第5章

数　组

“俄罗斯方块”每种形状由4个小方块组成，将其显示在屏幕上，可以通过4条语句实现，每一条语句显示一个小方块。当数据量较小时，这种方式还能勉强应付。但是当数据量较大时，例如“贪吃蛇”游戏中，“贪吃蛇”吃到了很多食物，整个身体变得比较长，由几十个方块组成，就不能采用这种方法。处理大量相关数据时，就需要更好的方式进行存储和处理。在生活中如果数据较多，人们会采用一张表格记录数据，这样处理数据会方便很多。C语言也提供了一种类似表格一样的结构去存储数据，被称为数组，数组能高效、便捷地处理数据。

5.1 一维数组

数组是同类型有序数据的集合，其中的每一个元素都是相同类型。例如数组有10个元素，这10个元素都必须是同一类型。数组是存储在一段连续的空间上，可以通过数组名称加索引（也被称为下标或者偏移量）访问数组中的元素，如图5.1所示。

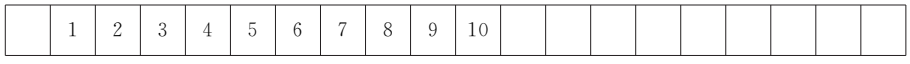

图5.1　数组存储数据示意图

这就像军训时，学生们站成一排，教官可以通过位置指挥学生队列。例如，让第5列的学生出列，对应位置的学生接到指令后就出列。

5.1.1 一维数组的定义

使用数组与使用变量类似，使用之前需要先定义，普通变量能使用的类型，数组也能使用。

定义一维数组的形式：

```
类型说明符 数组名[数组大小];
```

例如：

```
int name[100];
```

表示数组名为 name，此数组有 100 个元素，每个元素都可以存储 int 类型的值。[ ]表明 name 是一个数组，方括号中的数字表明数组中元素的个数。

定义数组时，需要指明元素的类型和元素的数量，并且给数组命名，数组名的命名规则与变量名的规则一致。另外，数组大小是一个常量表达式，C 语言不允许对数组的大小进行动态定义，所以不能是变量。

5.1.2 一维数组的初始化

一维数组的初始化与变量的初始化一样，也可以在定义时为数组中的元素赋初值。

例如：

```
int cols[5] = {6,5,4,3,2 };
```

需要注意的是：

(1) 大括号中的数字个数不能超过数组元素的个数，但是可以少于数组元素的个数。例如：

```
int cols[5] = {6,5,4 };
```

上述数组有 5 个元素，花括号中只提供了 3 个初值，表示只给前面 3 个赋初值，后面 2 个元素都为 0。

(2) 如果数组中元素初始值都为 0，可以写成：

```
int cols[5] = {0};
```

表示数组有 5 个元素，每个元素初始值都是 0。

(3) 如果给数组的全部元素都赋初值，可以省略表示数组大小的常量表达式。例如：

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

系统会根据初值个数来确定元素的个数。如果想定义的数组大小与提供的个数不相同，则不能省略常量表达式。

5.1.3 一维数组的引用

如果需要使用数组中的元素，可以通过数组名称加下标(也被称为索引)进行访问。数组元素的形式为：

```
数组名[下标];
```

C 语言中数组下标的值必须是整数，并且是从 0 开始的。例如：

```
int cols[5] = {6,5,4,3,2 };
```

cols[0]表示第一个元素，值为 6，cols[1]表示第 2 个元素，值为 5，如图 5.2 所示。

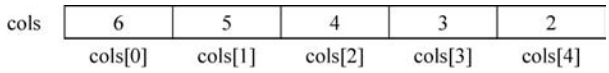

图 5.2　数组元素引用示意图

在生活中，对通过索引值获得相应的内容并不陌生。例如老师在上课时，常常会讲到翻看数学书的第 40 页，数学书就相当于数组名，不同的数组名意味着不同的书，而第 40 页中的 40 就是索引值。从数组中引用元素的过程与这类似，通过数组名找到对应的数组，然后通过索引找到对应位置的值。

例如：

```
cols[0] = cols[1] + cols[2];
```

表示将 cols[1] 的值 5 与 cols[2] 的值 4 相加，得到的结果存储到 cols[0]中，cols[0]的值变为 9。

使用数组时，要防止数组下标越界。例如，数组 cols 有 5 个元素，使用该数组时，要确保下标范围为 0～4。

【例 5.1】 编写程序，使用数组保存“俄罗斯方块”的位置信息，并显示在屏幕上，如图 5.3 所示。


视频讲解

使用数组 rows、cols 分别存储 4 个小方块的行、列信息，然后使用变量 i 存储下标，通过循环语句就能遍历数组中所有的元素，根据元素存储的位置信息，点亮对应位置的灯，就能显示出“俄罗斯方块”，代码如下：

```
#include"screen.h"
#define SIZE 8
int main(){
    initGame(SIZE);

    int rows[4] = {0,0,0,1};
    int cols[4] = {3,4,5,4};

    int i;

    for(i = 0; i < 4; i++){              //下标为 0～3
       turnOn(rows[i], cols[i]);
    }
    return 0;
}
```

编译并运行代码之后，在屏幕上显示出“俄罗斯方块”。

【例 5.2】 编写程序，实现 4 个“士兵”不断上下巡逻，严防敌人入侵，如图 5.4 所示。


视频讲解

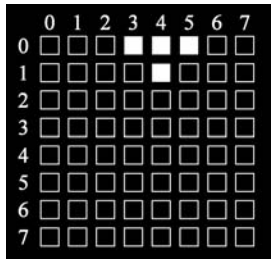

图 5.3　“俄罗斯方块”示意图

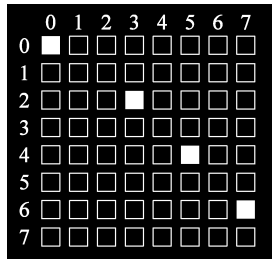

图 5.4　“士兵”巡逻示意图

在第4章综合案例中，已经完成了一个“士兵”巡逻。完成一个士兵巡逻的任务，需要3个变量才能实现，分别定义变量记录“士兵”的行、列位置信息和运动方向信息。现在有4个“士兵”则需要使用3个数组记录相应的信息，数组rows、cols分别记录“士兵”的行、列信息，数组dcols记录“士兵”的运动方向。数组dcols中的元素值为1时表示向下运动，为－1时向上运动，代码如下：

```
#include"screen.h"
#define SIZE 8

int main(){
    initGame(SIZE);

    int rows[4] = {0,2,4,6};                                    //存储每个"士兵"的行位置信息
    int cols[4] = {0,3,5,7};                                    //存储每个"士兵"的列位置信息
    int drows[4] = {1,1,1,1};                                   //存储每个"士兵"的运动方向信息

    int i;
    while(1){
        clearScreen();

        for(i = 0; i < 4; i++){
          turnOn(rows[i], cols[i] );
        }

        for(i = 0; i < 4; i++){
            rows[i] = rows [i] + drows[i];
        }

        for(i = 0; i < 4; i++){
            if(rows [i] == 0 || rows[i] == SIZE - 1){  //运动到边界时,改变方向
               drows[i] = - drows [i];
            }
        }
    }

    return 0;
}
```

编译并运行代码，4个“士兵”不断上下巡逻。

该程序如果不使用数组记录数据，则需要使用12个变量记录各种数据，程序将会非常烦琐，容易出现错误。程序中使用数组可以高效、便捷地处理数据，使其变得简洁。

【例5.3】 编写程序，实现按键W、S、A、D控制“贪吃蛇”上、下、左、右运动。图5.5所示为“贪吃蛇”向下运动示意图。

“贪吃蛇”与“俄罗斯方块”的运动规则不同，“俄罗斯方块”是整体运动，每个小方块运动方向都一致，例如向左运动，则4个小方块都相应左移一位。而“贪吃蛇”则不同，例如“贪吃蛇”向下运动，从图5.5中可以看出，0号位置的小方块往下运动，而其他位置的小方块运动方向却各不相同，从中间的图中可知1号和2号方块向左运动，3号方块向上运动，每个方块运动方向好像没有规律

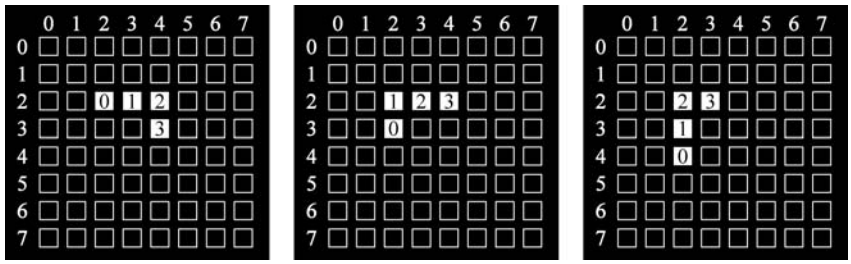

图 5.5　“贪吃蛇”向下运动示意图

似的，但是如果将“贪吃蛇”分为两部分：“蛇头”和“蛇身”，就能发现规律，0 号位置的方块为“蛇头”，其余位置的方块为“蛇身”。“贪吃蛇”运动的规律为：

（1）“蛇头”作为一个独立的部分，会根据按键控制的方向运动到相应的位置，例如向下运动，则“蛇头”行的值增加 1。

（2）“蛇身”每一部分移动到的位置，都是移动前与之相邻方块的位置。例如，2 号方块移动到 1 号方块原来所在的位置，1 号方块移动到 0 号方块原来所在的位置。

“蛇头”运动对应的代码如下：

```
char ch = getch();
/*向上运动*/
if(ch == 'w'){
  rows[0] = rows[0] - 1;
}
/*向下运动*/
if(ch == 's'){
  rows[0] = rows[0] + 1;
}
/*向左运动*/
if(ch == 'a'){
  cols[0] = cols[0] - 1;
}

/*向右运动*/
if(ch == 'd'){
  cols[0] = cols[0] + 1;
}
```

“蛇身”运动数据变化的情况：

```
rows[i] = rows[i-1];
cols[i] = cols[i-1];
```

也就是数组中的元素(除 3 号元素外)向后移一位，如图 5.6 所示。

对应的代码如下：

```
for( i = 3; i > 0; i--){
   rows[i] = row[i - 1];
   cols[i] = cols[i - 1];
}
```

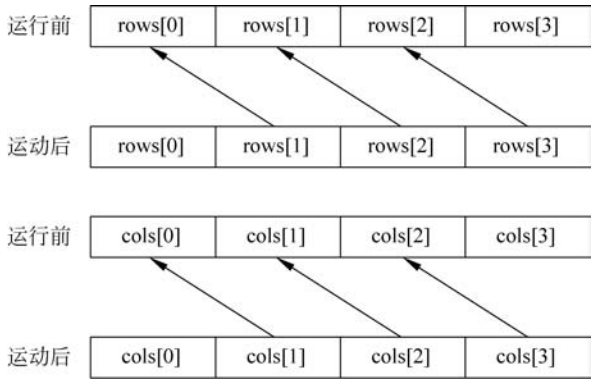

图 5.6　数组元素变化示意图

需要注意的是，要从最后一个元素倒着开始变化，如果代码如下：

```
for( i = 1; i < = 3; i++){
   rows[i] = rows[i - 1];
   cols[i] = cols[i - 1];
}
```

执行上述代码，循环过程如下：

当 i＝1 时，

```
rows[1] = rows[0]; cols[1] = cols[0];
```

当 i＝2 时，

```
rows[2] = rows[1]; cols[2] = cols[1];
```

当 i＝3 时，

```
rows[3] = rows[2]; cols[3] = cols[2];
```

则最终运行结果为：

```
rows[3] = rows[2] = rows[1] = rows[0];
cols[3] = cols[2] = cols[1] = cols[0];
```

最后所有元素的值都是数组中第 0 个元素的值，所以必须从数组中最后一个元素开始变化。同理，必须先处理完“蛇身”的运动，再处理“蛇头”的运动，否则 1 号位置的方块运动到“蛇头”移动后的位置，会导致“蛇头”“蛇身”分家。

完整代码如下：

```
#include"screen.h"
#define SIZE 8

int main(){
    initGame(SIZE);

    /*"贪吃蛇"初始位置*/
```

```
    int rows[4] = {2,2,2,3};
    int cols[4] = {2,3,4,4};

    int i;
    char ch;
    while(1){
       /*屏幕上显示"贪吃蛇"*/
       clearScreen();
       for(i = 0; i<4; i++){
          turnOn(rows[i], cols[i]);
       }

       ch = getch();
       /*按键W、S、A、D控制"贪吃蛇"上、下、左、右运动*/
       if(ch == 'w'||ch == 's'||ch == 'a'||ch == 'd'){
           for(i = 3; i>0; i--){
                rows[i] = rows[i-1];
                cols[i] = cols[i-1];
           }
       }

       /*向上运动*/
       if(ch == 'w'){
           rows[0] = rows[0] - 1;
       }

       /*向下运动*/
       if(ch == 's'){
           rows[0] = rows[0] + 1;
       }

       /*向左运动*/
       if(ch == 'a'){
           cols[0] = cols[0] - 1;
       }

       /*向右运动*/
       if(ch == 'd'){
           cols[0] = cols[0] + 1;
       }
    }

    return 0;
}
```

编译并运行代码，按键 W、S、A、D 可以控制“贪吃蛇”运动。每一次按下按键，“贪吃蛇”运动一下，这与经典的“贪吃蛇”游戏有些区别，经典游戏中的“贪吃蛇”会朝着一个方向不停地运动，除非通过按键改变方向。这个问题会在本章综合案例中解决。

5.2 二维数组

例5.1中，“俄罗斯方块”只有一种形状，而实际的游戏中总共有7种不同形状的方块。每一种形状都需要使用一个一维数组保存方块的位置信息，则总共需要7个一维数组保存7种形状。虽然该方案可以解决问题，但是代码会变得非常烦琐。例如，随机产生某种形状的方块，则需要使用多条选择语句，根据不同的条件产生对应形状的方块，如果使用二维数组就能简化代码。二维数组与一维数组很相似，一维数组是多个相同类型元素的集合，而二维数组就是多个类型和大小都相同的一维数组的集合，因此二维数组也被称为数组的数组。

5.2.1 二维数组的定义

二维数组的定义一般形式为：

```
类型说明符 数组名[行大小][列大小]
```

其中，行、列的大小必须是整型常量表达式。

例如：

```
int value[3][4];
```

表示数组value是一个二维数组，由3个一维数组组成，分别是一维数组value[0]、value[1]、value[2]，每个一维数组有4个元素，数组共有3×4＝12个数组元素，如图5.7所示。

value[0]	value[0][0]	value[0][1]	value[0][2]	value[0][3]
value[1]	value[1][0]	value[1][1]	value[1][2]	value[1][3]
value[2]	value[2][0]	value[2][1]	value[2][2]	value[2][3]

图5.7 二维数组示意图

与一维数组一样，二维数组在内存中也是按顺序存放的，先存放第0行的4个元素，接着存放第1行的4个元素。

有了二维数组的基础，三维甚至更多维数组也容易掌握。例如，定义三维数组的方法是：

```
int image[3][8][8];
```

可以把一维数组想象成一行数据，二维数组想象成一张表，三维数组想象成多张表。数组image中有3个8×8的二维数组。

5.2.2 二维数组的初始化

二维数组的初始化有如下两种方法。

(1) 分行给二维数组赋初值。例如：

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

这种方法的优点是比较直观，将第一个花括号内的数据赋给第0行的元素、第二个花括号内的数据赋给第1行的元素……即每行看作一个元素，按行赋初值。

(2) 将所有数据写在一个花括号内，按数组排列的顺序对各元素赋初值。例如：

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

只要保证个数正确，两种初始化效果是一样的。

与一维数组相似，二维数组也可以只对部分元素赋值。例如：

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

只对第 0 行的元素赋值，其余元素的值自动为 0。

5.2.3　二维数组的引用

如果要访问二维数组中的某个元素，可以通过数组名、行下标和列下标。其形式为：

```
数组名[行下标表达式][列下标表达式]
```

与一维数组一样，行、列下标都是从 0 开始的。例如：

```
int value[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
```

其中，value[0][3]的值 3，value[1][2]的值为 6，而 value[3][2]就是错误引用，行下标只能为 0，1，2，当行下标值为 3 时就越界了。

【例 5.4】 编写程序，实现每次按下按键 K，显示新的一种“俄罗斯方块”形状，“俄罗斯方块”形状如图 5.8 所示。


视频讲解

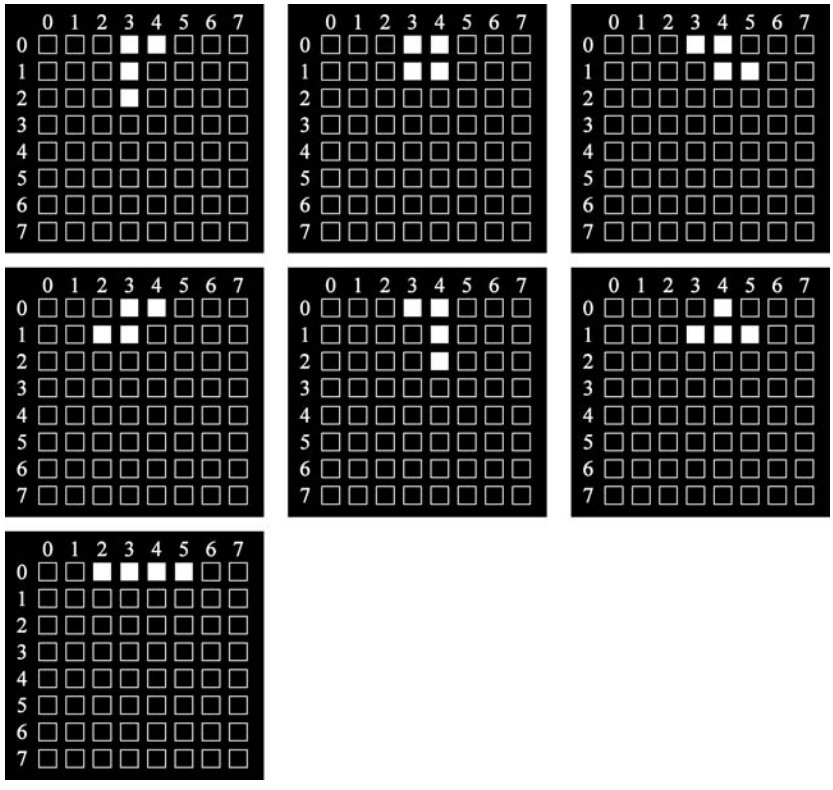

图 5.8　“俄罗斯方块”形状示意图

使用二维数组保存 7 种形状的位置信息，并且定义变量 index 记录当前显示的形状序列值，按下按键 K，index 增加 1，就能显示新的一种形状，当 index 的值等于 7 时，重新赋值为 0，这样就可以循环显示。还可以利用求余运算简化处理，对 7 求余，余数范围为 0～6，当显示到最后一种形状时，再继续单击按键，又可以显示第 0 种形状，这就是求余运算的妙用。代码如下：

```
#include"screen.h"
#define SIZE 8

int main(){
    initGame(SIZE);

   /*二维数组存储 7 种形状*/
   int rows[7][4] = {{0,0,1,2},{0,0,1,1},{0,0,1,1},{0,0,1,1},
                     {0,0,1,2},{0,1,1,1},{0,0,0,0}};
    int cols[7][4] = {{3,4,3,3},{3,4,3,4},{3,4,4,5},{3,4,3,2},
                      {3,4,4,4},{4,3,4,5},{2,3,4,5}};

    int i;
    int index = 0;
    char ch;
    while(1){
       clearScreen();
       for(i = 0; i < 4; i++){
           turnOn(rows[index][i], cols[index] [i]);
       }

       ch = getch();
       if(ch == 'k'){
           index = (index + 1) % 7;              //更新形状
       }
    }


    return 0;
}
```

运行程序，按下按键 K 可以显示不同的形状。

二维数组应用非常广泛，在第 1 章时，讲解二进制原理使用了一幅图，如图 5.9 所示。

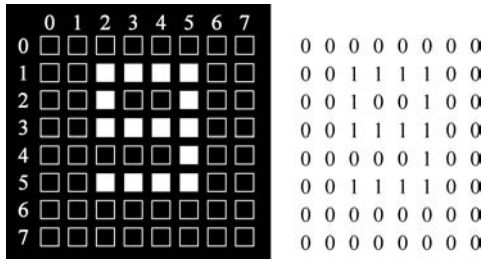

图 5.9　图像显示原理示意图

“屏幕”中显示的图像与旁边的数据形成一一对应的关系，使用二维数组保存图中右边的数据，然后根据数组中的数据点亮对应位置的灯，就能显示出图像来。只需要改变数组中的数据，屏幕就能根据数据显示各种图像，例如汉字、英文字符等。

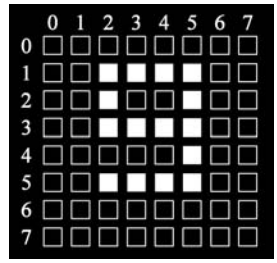

图 5.10　显示“9”

视频讲解

【例 5.5】　编写程序，显示图 5.10 所示的“9”的图像。

使用二维数组保存整个点阵的方块信息，如果该位置需要点亮，则数组对应位置的值为 1；如果该位置不需要点亮，则数组对应位置的值为 0。这样，二维数组与图像之间形成了一一对应的关系。显示出完整的“9”，只需要在二维数组中，根据“9”的形状，设置好数值即可。代码如下：

```
#include"screen.h"
#define SIZE 8
int main(){
   initGame(SIZE);

   /* 数字 9 对应的二维数组 */
   int value[SIZE][ SIZE] = {
        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,1,0,0,1,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},
   };


   int row,col;

   for(row = 0; row < SIZE; row++){
       for( col = 0; col < SIZE; col++){
           if(value [row][col] == 1){
                turnOn(row,col);
           }
       }
   }

   return 0;
}
```

编译并运行代码，屏幕上显示出“9”。只需要修改二维数组中的值，就可以显示各种各样的图像，图像在计算机中就可以用二维数组存储。如果想实现一个简单有趣的动画，可以使用三维数组，里面有多个二维数组，每个二维数组都存放一张图像信息。

视频讲解

【例 5.6】　编写程序，实现“倒计时”小动画，如图 5.11 所示。

图 5.11 中总共有 9 幅图像，每一幅图像需要一个 8×8 大小的二维数组保存数据信息。所以，

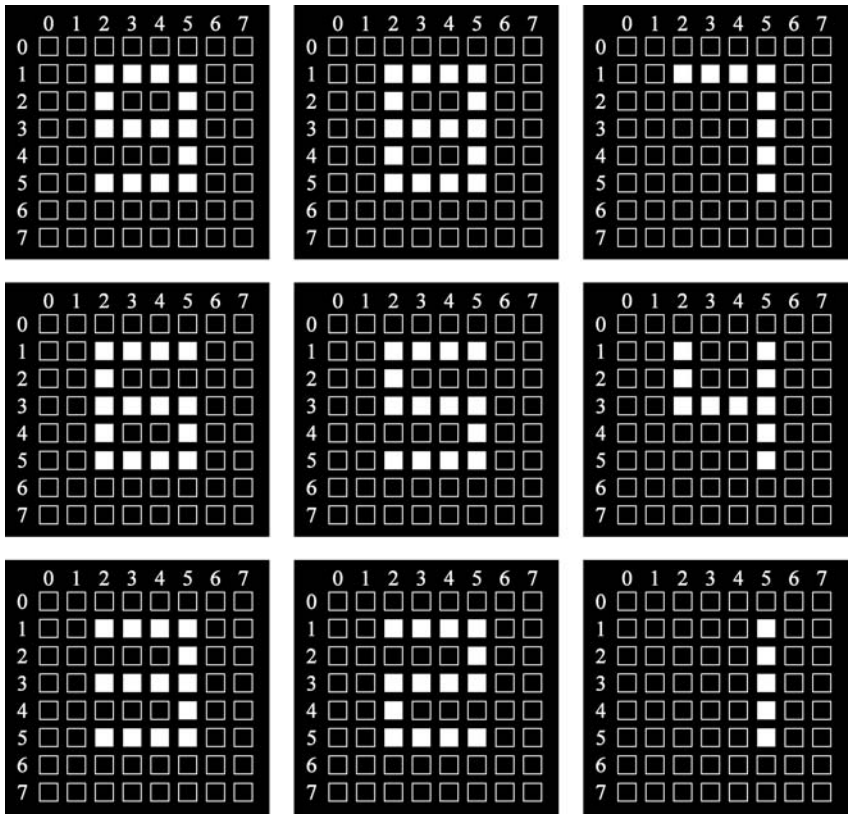

图 5.11 “倒计时”示意图

可以使用一个三维数组保存 9 幅图像的数据，大小为 9×8×8；也可以使用一个二维数组保存所有图像的数据，数组的大小为 72×8，0～7 行保存第一幅图像，8～15 行保存第二幅图像，以此类推。

使用一个二维数组保存所有图像的数据，代码如下：

```
#include"screen.h"
#define SIZE 8
#define IMAGENUM 9

int main(){
   initGame(SIZE);

   /*保存 9 幅图像的二维数组*/
   int value[ IMAGENUM * SIZE][ SIZE] = {
         {0,0,0,0,0,0,0,0},
         {0,0,1,1,1,1,0,0},
         {0,0,1,0,0,1,0,0},
         {0,0,1,1,1,1,0,0},
         {0,0,0,0,0,1,0,0},
         {0,0,1,1,1,1,0,0},
         {0,0,0,0,0,0,0,0},
         {0,0,0,0,0,0,0,0},
```

```
        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,1,0,0,1,0,0},
        {0,0,1,1,1,1,0,0 },
        {0,0,1,0,0,1,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},

        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,1,0,0 },
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},

        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,1,0,0,0,0,0},
        {0,0,1,1,1,1,0,0 },
        {0,0,1,0,0,1,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},

        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,1,0,0,0,0,0},
        {0,0,1,1,1,1,0,0 },
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},

        {0,0,0,0,0,0,0,0},
        {0,0,1,0,0,1,0,0},
        {0,0,1,0,0,1,0,0},
        {0,0,1,1,1,1,0,0 },
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},

        {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,1,0,0},
        {0,0,0,0,0,1,0,0},
        {0,0,1,1,1,1,0,0 },
```

```
            {0,0,0,0,0,1,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},

            {0,0,0,0,0,0,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,1,1,1,1,0,0 },
            {0,0,1,0,0,0,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},

            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0 },
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
    };

    int index;
    int row,col;

    for(index = 0; index < IMAGENUM; index++){
         clearScreen();
         for(row = 0; row < SIZE; row++){
             for( col = 0; col < SIZE; col++){
                if(value[index * SIZE + row][col] == 1){
                     turnOn(row,col);
                }
             }
         }
    }

    return 0;
}
```

也可以使用三维数组完成任务，代码如下：

```
#include"screen.h"
#define SIZE 8
#define IMAGENUM 9

int main(){
    initGame(SIZE);
```

```
/* 保存 9 幅图像的三维数组 */
int value [ IMAGENUM] [SIZE][ SIZE] = {
      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,0,0,0},
      {0,0,0,0,0,0,0,0}},

      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,0,0,0},
      {0,0,0,0,0,0,0,0}},

      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,0,0,0,0,0,0},
      {0,0,0,0,0,0,0,0}},

      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,0,0,0},
      {0,0,0,0,0,0,0,0}},

      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,1,0,0,0,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,1,0,0},
      {0,0,1,1,1,1,0,0},
      {0,0,0,0,0,0,0,0},
      {0,0,0,0,0,0,0,0}},

      {{0,0,0,0,0,0,0,0,0},
      {0,0,1,0,0,1,0,0},
      {0,0,1,0,0,1,0,0},
```

```
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0}},

            {{0,0,0,0,0,0,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0}},

            {{0,0,0,0,0,0,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,1,0,0,0,0,0},
            {0,0,1,1,1,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0}},

            {{0,0,0,0,0,0,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,1,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0}},

    };

    int index;
    int row,col;

    for(index = 0; index < IMAGENUM; index++){
         clearScreen();
         for(row = 0; row < SIZE; row++){
             for( col = 0; col < SIZE; col++){
                 if(value[index] [row][col] == 1){
                      turnOn(row,col);
                 }
             }
         }
    }

    return 0;
}
```

通过这个例子，可以感受到三维数组与二维数组没有本质区别，将二维数组按行分割就是三维数组。处理三维数组通常需要三重循环，四维数组需要四重循环。目前读者只需要会使用二维数组即可。三重循环有一些复杂，在程序中通常只需要使用二重循环，在第 6 章和第 10 章，会有不同的方法简化程序，将循环的层级降下来。

5.3 综合案例："贪吃蛇"游戏


视频讲解

"贪吃蛇"是一款非常经典的休闲益智类游戏，玩法非常简单，通过上、下、左、右键控制蛇的运动方向，使蛇可以吃到食物。吃到食物之后，蛇会变得越来越长，如果撞上自己的身体或者墙壁，游戏就结束。这款游戏有几十年的历史，在此期间，衍生了各种版本，如增加多人对战模式、障碍物等新型玩法。

本章需要完成的版本非常简单，按键控制"贪吃蛇"上、下、左、右运动，当"贪吃蛇"吃到食物时，身体会变得越来越长。如图 5.12 所示，上面 4 个小方块构成的是"贪吃蛇"，中间单独的方块是"食物"。

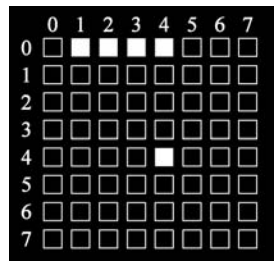

图 5.12　"贪吃蛇"游戏示意图

1. 初始化数据

"贪吃蛇"游戏主要包含两个重要角色："贪吃蛇"和"蛋"。完成游戏的第一步，需要找到合适的数据类型存储游戏元素。"贪吃蛇"由一组方块组成，可以使用一维数组来存储"贪吃蛇"的位置信息。简易版的"贪吃蛇"游戏，假定每次屏幕上只出现一个"蛋"，所以只需要单个变量就能存储"蛋"的位置信息。

"贪吃蛇"吃到"蛋"之后，身体会变长，而数组一旦定义之后，长度就固定了，那有没有可变长的数组呢？C 语言在新的 C99 标准中支持"可变长数组"，但是很多编译器暂时不支持 C99 标准。不妨先将问题简化处理，先定义一个空间较大的数组，就像平时做预算时，把额度定得大一些，留下空间以备不时之需。同理，最开始时，将数组的大小设置大一些，留下足够的空间，应付不断变长的"贪吃蛇"。

选择好了合适的数据类型，就可以定义合适的数据类型保存数据。

保存"蛋"位置的变量为：

```
int foodRow,foodCol;
```

保存"贪吃蛇"的数据数组为：

```
int snakeRows[100] = {0,0,0,0};
int snakeCols[100] = {4,3,2,1};
```

另外，还需要变量记录贪吃蛇的长度，初始长度为 4：

```
int len = 4;
```

2. 显示"贪吃蛇"

初始化数据之后，根据数据信息，可以将"贪吃蛇""蛋"显示在屏幕上，代码如下：

```
#include"screen.h"
#define SIZE 8

int main(){
    initGame(SIZE);

    int foodRow = 4;
    int foodCol = 4;

    int snakeRows[100] = {0,0,0,0};
    int snakeCols[100] = {4,3,2,1};
    int len = 4;

    int i;

    while(1){
         clearScreen();
         for(i = 0; i < len; i++){
           turnOn(snakeRows[i], snakeCols[i] );
         }
         turnOn(foodRow,foodCol);

    }


    return 0;
}
```

编译并运行代码，“贪吃蛇”和“蛋”显示在屏幕上。

3. 按键控制“贪吃蛇”运动方向

接下来实现按键 W、S、A、D 控制“贪吃蛇”上、下、左、右运动。例 5.3 中实现的“贪吃蛇”游戏与经典版的“贪吃蛇”游戏有一点点区别，经典版中的游戏中，“贪吃蛇”一直沿着某个方向运动，按键按下改变其运动方向。这个问题很容易就能解决，需要使用到项目提供的 getKey()函数，getKey()函数是在 getch()函数的基础上加工而成的。它们之间的区别在于 getch()函数是阻塞式函数，意思是这个函数不执行完，程序就一直停在这里，使用 getch()函数时，必须按下任意键，才能执行后面的程序。getKey()函数是非阻塞式函数，如果想实现经典版“贪吃蛇”运动模式，就需要改用项目提供的 getKey()函数。解决方法是定义一个变量记录“贪吃蛇”运动的方向，分别使用 1、2、3、4 代表方向上、下、左、右，如果有方向键按下，“贪吃蛇”的运动方向根据按键发生变化，否则就按原方向继续运动，代码如下：

```
#include"screen.h"
#define SIZE 8

int main(){
    initGame(SIZE);
```

```
int foodRow = 4;
int foodCol = 4;

/*"贪吃蛇"初始位置*/
int rows[100] = {2,2,2,3};
int cols[100] = {2,3,4,4};
int len = 4;

int i;
char ch;
int dir = 2;                    //初始方向为向下
while(1){
/*屏幕上显示"贪吃蛇"*/
   clearScreen();
   for(i = 0; i < len; i++){
      turnOn(rows[i], cols[i]);
   }
   turnOn(foodRow,foodCol);

   ch = getKey();
  /*按键方向键控制"贪吃蛇"运动方向*/
   if(ch == 'w'){
       dir = 1;
   }

   if(ch == 's'){
       dir = 2;
    }

   if(ch == 'a'){
       dir = 3;
    }

   if(ch == 'd'){
      dir = 4;
    }


   for(i = 3; i > 0; i--){
        rows[i] = rows[i-1];
        cols[i] = cols[i-1];
   }


   if(dir == 1){
       rows[0] = rows[0] - 1;
   }

   if(dir == 2){
```

```
            rows[0] = rows[0] + 1;
        }

        if(dir == 3){
            cols[0] = cols[0] - 1;
        }

        if(dir == 4){
            cols[0] = cols[0] + 1;
        }

    }

    return 0;
}
```

编译并运行代码，“贪吃蛇”会一直沿着某个方向运动，除非通过按键改变运动方向。如果贪吃蛇运动到屏幕的边界，即将要出界，该如何处理？读者可以尝试完善程序，解决这个问题。

完成了通过按键控制“贪吃蛇”运动方向的任务之后，如果继续完成“贪吃蛇”吃到“蛋”变长的任务，容易出现错误。因为代码已经较为复杂了，继续再往上添加新的代码，可能产生错误，导致整个代码都无法运行。第 6 章将会讲解模块化程序设计，将程序分解成一个个独立的小模块，然后再将其组合成完整的项目。模块化的设计会降低程序设计的复杂度和难度，使得程序结构更清晰、层次更明确、更容易扩充、可维护性高。

习题

5.1 “int a[10] = {1,2,3};”中数组 a 总共有_____个元素。

A. 3　　B. 11　　C. 9　　D. 10

5.2 “int a[2][3] = {1,2,3,4,5,6};”中元素 a[1][2]的值为_____。

A. 2　　B. 4　　C. 5　　D. 6

5.3 编写程序，按键控制“飞机”上、下、左、右移动，如图 5.13 所示。

5.4 编写程序，显示一颗“爱心”，如图 5.14 所示。

5.5 编写程序，显示“打砖块”游戏中的砖墙，如图 5.15 所示。

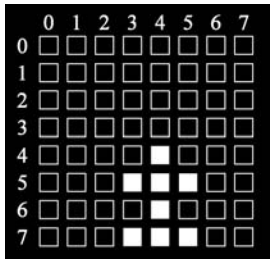

图 5.13 “飞机”示意图

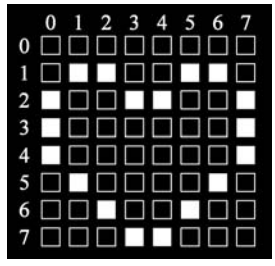

图 5.14 “爱心”示意图

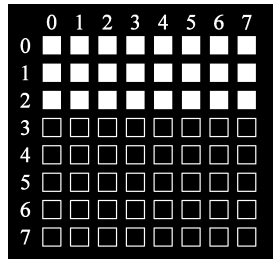

图 5.15 “砖墙”示意图

5.6 编写程序，按键 K 控制“俄罗斯方块”旋转，如图 5.16 所示。

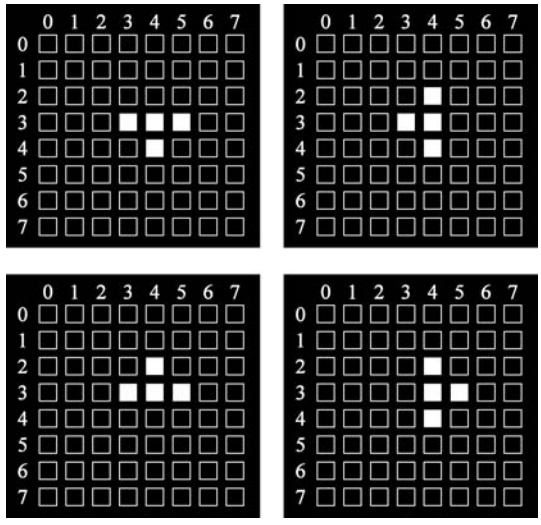

图 5.16　“俄罗斯方块”旋转示意图

5.7　编写程序，实现“俄罗斯方块”游戏中消行，当一行满行之后，然后消掉该行，该行上面的方块都往下掉一行，如图 5.17 所示。

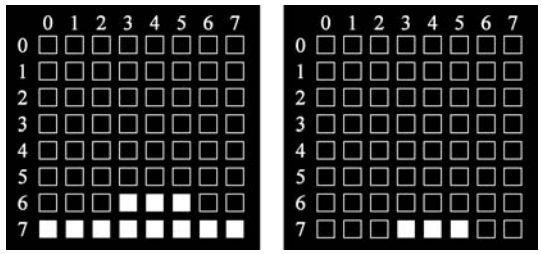

图 5.17　“俄罗斯方块”消行示意图

5.8　编写程序，实现“填满的爱心”小动画，如图 5.18 所示。

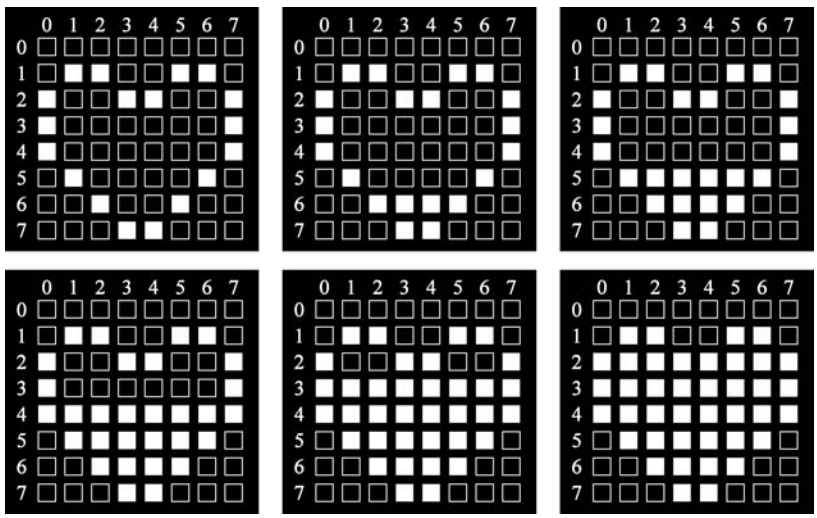

图 5.18　“填满的爱心”示意图