第4章　数组和以null结束的字符串

数组是同一类型变量的集合，可通过一个共同的名字引用这些变量。数组中的特定元素可通过一个下标访问。在C/C++中，所有的数组都是由连续的存储单元组成的。最低地址对应数组中的第一个元素，最高地址对应数组中的最后一个元素。数组可以是一维的，也可以是多维的。最常见的数组是以null结束的字符串，它只是一个以null结束的字符数组。

数组和指针是紧密相关的，讨论其中之一往往涉及到另一个。本章的重点是数组，第5章详细讨论指针。只有阅读了这两章，才能全面理解这些重要的结构。

4.1　一维数组

声明一个一维数组的一般形式是：

```
type var_name[size];
```

像其他变量一样，数组必须明确声明，这样编译器才可以为它们分配内存空间。这里，type声明了数组的基本类型，它是数组中每个元素的类型。size定义了数组中包含多少个元素。例如，为了声明一个有100个元素的数组balance，且其类型为double，可以使用下面这条语句：

```
double balance[ 100] ;
```

可以通过给数组名加下标来访问数组中的元素。通过在数组名后的方括号中放入元素的下标，可以做到这一点。例如，

```
balance[ 3]  = 12.23;
```

给balance中的第3个元素赋予值12.23。

在C/C++中，所有数组的第一个元素的下标都是0。因此，当写

```
char p[ 10] ;
```

时，你在声明一个具有10个元素的字符数组，其元素是从p[0]到p[9]。例如，下面的程序将数字0到99装入一个整数数组：

```
#include <stdio.h>

int main(void)
{
  int x[ 100] ; /* this declares a 100-integer array */
  int t;

  /* load x with values 0 through 99 */
  for(t=0; t<100; ++t) x[ t]  = t;

  /* display contents of x */
  for(t=0; t<100; ++t) printf("%d ", x[ t] );
```

```
  return 0;
}
```

存放一个数组所要求的存储量与数组的类型和长度有直接的关系。对于一个一维数组，以字节计的总长度可计算如下：

总字节数（total bytes）= sizeof（基本类型）× 数组长度（size of array）

C/C++不对数组执行边界检查。可以重写数组的每一端，并写入一些其他变量的数据或者甚至是写入程序的代码。作为程序员，你应该在需要的地方提供边界检查。例如，下面的代码能够编译，并没有错误，但是却不是正确的，因为 for 循环将导致 count 数组越界。

```
int count[ 10], i;

/* this causes count to be overrun */
for(i=0; i<100; i++) count[ i] = i;
```

一维数组本质上是同类型信息的列表，它们是按下标顺序以连续的内存地址存储的。例如，图 4.1 显示了如果数组在内存地址 1000 处开始，它是怎样出现在内存的，它的声明如下所示：

```
char a[ 7];
```

元素	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
内存地址	1000	1001	1002	1003	1004	1005	1006

图 4.1 在地址 1000 处开始的包含 7 个元素的字符数组

4.2 生成指向数组的指针

通过指定数组名，而不是任何下标，可以生成一个指向数组第一个元素的指针。例如，给定下面的语句，

```
int sample[ 10];
```

使用名字 sample，可以生成一个指向第一个元素的指针。因此，下面的代码段把 sample 的第一个元素的地址赋给了 p：

```
int *p;
int sample[ 10];
p = sample;
```

也可以使用 & 运算符来指定数组第一个元素的地址。例如，sample 和 &sample[0]产生同样的结果。然而，在所编写的专业级 C/C++ 代码中，几乎永远不会看到 &sample[0]。

4.3 向函数传递一维数组

在 C/C++ 中，不能把整个数组作为一个参数传递给函数。然而，可以通过指定数组的名字而不是下标，把一个指向数组的指针传递给函数。例如，下面的代码段把 i 的地址传给了 func1()：

```
int main(void)
{
  int i[ 10];
```

```
    func1(i);
    .
    .
    .
}
```

如果函数收到一个一维数组，可以用三种方式声明它的形参：作为一个指针，作为一个固定长度的数组，或作为一个不定长的数组。例如，要接收 i，称为 func1( )的函数可以声明为：

```
void func1(int *x) /* pointer */
{
   .
   .
   .
}
```

或

```
void func1(int x[10]) /* sized array */
{
   .
   .
   .
}
```

或

```
void func1(int x[]) /* unsized array */
{
   .
   .
   .
}
```

所有的三种声明方法都生或类似的结果，因为每个都通知编译器将要收到一个整数指针。第一个声明实际上使用一个指针，第二个采用标准数组声明，最后一个，即数组声明的一个修改版本简单地规定：将要收到一个某种长度的 int 型的数组。如你所见，就函数所关心的，数组的长度并不重要，因为 C/C++ 不执行边界检查。事实上，就编译器所关心的，

```
void func1(int x[32])
{
   .
   .
   .
}
```

上面的代码也是正确的，因为编译器生或指示 func1( )接收一个指针的代码——实际上它不创建一个包含 32 个元素的数组。

4.4　以 null 结束的字符串

迄今为止，一维数组最常见的用途是作为字符串。C++ 支持两种字符串。第一种是以 null

结束的字符串，它是一个以null结束的字符数组（null就是0）。因此，以null结束的字符串包含这样的字符：这些字符组成字符串，后跟一个空值。这是由C定义的惟一一种字符串，仍然在广泛使用着。有时以null结束的字符串也称为C串。C++也定义了一个字符串，称为string，它对字符串处理提供面向对象的方法，本书后面将会描述它。这里只讨论以null结束的字符串。

当声明一个将存放以null结束的字符串的字符数组时，需要把它声明为比它要存放的最大字符串长一个字符。例如，为了声明一个能存放含10个字符的字符串的数组str，应该这样写：

```
char str[11];
```

这为字符串末尾处的null留下了空间。

在程序中使用加了引号的字符串常量时，你也在创建一个以null结束的字符串。一个字符串常量是封在双引号中的一个字符列表。例如，

```
"hello there"
```

不需要用手工的方式在字符串常量的末尾处添加null——编译器会自动添加。

C/C++支持很多操作以null结束的字符串的函数。最常见的如下表所示：

名称	函数
strcpy(*s1*, *s2*)	把s2复制到s1中
strcat(*s1*, *s2*)	把s2连接到s1的末尾
strlen(*s1*)	返回s1的长度
strcmp(*s1*, *s2*)	如果s1与s2相同，返回0；如果s1<s2，返回小于0的数；如果s1>s2，返回大于0的数
strchr(*s1*, *ch*)	返回一个指向s1中第一次出现的ch的指针
strstr(*s1*, *s2*)	返回一个指向s1中第一次出现的s2的指针

这些函数使用标准的头文件string.h（C++程序也可以使用C++风格的头文件<cstring>）。下面的程序演示了这些字符串函数的使用情况：

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

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  printf("lengths: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf("The strings are equal\n");

  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "This is a test.\n");
  printf(s1);
  if(strchr("hello", 'e')) printf("e is in hello\n");
  if(strstr("hi there", "hi")) printf("found hi");

  return 0;
}
```

如果运行这个程序并键入字符串 "hello" 和 "hello"，则输出是：

```
lengths: 5 5
The strings are equal
hellohello
This is a test.
e is in hello
found hi
```

记住，如果字符串相等，strcmp( )返回假。如果正在测试是否相等（如刚才所示），一定要使用逻辑运算符！来颠倒条件。

尽管 C++ 定义了一个字符串类，在现有的程序中，仍然广泛使用着以 null 结束的字符串。它们还将得到广泛的使用，因为它们的高效性以及它们可以使程序员能够详细控制字符串操作。然而，对许多简单的字符串处理工作来讲，使用 C++ 的字符串类是很方便的。

4.5　二维数组

C/C++ 支持多维数组，多维数组最简单的形式是二维数组。本质上，二维数组是一维数组的数组。要声明一个大小为 10，20 的二维整数数组 d，可以这样写：

```
int d[10][20];
```

请仔细注意这个声明。有些计算机语言使用逗号来分隔数组的维，而 C/C++ 把每维放在它自己的括号里。

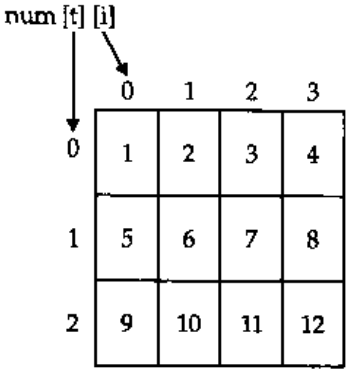
类似地，要访问数组 d 的点 1,2，可以使用下面的代码：

```
d[1][2]
```

下面的例子装入一个数字从 1 到 12 的二维数组并一行一行地显示它们。

```
#include <stdio.h>

int main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

  /* now print them out */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }

  return 0;
}
```

在这个例子中，num[0][0]的值是 1，num[0][1]的值是 2，num[0][2]的值是 3，等等。num[2][3]的值是 12。可以形象地表示 num 数组如下：



二维数组存储在一个行列矩阵中，其中第一个下标表示行，第二个下标表示列。这意味着当按数组元素实际存放在内存中的顺序访问数组中的元素时，最右边的下标比最左边的下标改变得更快。关于二维数组在内存中的图形表示情况，参见图4.2。

如果有：char ch[4] [3]

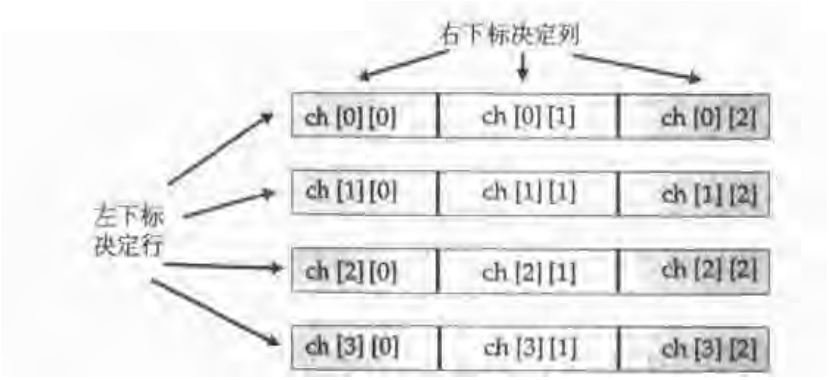


图4.2 内存中的二维数组

在二维数组中，下面的等式生成需要存储二维数组的内存字节数：

字节 = 第1组下标的大小 × 第2组下标的大小 × sizeof（基本类型）

因此，假定4字节的整数，一个具有维数为10,5的整数数组会分配10 × 5 × 4或200字节。

当把一个二维数组用做一个函数的参数时，实际上只传递了一个指向第一个元素的指针。然而，接受二维数组的参数必须定义至少最右边维的大小（如果喜欢，可以指定左边的维，但它是不必要的）。之所以需要最右边的维，是因为如果编译器要给数组标上正确的下标，它需要知道每行的长度）。例如，一个接受二维整数数组（其维数是10,10）的函数是这样声明的：

```
void func1(int x[][10])
{
    .
    .
    .
}
```

为了正确地执行函数中像下面这样的表达式，编译器需要知道右边维数的大小。

```
x[2][4]
```

如果不知道行的长度，编译器就不能确定第三行在哪里开始。

下面的程序段使用一个二维数组来存储一个老师所教的班中每位学生的分数等级。这个程

序假定这名老师教三个班，每班最多有 30 名学生。注意每个函数访问数组 grade 的方式。

```
/* A simple student grades database. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define CLASSES 3
#define GRADES 30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

int main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(E)nter grades\n");
      printf("(R)eport grades\n");
      printf("(Q)uit\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='E' && ch!='R' && ch!='Q');

    switch(ch) {
      case 'E':
        enter_grades();
        break;
      case 'R':
        disp_grades(grade);
        break;
      case 'Q':
        exit(0);
    }
  }

  return 0;
}

/* Enter the student's grades. */
void enter_grades(void)
{
  int t, i;

  for(t=0; t<CLASSES; t++) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i] = get_grade(i);
  }
}
```

```
/* Read a grade. */
int get_grade(int num)
{
  char s[80];

  printf("Enter grade for student # %d:\n", num+1);
  gets(s);
  return(atoi(s));
}

/* Display grades. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Class # %d:\n", t+1)
    for(i=0; i<GRADES; ++i)
      printf("Student #%d is %d\n", i+1, g[t][i]);
  }
}
```

4.5.1 字符串数组

在编程中使用字符串数组是很常见的。例如，数据库的输入处理器可以根据一个有效命令数组来验证用户命令。要创建一个以null结束的字符串数组，使用一个二维字符数组。左边下标的大小决定字符串数，而右边下标的大小规定了每个字符串的最大长度。下面的代码声明了一个具有30个字符串的数组，其中每个字符串包含的字符最多为79个，另加一个null结束符。

```
char str_array[30][80];
```

很容易访问各个字符串：可以简单地只指定左边的下标。例如，下面的语句调用以str_array中第3个字符串为参数的gets( )。

```
gets(str_array[2]);
```

前面的例子在功能上等价于：

```
gets(&str_array[2][0]);
```

但是在专业人士编写的C/C++代码中，第一种形式更常见。

为了更好地理解字符串数组是如何工作的，让我们研究一下下面的程序，这个程序使用一个字符串数组作为一个简单文本编辑器的基础。

```
/* A very simple text editor. */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
  register int t, i, j;
```

```
  printf("Enter an empty line to quit.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* quit on blank line */
  }

  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++) putchar(text[i][j]);
    putchar('\n');
  }

  return 0;
}
```

这个程序一直输入文本行，直到遇到一个空行为止。然后，它一次一个字符地重新显示每一行。

4.6　多维数组

C/C++ 允许二维以上的数组存在，其维数限制（如果有的话）是由编译器决定的。多维数组声明的一般形式是：

type name[Size1][Size2][Size3]...[SizeN];

三维以上的数组不常使用，因为它们要求更多的内存。例如，维数为 10,6,9,4 的四维字符数组要求

```
10 * 6 * 9 * 4
```

或 2160 字节。如果这个数组存放 2 字节整数，将需要 4320 字节。如果这个数组存放 double 型数（假定每个 double 型数为 8 个字节），则要求 17 280 字节。

所要求的存储空间与维数成指数型增长。例如，如果前面的数组增加了一个大小为 10 的第五维，那么，将要求 172 800 个字节。

在多维数组中，计算机需要花时间来计算每个下标。这意味着访问多维数组中的一个元素比访问一维数组中的元素要慢。

当把多维数组传递给函数时，必须声明除了最左边维以外的所有维。例如，如果声明数组 m 为：

```
int m[4][3][6][5];
```

接收 m 的函数 func1( )会像这样：

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

当然，如果喜欢，可以包含第一维。

4.7 带下标的指针

在C/C++中，指针和数组是密切相关的。我们知道，没有下标的数组名是一个指向数组中第一个元素的指针。例如，考虑下面的数组：

```
char p[ 10] ;
```

下面的语句与上面的相同：

```
p
&p[ 0]
```

其另一种方式，

```
p == &p[ 0]
```

的求值结果为真，因为一个数组中第一个元素的地址与这个数组的地址是一样的。

正如所述，没有下标的数组名生成一个指针。反之，一个指针可以带有下标，仿佛它被声明为一个数组。例如，考虑下面的代码段：

```
int *p, i[ 10] ;
p = i;
p[ 5] = 100; /* assign using index */
*(p-5) = 100; /* assign using pointer arithmetic */
```

两个赋值语句都给i的第6个元素赋予了值100。第一个语句给p加下标，第二个使用指针算法。无论是哪一种方式，结果是一样的（第5章将讨论指针和指针运算）。

同样的概念也适用于二维或多维数组。例如，假定a是一个10 × 10的整数数组，那么这两条语句是等价的：

```
a
&a[ 0][ 0]
```

此外，数组a中的元素0,4可以以两种方式引用：通过数组下标a[0][4]，或通过指针*((int *)a+4)。类似地，元素1,2为a[1][2]或*((int *)a+12)。一般来讲，对于任何二维数组

a[j][k]等价于 *((*base-type* *)a+(j**row length*)+k)

为了让指针运算得以正确操作，必须把指向数组的指针转换为它的基类型的指针。有时我们使用指针来访问数组，因为指针运算经常比数组下标更快。

二维数组可以被还原为指向一维数组的数组的指针。因此，使用一个分开的指针变量比使用指针来访问二维数组行中的元素更容易。下面的函数演示了这项技术，它将显示全局整数数组num中指定行的内容：

```
int num[ 10][ 10] ;
  .
  .
  .
void pr_row(int j)
{
  int *p, t;
```

```
  p = (int *) &num[j][0]; /* get address of first
                             element in row j */
  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

可以通过让调用参数为行、行的长度和指向第一个数组元素的指针来归纳这个例程，如下所示：

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;
  p = p + (j * row_dimension);
  for(t=0; t<row_dimension; ++t)
    printf("%d ", *(p+t));
}
  .
  .
  .
void f(void)
{
  int num[10][10];
  pr_row(0, 10, (int *) num); /* print first row */
}
```

二维以上的数组可以用类似的方法还原。例如，一个三维数组可以还原为一个指向二维数组的指针，而这个二维数组可以还原为指向一个一维数组的指针。一般来讲，n维数组可以还原为一个指针和一个(n-1)维数组。这个新的数组可以用同样的方法再次还原。当生成一维数组时，这个过程结束。

4.8 数组初始化

C/C++允许在声明数组时初始化它们。数组初始化的一般形式类似于其他变量的初始化，如下所示：

type_specifier array_name[*size1*]...[*sizeN*] = { *value_list* };

value_list是一个用逗号分开的值列表，它的类型与type_specifier兼容。第一个值放在数组的第一个位置，第二个值放在第二个位置，等等。注意跟在}后面的分号。

在下面的例子中，一个包含10个元素的整数数组用数1到10初始化：

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

这意味着i[0]的值为1，而i[9]的值为10。

存放字符串的字符数组允许使用简写的初始化，其形式是：

char *array_name*[*size*] = *"string"*;

例如，下面的代码段初始化str为词语“I like C++”。

```
char str[11] = "I like C++";
```

它与下面的写法是一样的：

```
char str[ 11] = { 'I', ' ', 'l', 'i', 'k', 'e',' ', 'C',
                  '+', '+', '\0'};
```

因为以null结束的字符串用空格结束，所以必须确保你声明的数组长的足以包含这个空格。这就是为什么str是11个字符长，即使“I like C++”只有10个字符。当使用字符串常量时，编译器自动提供null结束符。

多维数组的初始化与一维数组相同。例如，下面的代码用数1到10和它们的平方初始化sqrs。

```
int sqrs[ 10][ 2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100
};
```

在初始化一个多维数组时，可以在每一维的初始化部分周围添加括号，这称为子聚集分组（subaggregate grouping）。例如，下面是书写前面的声明的另一种方式。

```
int sqrs[ 10][ 2] = {
  { 1, 1},
  { 2, 4},
  { 3, 9},
  { 4, 16},
  { 5, 25},
  { 6, 36},
  { 7, 49},
  { 8, 64},
  { 9, 81},
  { 10, 100}
};
```

当使用子聚集分组时，如果没有为给定的每组提供足够的初始化，其余的成员将被自动设置为0。

4.8.1 变长数组的初始化

假定你正在使用数组初始化来构建一个错误信息表，如下所示：

```
char e1[ 12] = "Read error\n";
char e2[ 13] = "Write error\n";
char e3[ 18] = "Cannot open file\n";
```

像可能猜到的那样，用手工的方式计算每条消息中的字符以确定正确的数组维数是非常麻

烦的。幸运的是，可以让编译器自动计算数组的维数。如果在数组初始化语句中没有指定数组的长度，C/C++编译器自动创建一个足以存放所有初始部分的数组。这种数组称为不固定长度数组。使用这种方法，信息表变为

```
char e1[ ] = "Read error\n";
char e2[ ] = "Write error\n";
char e3[ ] = "Cannot open file\n";
```

给定了这些初始化后，下面这条语句：

```
printf("%s has length %d\n", e2, sizeof e2);
```

将显示

```
Write error has length 13
```

除了较少乏味外，变长数组初始化允许你改变任何信息而不用害怕使用了不正确的数组维数。

变长数组初始化并不局限于一维数组。对于多维数组，必须指定除了最左边以外的所有维（需要其他维来让编译器给数组加合适的下标）。用这种方式，可以构建变长表，且编译器自动地为它们分配足够的存储空间。例如，把 sqrs 声明为变长数组的代码如下所示：

```
int sqrs[ ][ 2] = {
  { 1, 1},
  { 2, 4},
  { 3, 9},
  { 4, 16},
  { 5, 25},
  { 6, 36},
  { 7, 49},
  { 8, 64},
  { 9, 81},
  { 10, 100}
};
```

比起定长版本的声明来说，这个声明的优点是可以延长或缩短此表而无需改变数组的维数。

4.9　棋盘游戏实例

下面的例子演示了在C/C++中可以操作数组的多种方式。本节开发一个简单的棋盘游戏程序。二维数组经常用于模拟棋盘游戏的矩阵。

计算机的玩法很简单。当轮到计算机开始下棋时，它使用get_computer_move( ) 扫描矩阵，查找未占据的单元。当找到一个时，它在那里放一个O。如果没有找到一个空的位置，它将报告双方下成平局并退出程序。函数 get_player_move( )提示下棋者键入要放入X的位置。矩阵右上角的位置定为 1,1，右下角的位置定为 3,3。

矩阵数组被初始化为包含空格。下棋者或计算机做的每一个动作都会把一个空格变为X或O。这样，就可以很容易地在屏幕上显示矩阵了。

每走完一步，程序调用 check( )函数。如果还没有分出输赢，这个函数返回一个空格；如果下棋者赢，返回一个 X；如果计算机赢，则返回一个 O。它扫描棋盘的行、列、对角，查找

包含所有X或所有O的那些位置。

disp_matrix( )函数显示游戏的当前状态。注意用空格初始化这个矩阵如何简化了这个函数。

这个例子中的各个例程以不同的方式访问矩阵。我们应该研究一下它们以确定理解了各种数组操作。

```
/* A simple Tic Tac Toe game. */
#include <stdio.h>
#include <stdlib.h>

char matrix[ 3][ 3]; /* the tic tac toe matrix */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

int main(void)
{
  char done;

  printf("This is the game of Tic Tac Toe.\n");
  printf("You will be playing against the computer.\n");

  done = ' ';
  init_matrix();
  do{
    disp_matrix();
    get_player_move();
    done = check(); /* see if winner */
    if(done!= ' ') break; /* winner!*/
    get_computer_move();
    done = check(); /* see if winner */
  } while(done== ' ');
  if(done=='X') printf("You won!\n");
  else printf("I won!!!!\n");
  disp_matrix(); /* show final positions */

  return 0;
}

/* Initialize the matrix. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++) matrix[ i][ j] = ' ';
}

/* Get a player's move. */
void get_player_move(void)
{
  int x, y;
```

```
  printf("Enter X,Y coordinates for your move: ");
  scanf("%d%*c%d", &x, &y);

  x--; y--;

  if(matrix[x][y] != ' '){
    printf("Invalid move, try again.\n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}

/* Get a move from the computer. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }
  if(i*j==9) {
    printf("draw\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
}

/* Display the matrix on the screen. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c | %c | %c ",matrix[t][0],
           matrix[t][1], matrix[t][2]);
    if(t!=2) printf("\n---|---|---\n");
  }
  printf("\n");
}

/* See if there is a winner. */
char check(void)
{
  int i;

  for(i=0; i<3; i++) /* check rows */
    if(matrix[i][0]==matrix[i][1] &&
      matrix[i][0]==matrix[i][2]) return matrix[i][0];

  for(i=0; i<3; i++) /* check columns */
    if(matrix[0][i]==matrix[1][i] &&
      matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* test diagonals */
```

```
    if(matrix[ 0][ 0] ==matrix[ 1][ 1]  &&
      matrix[ 1][ 1] ==matrix[ 2][ 2] )
        return matrix[ 0][ 0] ;

    if(matrix[ 0][ 2] ==matrix[ 1][ 1]  &&
      matrix[ 1][ 1] ==matrix[ 2][ 0] )
        return matrix[ 0][ 2] ;

    return ' ';
}
```