
62 }
```

运行结果如图 6-13 所示。

图6-13 【案例 5】运行结果

3．代码详解

代码 4 行与代码 5 行为两个求和函数的声明。

代码 9 行定义了一个二维数组 dataTable，用于存储数据表；代码 12~16 行用于初始化数组 dataTable；

代码 18 行定义了一个数组指针，指向二维数组 dataTable；代码 20~25 行使用指针获取数组中的数据并输出；

代码 26 行之后为求和部分，代码 27 行定义了一个函数指针；代码 30 行和代码 32 行分别用于输入求和操作的参数；代码 33~42 行用于为函数指针赋值，选择将要执行的函数；

代码 43 行利用函数指针对函数进行调用；代码 44 行输出求和结果。

代码 48~54 行为按行求和的的函数，其返回值类型为 void，参数列表为：二维数组指针、行值和用于记录和的变量的指针。在函数实现的过程中，根据二维数组的逻辑结构，逐个相加，并在函数内部根据传入的指针直接修改变量 sum 的值。

代码 56~62 行为按列求和的函数，其原理与实现步骤与按行求和基本相同，此处不再赘述。

【案例 6】点名册

案例描述

在大学的课堂上，本节课坐在你旁边的可能是位女同学，下节课坐在你旁边的可能是一位男同学；这一节课你可能坐在教室的前三排，下次再来这个教室上课，若来的晚，可能就坐在了教室的最后一排。由于大学的课堂每个人的座位不确定，授课的老师很难将学生的姓名与学生本人对应起来，所以大学往往采取课堂点名的制度来确定本节课上课的学生，此时就需要使用到点名册。

案例要求编程实现一份基于指针的点名册，记录学生的姓名，并能实现学生姓名的输出；点名册中的学生姓名由多个字符组成，点名册中包含不止一名学生。

案例分析

若将每个学生的姓名视为一个字符数组，则点名册中的内容可以视为多个字符数组的集合。如若每个学生姓名所占用的存储空间都相同，那么点名册可以视为一个二维数组，但实际上，学生姓名字节数可能各不相同。所以，需要考虑的问题有两个：

（1）如何使用不同长度的字符数组存储学生的姓名；

（2）如何将多个存储学生姓名且长度不同的字符数组联系起来，使之成为一个整体。

考虑到学生姓名逐条存储，类似于二维数组的存储形式，但二维数组中的每行和每列的字节数相同，若使用二维数组存储，必然会造成空间的浪费。那么该如何解决这个问题呢？在解决问题之前，我们先来学习一些新知识。

必备知识

1．通过指针引用字符串

对于下面这条语句：

```
printf(" %s", "hello world! ");
```

相信大家都不陌生。此条语句的功能是格式化地将字符串“hello world!”直接输出，这是 C 语言中字符串最常见的使用方式。

字符串由若干个字符组成，字符型变量作为 C 语言中一种基础的变量类型，与其它变量一样，都会占用存储空间。我们已经知道指针的本质就是地址，既然字符串中的字符占用存储空间，那么显然它也可以通过指针进行操作。

在 C 语言中，字符串一般存放在字符数组中。对字符串进行操作有两种方式：

（1）使用数组名加下标的方式获取字符串中的某个字符；使用数组名与格式控制符“%s”输出整个字符串。具体示例如下：

```
char s[]="this is a string.";
printf("%c\n",s[3]);                    //输出字符数组 s 下标为 3 的位置上存储的字符
printf("%s\n",s);                       //通过数组名或者字符串名输出
```

此段代码中的 printf()对应的输出结果如下：

```
s
this is a string.
```

（2）声明一个字符型的指针，使该指针指向一个字符串常量，通过该指针引用字符串常量。具体示例如下：

```
char* s="hello world!";
printf("%c\n",s[3]);                    //通过下标取值法获取字符串中的第 4 个字符并输出
printf("%c\n",*(s+1));                  //通过指针取值法获取字符串中的第 2 个字符并输出
printf("%s\n",s);                       //通过首指针获取字符串并输出
```

此段代码中的 printf()语句对应的输出结果如下：

```
l
e
hello world
```

需要注意的是，在将指针指向字符串常量时，指针接收的是字符串中第一个字符的地址，而非整个字符串变量。另外虽然字符型的指针和字符数组名都能表示一个字符串，但是它们之间存在细微的差别：字符串的末尾会有一个隐式的结束标志'\0'，而数组中不会存储这个结束标志，只会显式地存储字符串中的可见字符。

2．指针数组

之前使用到的数组有整型数组、字符型数组或由其它基本数据类型的变量组成的数组。指针变量也是 C 语言中的一种变量，同样的，指针变量也可以构成数组。若一个数组中的所有元素都是指针类型，那么这个数组是指针数组，该数组中的每一个元素都存放一个地址。

定义一维指针数组的语法格式如下：

```
类型名* 数组名[数组长度];
```

根据上述语法格式，假设要定义一个包含 5 个整型指针的指针数组，其实现如下：

```
int* p[5];
```

此条语句定义了一个长度为 5 的指针数组 p，数组中元素的数据类型都是 int*。由于“[]”的优先级比“*”高，所以数组名 p 先和“[]”结合，表示这是一个长度为 5 的数组，再与“*”结合，表示该数组中元素的数据类型都是 int*型，每个元素都指向一个整型变量。

指针数组是一个数组，那么指针数组的数组名是一个地址，它指向该数组中的第一个元素，也就是该数组中存储的第一个地址。指针数组名的实质就是一个指向数组的二级指针。一个单纯的地址没有意义，地址应作为变量的地址存在，所以指针数组中存储的指针应该指向实际的变量。假设现在使用一个字符型的指针数组 a，依次存储如下的多个字符串：

```
"this is a string"
"hello world"
"I love China"
```

则该指针数组的定义如下：

```
char* a[3]={ "this is a string", "hello world", "I love China"};
```

根据以上分析可知，数组名指向数组元素，数组元素指向变量，数组名是一个指向指针的指针。数组名、数组元素与数组元素指针指向的数据其逻辑关系如图 6-14 所示。

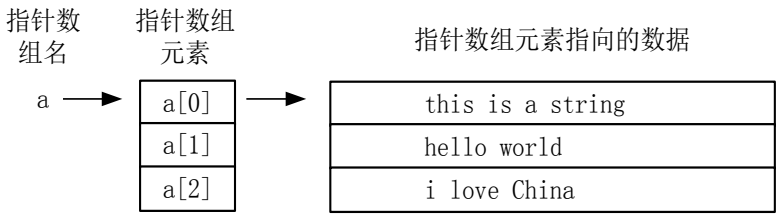


图6-14　指针数组逻辑示意图

图 6-14 中，指针数组名 a 代表的指针指向指针数组中第一个元素 a[0]所在的地址，a+1 即为第二个元素 a[1]所在的地址，依次类推，a+2 为第三个元素 a[3]所在地址。

3．二级指针

一级指针是指向变量的指针，根据该指针找到的数据为普通变量；二级指针是指向指针的的指针，根据该指针可找到指向变量的指针。根据二级指针中存放的数据，二级指针可分为指向指针变量的指针，和指向指针数组的指针。

（1）指向指针变量的指针

定义一个指向指针变量的指针，其格式如下：

```
变量类型 **变量名;
```

假设现有如下定义：

```
int a=10;                    //整型变量
int *p=&a;                   //一级指针 p，指向整型变量 a
int **q=p;                   //二级指针 q，指向一级指针 p
```

则指针 q 是一个二级指针，其中存储一级指针 p，也就是整型变量 a 的地址。逻辑关系如图 6-15 所示。

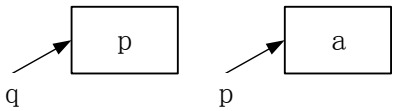


图6-15　指向指针变量的指针

根据运算符的结合性可知，“*”运算符是从右向左结合，所以**p 相当于*(*P)，其中*p 相当于一个一级指针变量，若将定义中最左边的“*”运算符与变量类型结合，则以上语句可视为如下形式：

```
int* (*q) = p;
```

此条语句中，*q 表示一个指针变量，而其变量类型 int*表示该变量指向的仍为一个 int*型的数据，所以这条语句定义了一个指向指针变量的指针。

（2）指向指针数组的指针

假设要定义一个指针 p，使其指向指针数组 a[]，则其定义语句如下：

```
char *a[3]={0};
char **p=a;
```

该语句中定义的 p 是指向指针型数据的指针变量，初始时指向指针数组 a 的首元素 a[0]，a[0]为一个指针型的元素，指向一个 char 型数组的首元素，而指针 p 初始时的值为该元素的地址。

当然若再次定义指向该指针的指针，会得到三级指针。因为指针本来就是 C 语言中较为难理解的部分，若能掌握指针的精髓，将其充分利用，自然能够提高程序的效率，大大的优化代码，但是指针功能太过强大，若是因指针使用引发错误，很难查找与补救，所以程序

中使用较多的一般为一级指针，二级指针使用的频率要远远低于一级指针，再多重的指针使用的频率更是低，这里就不再讲解。

案例实现

1．案例设计

点名册中的每个学生姓名都可定义为一个字符数组，为了能统一操作点名册中的学生姓名，应使用指针数组，使数组中的每个指针都指向一个学生姓名。同时可以定义一个二级指针，使该指针指向指针数组，使用二级指针读取点名册中的学生姓名。

2．完整代码

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

int main()
{
    char buf[1024];                          //定义缓冲数组
    char * strArray[1024];                   //定义指针数组
    char ** pArray;                          //定义二级指针
    int i, arrayLen = 0;
    printf("请输入学生姓名，以文字“end”结束：\n");
    while (1)
    {
        scanf("%s",buf);                     //将输入的学生姓名存入缓冲数组
        if (strcmp(buf, "end") == 0)         //判断输入是否结束
        {
            printf("结束输入。\n");
            break;
        }
        //为指针数组中的指针元素开辟空间（不可忘记'\0'）
        strArray[arrayLen] = (char *)malloc(strlen(buf) + 1);
        //将缓冲数组的字符串赋值到指针元素指向的空间中
        strcpy(strArray[arrayLen], buf);
        arrayLen++;
    }
    //为二级指针申请 len 个 char*型的存储单元
    pArray = (char **)malloc(sizeof(char *)* arrayLen);
    for (i = 0; i < arrayLen; i++)
    {
        //为二级指针指向的存储单元一一赋值，使其分别指向指针数组中存储的字符串
```

```
31          *(pArray + i) = strArray[i];
32      }
33      printf("您之前输入的文字：\n");
34      for (i = 0; i < arrayLen; i++)          //根据二级指针找到字符串并逐一输出
35      {
36          printf("%s\n",*(pArray + i));
37      }
38      //数组指针空间释放
39      for (i = 0; i < arrayLen; i++)
40      {
41          free(strArray[i]);
42      }
43      //释放二级指针
44      free(pArray);
45      return 0;
46  }
```

程序运行结果如图 6-16 所示。

图6-16 【案例 6】运行结果

3．代码详解

此段代码中用到了两个字符串相关函数：strcmp()函数和 strcpy()函数，这两个函数包含在头文件 string.h 中，其中 strcmp()函数的功能是判断字符串是否相等，若相等则返回 0；strcpy()函数的功能是字符串拷贝，可以将参数列表中第二个字符串的值赋给第一个字符串。

代码 7 行定义了一个字符数组 buf[]，用于接收从输入设备输入的字符串，代码 14 行为接收语句，代码 15~19 行判断输入是否结束；

代码 8 行定义了一个字符型的数组指针 strArray[]，其中的指针元素用于指向从输入设备输入的多个字符串，从输入设备输入的字符串存储在字符数组中，为了有效保存每次的字符串，避免本次输入的字符串被之后输入的字符串覆盖，需要动态地为指针元素开辟存储空间，来存储字符串。代码 21 行动态地申请存储空间，代码 23 行将缓冲数组 buf[]中的字符串存储到申请的堆空间中；

以上的字符串获取、空间申请、字符串赋值都发生在代码 12~25 行的 while 循环中，其间可多次获取字符串，多次开辟不同的堆空间，并为堆空间赋值；

代码 27 行定义了一个二级指针，同时为该二级指针申请了大小为 sizeof(char*)*arrayLen 的存储空间，即申请了 arrayLen 个字符指针型的空间；代码 28~32 行为二级指针中的 char*

空间一一赋值，使其逐个指向字符数组中的字符串；

代码 34~37 行根据二级指针找到字符串并逐一输出；

代码 39~42 行为堆空间的释放。之前的案例中已经讲过，动态申请的空间需要手动释放。分析之前代码可知，程序为指针数组中每个指针指向的字符串开辟了空间，也为二级指针开辟了空间，这些空间都需要逐一释放。所以代码 39~42 行使用一个 for 循环逐个释放指针元素指向的空间，代码 44 行释放二级指针指向的空间。

多学一招：const 修饰符

在程序开发中，有时并不希望使用者修改程序中的某些数据，此时可以使用 const 修饰符对该数据进行修饰，从而提高程序的安全性和可靠性。

const 通常与指针配合使用，根据 const 常量在定义时出现的位置，const 与指针配合有以下三种用法：

1、常量指针

在定义指针时，const 放在数据类型之前，则构成常量指针。常量指针的语法格式如下：

```
const 数据类型* 指针变量名;
```

根据此种语法格式定义的指针，将变为常量指针，也就是说，该指针指向的数据是一个常量，该数据不能被修改。示例如下：

```
int num=10;
const int* p=&num;
```

在以上示例中，p 指向的 int 型变量 10 不能被修改，此时若对 num 重新赋值：

```
num=5;
```

则在调试时会出错，提示表达式中的 num 必须是可修改的左值。

2、指针常量

若 const 放在指针名之前，则该指针与 const 组成一个指针常量，其语法格式如下：

```
数据类型* const 指针变量名;
```

指针常量是一个指针型的常量，表示该指针的指向不能被修改。假设有如下定义：

```
int a = 10;
int b = 5;
int* const p=&a;
```

若此时改变指针 p 的指向，对其进行如下操作：

```
p=&b;
```

则在调试时会出错，提示表达式中的 p 必须是可修改的左值。

3、指向常量的常指针

若 const 既出现在数据类型之前，又出现在指针变量名之前，则此时为一个指向常量的常指针，其语法格式如下：

```
const 数据类型* const 指针变量名;
```

此时不光指针指向的变量不能被修改，指针的指向同样不能被修改。

【案例 7】综合案例——天生棋局

案例描述

中国传统文化源远流长，博大精深，包含着华夏先哲的无穷智慧，也是历朝历代炎黄子孙生活的缩影。围棋作为中华民族流传已久的一种策略性棋牌游戏，蕴含着丰富的汉民族文化内涵，是中国文明与中华文化的体现。本案例要求创建一个棋盘，在棋盘生成的同时初始化棋盘，根据初始化后棋盘中棋子的位置来判断此时的棋局是否是一局好棋。具体要求如下：

（1）棋盘的大小根据用户的指令确定；

（2）棋盘中棋子的数量也由用户设定；

（3）棋子的位置由随机数函数随机确定，若生成的棋盘中有两颗棋子落在同一行或同一列，则判定为“好棋”，否则判定为“不是好棋”。

案例分析

本案例需要根据用户输入的数据分别确定棋盘的大小和棋子的数量，所以棋盘的大小是不确定的。为了避免存储空间的浪费，防止因空间不足造成的数据丢失，本案例可动态地申请堆上的空间，来存储棋盘。

从棋盘的创建到释放，大致包含以下几个步骤：

（1）创建棋盘。棋盘的创建应包含空间的申请，用于存储棋盘中对应的信息；

（2）初始化棋盘。创建好的棋盘是一个空的棋盘，棋盘在显示之前应先被初始化；

（3）输出棋盘。创建并初始化的棋盘包含棋盘的逻辑信息，棋盘的输出应包含棋盘的格局；

（4）销毁棋盘。动态申请的空间需要被释放。

当然在创建棋盘之前，需要获取用户设置的棋盘信息，在初始化棋盘之时，也应根据用户设置的棋子数量来设置棋盘信息。

案例实现

1．案例设计

根据案例分析中的棋局生成步骤设计程序，可将程序代码模块化为 4 个功能函数和一个主函数。

（1）创建棋盘

案例分析中已经提出，棋盘信息存放在动态生成的堆空间中。棋盘由 n×n 个表格组成，其形式类似于矩阵，所以本案例中设计使用二级指针指向棋盘地址。在该函数中应实现棋盘空间的动态申请，并返回一个指向棋盘的二级指针。

（2）初始化棋盘

如图 6-17 由 9×9 个方格组成，它代表一个 10×10 的棋盘。棋子可以落于每个方格的四个顶点，该棋盘最多可容纳 100 个棋子。在创建棋盘时，实质上只开辟了存储空间，空间中

尚未存放棋盘信息，所以在生成棋盘之前需要初始化棋盘信息。

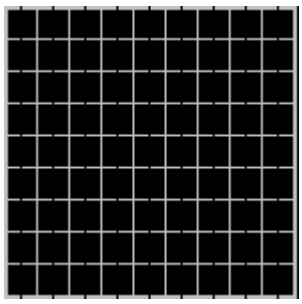


图6-17　10×10 棋盘示例

棋盘信息的初始化可利用指针完成。当棋盘上棋子的数量确定后，在棋盘的范围内使用随机数函数随机确定每个棋子的位置。

（3）输出棋盘

根据由前两个函数确定的棋盘信息搭建棋盘，棋盘的外观可使用制表符搭建。若棋盘对应的位置上有棋子，则将制表符替换为表示棋子的符号。

（4）销毁棋盘

在创建棋盘时申请的堆空间，应在使用完毕之后手动释放。

（5）主函数

主函数中实现棋盘大小和棋子数量的设置，其中应定义一个二级指针，指向创建棋盘的函数返回的棋盘地址，随后依次调用初始化棋盘函数、输出棋盘函数和销毁棋盘的函数。

2．完整代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int ** createBoard(int n)              //创建一个棋盘
{
    int **p = (int**)calloc(sizeof(int*), n);
    int i = 0;
    for (i = 0; i<n; i++)
    {
        p[i] = calloc(sizeof(int), n);
    }
    return p;
}
//初始化棋盘
int initBoard(int **p, int n,int tmp)  //用随机数函数设置棋子位置
{
    int i, j;
    int t = tmp;
```

```
21      while (t>0)
22      {
23          i = rand() % n;
24          j = rand() % n;
25          if (p[i][j] == 1)                //坐标内已有棋子则再次循环
26              continue;
27          else
28          {
29              p[i][j] = 1;
30              t--;
31          }
32      }
33      return 0;
34  }
35  //输出棋盘
36  int printfBoard(int **p, int n)
37  {
38      int i, j;
39      for (i = 0; i<n; i++)                //用行列计数变量检测同一行或列是否多于 2
个棋子
40      {
41          for (j = 0; j<n; j++)
42          {
43              if (p[i][j] == 1)            //输出棋子
44              {
45                  printf("●");
46              }
47              else                         //搭建棋盘
48              {
49                  if (i == 0 && j == 0)
50                      printf("┌");
51                  else if (i == 0 && j == n - 1)
52                      printf("┐");
53                  else if (i == n - 1 && j == 0)
54                      printf("└");
55                  else if (i == n - 1 && j == n - 1)
56                      printf("┘");
57                  else if (j == 0)
58                      printf("├");
59                  else if (i == n - 1)
60                      printf("┴");
61                  else if (j == n - 1)
62                      printf("┤");
63                  else if (i == 0)
64                      printf("┬");
```

```
65                  else
66                      printf("十");
67              }
68          }
69          putchar('\n');
70      }
71      for (i = 0; i<n; i++)               //用行列两个循环判断是否行列上有两个相邻
的棋子
72      {
73          for (j = 0; j<n; j++)
74          {
75              if (p[i][j] == 1)
76              {
77                  if (j>0 && p[i][j - 1] == 1)        //判断同一行有无相邻棋子
78                  {
79                      printf("好棋! \n");
80                      return 0;
81                  }
82                  if (i>0 && p[i - 1][j] == 1)        //判断同一列有无相邻棋子
83                  {
84                      printf("好棋\n");
85                      return 0;
86                  }
87              }
88          }
89      }
90      printf("不是好棋\n");
91      return 0;
92  }
93  //销毁棋盘
94  void freeBoard(int **p, int n)
95  {
96      int i;
97      for (i = 0; i<n; ++i)
98      {
99          free(p[i]);                     //释放一级指针指向的空间
100     }
101     free(p);                            //释放二级指针指向的空间
102 }
103 int main()
104 {
105     srand((unsigned int)time(NULL));
106     int n = 0, tmp = 0;
107     printf("设置棋盘大小:");
108     scanf("%d", &n);                    //输入棋盘行（列）值
```

```
109    int **p = createBoard(n);          //创建棋盘
110    printf("设置棋子数量 :");
111    scanf("%d", &tmp);                 //输入棋盘上的棋子数量
112    initBoard(p, n, tmp);              //初始化棋盘
113    printfBoard(p, n);                 //打印棋盘
114    freeBoard(p, n);                   //释放棋盘
115    return 0;
116}
```

运行结果如图 6-18 与图 6-19 所示。

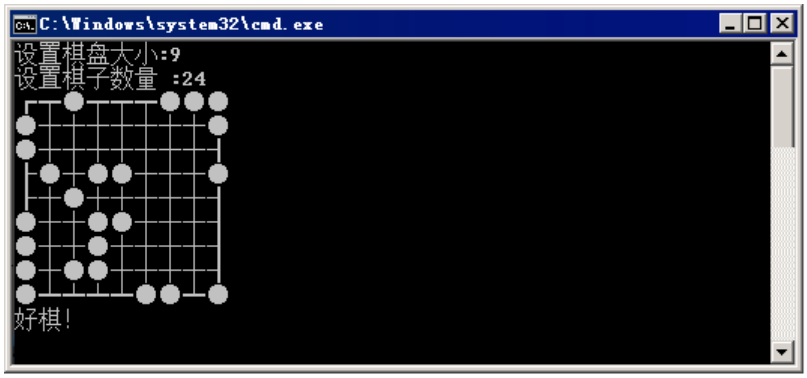


图6-18 【案例 7】运行结果——好棋

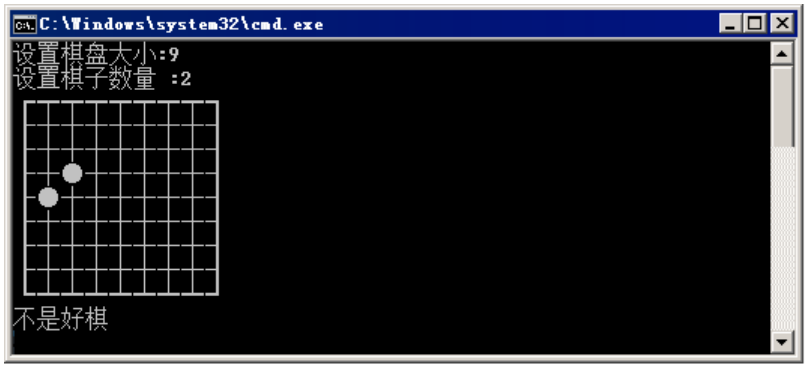


图6-19 【案例 7】运行结果——不是好棋

3・代码详解

本段代码分为 5 个函数，包括 1 个主函数和 4 个功能函数。主函数是程序的入口，功能函数分别实现这些功能：创建自定义大小的棋盘、初始化创建的棋盘、输出棋盘、销毁棋盘。程序开始运行之后，在主程序中依次调用功能函数，即可实现“天生棋局”。

代码 103~116 行为主函数部分，主函数开始之后首先调用 srand()函数设置随机数种子，使函数调用过程中的 rand()函数在每次被调用时可以生成不同的数值；其次调用 scanf()函数输入一个数据，并将该数据传入 createBoard()函数中，用于生成棋盘；然后再次输入一个数据，将该数据传入 initBoard()函数，初始化棋盘；之后将创建并初始化的棋盘使用 printfBoard()函数输出；最后使用 freeBoard()函数销毁棋盘。

代码 6~15 行为 createBoard()函数，该函数接收一个用于控制棋盘大小的整型参数，并在函数体中使用该参数从堆上申请一组空间，将空间首地址存储在一个二级指针中；代码 10~13 行使用一个 for 循环，为这一组堆空间逐一赋值，即赋予每个堆空间一个指向堆空间的地址；代码 14 行将二级指针地址返回。

代码 17~34 行为 initBoard()函数，该函数接收一个二级指针、一个控制棋盘大小的整型变量 n、一个控制棋盘中棋子数量的整型变量 tmp；代码 21~32 行使用 while 循环和 rand()函数，在棋盘中随机设置 tmp 个棋子，将棋盘初始化。

代码 36~92 行为 printfBoard()函数，该函数的功能为打印棋盘、判断棋局情况，并输出判断结果，它接收一个二级指针和一个控制棋盘大小的整型变量 n，在函数体中使用双层 for 循环控制棋盘的打印，使用 if…else…语句控制棋盘的布局；之后再使用双层 for 循环遍历棋盘，判断棋局是否为好棋，并输出判断结果。

代码 94~102 行为 freeBoard()函数，该函数接收一个二级指针和控制棋盘大小的整型数据，代码首先在 for 循环中，使用 free()函数逐个释放二维指针指向的一组一级指针指向的堆空间；其次使用 free()函数释放二级指针指向的一组存放一级指针的堆空间。

本章小结

指针是 C 语言最重要的组成部分，本章通过几个简单案例，讲解了指针、指针变量、函数指针、字符串指针、二级指针、指针数组、数组指针的定义与使用方法，并讲解了如何使用指针引用一维数组与二维数组，以及如何在堆上分配和回收内存。通过本章的学习，读者应能掌握多种指针的定义与使用方法，使用指针优化代码，提高代码的灵活性。