CHAPTER 5

Arrays

You'll often need to store many data values of a particular kind in your programs. For example, if you were writing a program to track the performance of a basketball team, then you might want to store the scores for a season of games and the scores for individual players. You could then output the scores for a particular player over the season or work out an ongoing average as the season progresses. Armed with what you've learned so far, you could write a program that does this using a different variable for each score. However, if there are a lot of games in the season, this will be rather tedious because you'll need as many variables for each player as there are games. All your basketball scores are really the same kind of thing. The values are different, but they're all basketball scores. Ideally, you would want to group these values together under a single name—perhaps the name of the player—so that you wouldn't have to define separate variables for each item of data.

In this chapter, I'll show you how to do just that using arrays in C. I'll then show you how powerful referencing a set of values through a single name can be when you write programs that process arrays.

In this chapter you'll learn the following:

- What arrays are
- How to use arrays in your programs
- How memory is used by an array
- What a multidimensional array is
- How to write a program to work out your hat size
- How to write a game of tic-tac-toe

An Introduction to Arrays

The best way to show you what an array is and how powerful it can be is to go through an example in which you can see how much easier a program becomes when you use an array. For this example, you'll look at ways in which you can find the average score of the students in a class.

Programming Without Arrays

To find the average score of a class of students, assume that there are only ten students in the class (mainly to avoid having to type in a lot of numbers). To work out the average of a set of numbers, you add them all together and then divide by how many you have (in this case, by 10):

```
/* Program 5.1 Averaging ten numbers without storing the numbers */
#include <stdio.h>

int main(void)
{
  int number = 0;                    /* Stores a number             */
  int count = 10;                    /* Number of values to be read */
  long sum = 0L;                     /* Sum of the numbers          */
  float average = 0.0f;              /* Average of the numbers      */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i ++)
  {
    printf("Enter grade: ");
    scanf("%d", &number);            /* Read a number */
    sum += number;                   /* Add it to sum */
  }

  average = (float)sum/count;        /* Calculate the average */

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

If you're interested only in the average, then you don't have to remember what the previous grades were. All you're interested in is the sum of them all, which you then divide by `count`, which has the value 10. This simple program uses a single variable, `number`, to store each grade as it is entered within the loop. The loop repeats for values of `i` of 1, 2, 3, and so on, up to 9, so there are ten iterations. You've done this sort of thing before, so the program should be clear.

But let's assume that you want to develop this into a more sophisticated program in which you'll need the values you enter later. Perhaps you might want to print out each person's grade, with the average grade next to it. In the previous program, you had only one variable. Each time you add a grade, the old value is overwritten, and you can't get it back.

So how do you store the results? You could do this is by declaring ten integers to store the grades in, but then you can't use a `for` loop to enter the values. Instead, you have to include code that will read the values individually. This would work, but it's quite tiresome:

```
/* Program 5.2 Averaging ten numbers - storing the numbers the hard way */
#include <stdio.h>

int main(void)
{
  int number0 = 0, number1 = 0, number2 = 0, number3 = 0, number4 = 0;
  int number5 = 0, number6 = 0, number7 = 0, number8 = 0, number9 = 0;

  long sum = 0L;          /* Sum of the numbers     */
  float average = 0.0f;   /* Average of the numbers */

  /* Read the ten numbers to be averaged */
  printf("Enter the first five numbers,\n");
  printf("use a space or press Enter between each number.\n");
  scanf("%d%d%d%d%d", &number0, &number1, &number2, &number3, &number4);
  printf("Enter the last five numbers,\n");
  printf("use a space or press Enter between each number.\n");
  scanf("%d%d%d%d%d", &number5, &number6, &number7, &number8, &number9);
```

```
  /* Now we have the ten numbers, we can calculate the average */
  sum = number0 + number1+ number2 + number3 + number4+
        number5 + number6 + number7 + number8 + number9;
  average = (float)sum/10.0f;

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

This is more or less OK for ten students, but what if your class has 30 students, or 100, or 1,000? How can you do it then? Well, this is where this approach would become wholly impractical and arrays become essential.

What Is an Array?

An **array** is a fixed number of data items that are all of the same type. The data items in an array are referred to as **elements**. These are the most important feature of an array—there is a fixed number of elements and the elements of each array are all of type `int`, or of type `long`, or all of type whatever. So you can have arrays of elements of type `int`, arrays of elements of type `float`, arrays of elements of type `long`, and so on.

The following array declaration is very similar to how you would declare a normal variable that contains a single value, except that you've placed a number between square brackets [] following the name:

```
long numbers[10];
```

The number between square brackets defines how many elements you want to store in your array and is called the array **dimension**. The important feature here is that each of the data items stored in the array is accessed by the same name; in this case, `numbers`.

If you have only one variable name but are storing ten values, how do you differentiate between them? Each individual value in the array is identified by what is called an **index value**. An index value is an integer that's written after the array name between square brackets []. Each element in an array has a different **index value**, which are sequential integers starting from 0. The index values for the elements in the preceding `numbers` array would run from 0 to 9. The index value 0 refers to the first element and the index value 9 refers to the last element. To access a particular element, just write the appropriate index value between square brackets immediately following the array name. Therefore, the array elements would be referred to as `numbers[0]`, `numbers[1]`, `numbers[2]`, and so on, up to `numbers[9]`. You can see this in Figure 5-1.

Don't forget, index values start from 0, not 1. It's a common mistake to assume that they start from 1 when you're working with arrays for the first time, and this is sometimes referred to as *the off-by-one error*. In a ten-element array, the index value for the last element is 9. To access the fourth value in your array, you use the expression `numbers[3]`. You can think of the index value for an array element as the offset from the first element. The first element is the first element, so it has an offset of 0. The second element has an offset of 1 from the first element, the third element has an offset of 2 from the first element, and so on.

To access the value of an element in the `numbers` array, you could also place an expression in the square brackets following the array name. The expression would have to result in an integer value that corresponds to one of the possible index values. For example, you could write `numbers[i-2]`. If `i` had the value 3, this would access `numbers[1]`, the second element in the array. Thus there are two ways to specify an index to access a particular element of an array. You can use a simple integer to explicitly reference the element that you want to access. Alternatively, you can use an integer expression that's evaluated during the execution of the program. When you use an expression the only constraints are that it must produce an integer result and the result must be a legal index value for the array.

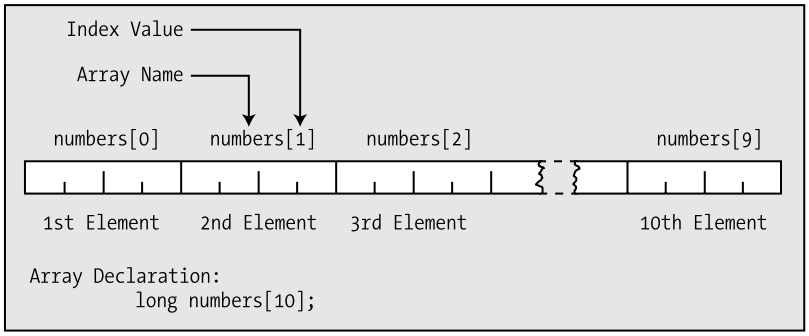


Figure 5-1. *Accessing the elements of an array*

Note that if you use an index value in your program that's outside the legal range for an array, the program won't work properly. The compiler can't check for this, so your program will still compile, but execution is likely to be less than satisfactory. At best you'll just pick up a junk value from somewhere so that the results are incorrect and may vary from one run to the next. At worst the program may overwrite something important and lock up your computer so a reboot becomes necessary. It is also possible that the effect will be much more subtle with the program sometimes working and sometimes not, or the program may appear to work but the results are wrong but not obviously so. It is therefore most important to check carefully that your array indexes are within bounds.

Using Arrays

That's a lot of theory, but you still need to solve your average score problem. Let's put what you've just learned about arrays into practice in that context.

TRY IT OUT: AVERAGES WITH ARRAYS

Now that you understand arrays, you can use an array to store all the scores you want to average. This means that all the values will be saved, and you'll be able to reuse them. You can now rewrite the program to average ten scores:

```
/* Program 5.3 Averaging ten numbers - storing the numbers the easy way */
#include <stdio.h>

int main(void)
{
  int numbers[10];                    /* Array storing 10 values     */
  int count = 10;                     /* Number of values to be read */
  long sum = 0L;                      /* Sum of the numbers          */
  float average = 0.0f;               /* Average of the numbers      */

  printf("\nEnter the 10 numbers:\n");       /* Prompt for the input */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i ++)
  {
    printf("%2d> ",i+1);
    scanf("%d", &numbers[i]);         /* Read a number */
```

```
 sum += numbers[i];                /* Add it to sum */
  }

  average = (float)sum/count;          /* Calculate the average */

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

The output from the program looks something like this:

```
Enter the ten numbers:
 1> 450
 2> 765
 3> 562
 4> 700
 5> 598
 6> 635
 7> 501
 8> 720
 9> 689
10> 527

Average of the ten numbers entered is: 614.700000
```

How It Works

You start off the program with the ubiquitous `#include` directive for `<stdio.h>` because you want to use `printf()` and `scanf()`. At the beginning of `main()`, you declare an array of ten integers and then the other variables that you'll need for calculation:

```
  int numbers[10];                  /* Array storing 10 values     */
  int count = 10;                   /* Number of values to be read */
  long sum = 0L;                    /* Sum of the numbers          */
  float average = 0.0f;             /* Average of the numbers      */
```

You then prompt for the input to be entered with this statement:

```
  printf("\nEnter the 10 numbers:\n");  /* Prompt for the input */
```

Next, you have a loop to read the values and accumulate the sum:

```
  for(int i = 0; i < count; i++)
  {
    printf("%2d> ",i+1);
    scanf("%d", &numbers[i]);          /* Read a number */
    sum += numbers[i];                 /* Add it to sum */
  }
```

The `for` loop is in the preferred form with the loop continuing as long as `i` is not equal to the limit, `count`. In general you should write your `for` loops like this if you can. Because the loop counts from 0 to 9, rather than from 1 to 10, you can use the loop variable `i` directly to reference each of the members of the array. The `printf()` call outputs the current value of `i+1` followed by `>`, so it has the effect you see in the output. By using `%2d` as the format specifier, you ensure that each value is output in a two-character field, so the numbers are aligned. If you had used `%d` instead, the output for the tenth value would have been out of alignment.

You read each value entered into element `i` of the array using the `scanf()` function; the first value will be stored in `number[0]`, the second number entered will be stored in `number[1]`, and so on up to the tenth value entered, which will be stored in `number[9]`. For each iteration of the loop, the value that was read is added to `sum`.

When the loop ends, you calculate the average and display it with these statements:

```
average = (float)sum/count;          /* Calculate the average */

printf("\nAverage of the ten numbers entered is: %f\n", average);
```

You've calculated the average by dividing the sum by `count`, which has the value 10. Notice how, in the call to `printf()`, you've told the compiler to convert `sum` (which is declared as type `long`) into type `float`. This is to ensure that the division is done using floating-point values, so you don't discard any fractional part of the result.

TRY IT OUT: RETRIEVING THE NUMBERS STORED

You can expand it a little to demonstrate one of the advantages. I've made only a minor change to the original program (highlighted in the following code in **bold**), but now the program displays all the values that were typed in. Having the values stored in an array means that you can access those values whenever you want and process them in many different ways.

```
/* Program 5.4 Reusing the numbers stored */
#include <stdio.h>

int main(void)
{
  int numbers[10];                   /* Array storing 10 values     */
  int count = 10;                    /* Number of values to be read */
  long sum = 0L;                     /* Sum of the numbers          */
  float average = 0.0f;              /* Average of the numbers      */

  printf("\nEnter the 10 numbers:\n");        /* Prompt for the input */

  /* Read the ten numbers to be averaged */
  for(int i = 0; i < count; i++)
  {
    printf("%2d> ",i+1);
    scanf("%d", &numbers[i]);        /* Read a number */
    sum += numbers[i];               /* Add it to sum */
  }

  average = (float)sum/count;        /* Calculate the average */

  for(int i = 0; i < count; i++)
    printf("\nGrade Number %d was %d", i+1, numbers[i]);

  printf("\nAverage of the ten numbers entered is: %f\n", average);
  return 0;
}
```

Typical output from this program would be as follows:

```
Enter the ten numbers:
 1> 56
 2> 64
 3> 34
 4> 51
 5> 52
 6> 78
 7> 62
 8> 51
 9> 47
10> 32

Grade No 1 was 56
Grade No 2 was 64
Grade No 3 was 34
Grade No 4 was 51
Grade No 5 was 52
Grade No 6 was 78
Grade No 7 was 62
Grade No 8 was 51
Grade No 9 was 47
Grade No 10 was 32
Average of the ten numbers entered is: 52.700001
```

How It Works

I'll just explain the new bit where you reuse the elements of the array in a loop:

```
for(int i = 0; i < count; i++)
  printf("\nGrade Number %d was %d", i+1, number[i]);
```

You simply add another `for` loop to step through the elements in the array and output each value. You use the loop control variable to produce the sequence number for the value of the number of the element and to access the corresponding array element. These values obviously correspond to the numbers you typed in. To get the grade numbers starting from 1, you use the expression `i+1` in the output statement so you get grade numbers from 1 to 10 as `i` runs from 0 to 9.

Before you go any further with arrays, you need to look into how your variables are stored in the computer's memory. You also need to understand how an array is different from the variables you've seen up to now.

A Reminder About Memory

Let's quickly recap what you learned about memory in Chapter 2. You can think of the memory of your computer as an ordered line of elements. Each element is in one of two states: either the element is on (let's call this state 1) or the element is off (this is state 0). Each element holds one binary digit and is referred to as a **bit**. Figure 5-2 shows a sequence of bytes in memory.

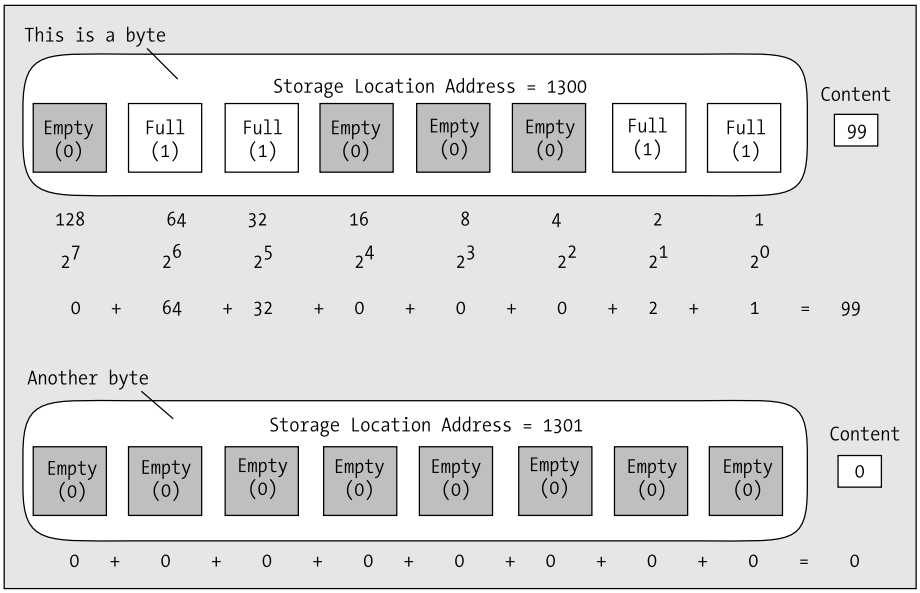


Figure 5-2. *Bytes in memory*

For convenience, the bits in Figure 5-2 are grouped into sets of eight, and a group of eight bits is called a **byte**. To identify each byte so that its contents may be accessed, the byte is labeled with a number starting from 0 for the first byte in memory and going up to whatever number of bytes there are in memory. This label for a byte is called its `address`.

You've already been using the `address of` operator, &, extensively with the `scanf()` function. You've been using this as a prefix to the variable name, because the function needs to store data that is entered from the keyboard into the variable. Just using the variable name by itself as an argument to a function makes the value stored in the variable available to the function. Prefixing the variable name with the `address of` operator and using that as the argument to the function makes the address of the variable available to the function. This enables the function to store information at this address and thus modify the value that's stored in the variable. The best way to get a feel for the `address of` operator is to use it a bit more, so let's do that.

TRY IT OUT: USING THE ADDRESS OF OPERATOR

Each variable that you use in a program takes up a certain amount of memory, measured in bytes, and the exact amount of memory is dependent on the type of the variable. Let's try finding the address of some variables of different types with the following program:

```
/* Program 5.5 Using the & operator */
#include<stdio.h>

int main(void)
{
  /* declare some integer variables */
  long a = 1L;
  long b = 2L;
  long c = 3L;
```

```
  /* declare some floating-point variables */
  double d = 4.0;
  double e = 5.0;
  double f = 6.0;

  printf("A variable of type long occupies %d bytes.", sizeof(long));
  printf("\nHere are the addresses of some variables of type long:");
  printf("\nThe address of a is: %p  The address of b is: %p", &a, &b);
  printf("\nThe address of c is: %p", &c);
  printf("\n\nA variable of type double occupies %d bytes.", sizeof(double));
  printf("\nHere are the addresses of some variables of type double:");
  printf("\nThe address of d is: %p  The address of e is: %p", &d, &e);
  printf("\nThe address of f is: %p\n", &f);
  return 0;
}
```

Output from this program will be something like this:

```
A variable of type long occupies 4 bytes.
Here are the addresses of some variables of type long:
The address of a is: 0064FDF4  The address of b is: 0064FDF0
The address of c is: 0064FDEC

A variable of type double occupies 8 bytes.
Here are the addresses of some variables of type double:
The address of d is: 0064FDE4  The address of e is: 0064FDDC
The address of f is: 0064FDD4
```

The addresses that you get will almost certainly be different from these. What you get will depend on what operating system you're using and what other programs are running at the time. The actual address values are determined by where your program is loaded in memory, and this can differ from one execution to the next.

How It Works

You declare three variables of type `long` and three of type `double`:

```
  /* declare some integer variables */
  long a = 1L;
  long b = 2L;
  long c = 3L;

  /* declare some floating-point variables */
  double d = 4.0;
  double e = 5.0;
  double f = 6.0;
```

Next, you output the size of variables of type `long`, followed by the addresses of the three variables of that type that you created:

```
  printf("A variable of type long occupies %d bytes.", sizeof(long));
  printf("\nHere are the addresses of some variables of type long:");
  printf("\nThe address of a is: %p  The Address of b is: %p", &a, &b);
  printf("\nThe address of c is: %p", &c);
```

The address of operator is the & that precedes the name of each variable. You also used a new format specifier, %p, to output the address of the variables. This format specifier is for outputting a memory address, and the value is presented in hexadecimal format. A memory address is typically 16, 32, or 64 bits, and the size of the address will determine the maximum amount of memory that can be referenced. A memory address on my computer is 32 bits and is presented as eight hexadecimal digits; on your machine it may be different.

You then output the size of variables of type double, followed by the addresses of the three variables of that type that you also created:

```
printf("\n\nA variable of type double occupies %d bytes.", sizeof(double));
printf("\nHere are the addresses of some variables of type double:");
printf("\nThe address of d is: %p  The address of e is: %p", &d, &e);
printf("\nThe address of f is: %p\n", &f);
```

In fact, the interesting part isn't the program itself so much as the output. Look at the addresses that are displayed. You can see that the value of the address gets steadily lower in a regular pattern, as shown in Figure 5-3. On my computer, the address of b is 4 lower than that of a, and c is also lower than b by 4. This is because each variable of type long occupies 4 bytes. There's a similar situation with the variables d, e, and f, except that the difference is 8. This is because 8 bytes are used to store a value of type double.

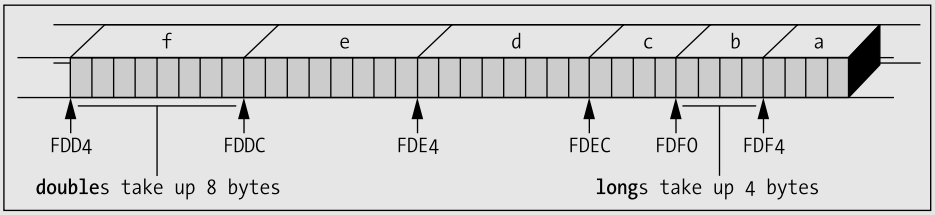


Figure 5-3. *Addresses of variables in memory*

■ **Caution** If the addresses for the variables are separated by greater amounts than the size value, it is most likely because you compiled the program as a **debug** version. In debug mode your compiler may allocate extra space to store additional information about the variable that will be used when you're executing the program in debug mode.

Arrays and Addresses

In the following array, the name number identifies the address of the area of memory where your data is stored, and the specific location of each element is found by combining this with the index value, because the index value represents an offset of a number of elements from the beginning of the array.

```
long number[4];
```

When you declare an array, you give the compiler all the information it needs to allocate the memory for the array. You tell it the type of value, which will determine the number of bytes that each element will require, and how many elements there will be. The array name identifies where in memory the array begins. An index value specifies how many elements from the beginning you have to go to address the element you want. The address of an array element is going to be the address

where the array starts, plus the index value for the element multiplied by the number of bytes required to store each element of the type stored in the array. Figure 5-4 represents the way that array variables are held in memory.

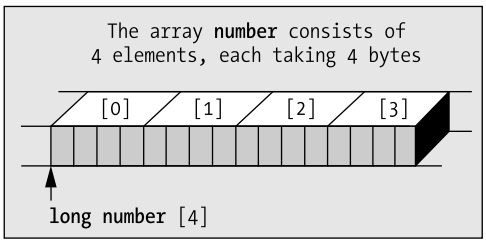


Figure 5-4. *The organization of an array in memory*

You can obtain the address of an array element in a fashion similar to ordinary variables. For an integer variable called `value`, you would use the following statement to print its address:

```
printf("\n%p", &value);
```

To output the address of the third element of an array called `number`, you could write the following:

```
printf("\n%p", &number[2]);
```

Remember that you use the value 2 that appears within the square brackets to reach the third element. Here, you've obtained the address of the element with the `address of` operator. If you used the same statement without the `&`, you would display the actual value stored in the third element of the array, not its address.

I can show this using some working code. The following fragment sets the value of the elements in an array and outputs the address and contents of each element:

```
int data[5];
for(int i = 0 ; i<5 ; i++)
{
  data[i] = 12*(i+1);
  printf("\ndata[%d] Address: %p  Contents: %d", i, &data[i], data[i]);
}
```

The `for` loop variable `i` iterates over all the legal index values for the `data` array. Within the loop, the value of the element at index position `i` is set to `12*(i+1)`. The output statement displays the current element with its index value, the address of the current array element determined by the current value of `i`, and the value stored within the element. If you make this fragment into a program, the output will be the following:

```
data[0] Address: 0x0012ff58  Contents: 12
data[1] Address: 0x0012ff5c  Contents: 24
data[2] Address: 0x0012ff60  Contents: 36
data[3] Address: 0x0012ff64  Contents: 48
data[4] Address: 0x0012ff68  Contents: 60
```

The value of `i` is displayed between the square brackets following the array name. You can see that the address of each element is 4 greater than the previous element so each element occupies 4 bytes.

Initializing an Array

Of course, you may want to assign initial values for the elements of your array, even if it's only for safety's sake. Predetermining initial values in the elements of your array can make it easier to detect when things go wrong. To initialize the elements of an array, you just specify the list of initial values between braces and separated by commas in the declaration. For example

```
double values[5] = { 1.5, 2.5, 3.5, 4.5, 5.5 };
```

declares the array `values` with five elements. The elements are initialized with `values[0]` having the value 1.5, `value[1]` having the initial value 2.5, and so on.

To initialize the whole array, there should be one value for each element. If there are fewer initializing values than elements, the elements without initializing values will be set to 0. Thus, if you write

```
double values[5] = { 1.5, 2.5, 3.5 };
```

the first three elements will be initialized with the values between braces, and the last two elements will be initialized with 0.

If you put more initializing values than there are array elements, you'll get an error message from the compiler. However, you are not obliged to supply the size of the array when you specify a list of initial values. The compiler can deduce the number of elements from the list of values:

```
int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Here the size of the array is determined by the number of initial values in the list so the `primes` array will have ten elements.

Finding the Size of an Array

You've already seen that the `sizeof` operator computes the number of bytes that a variable of a given type occupies. You can apply the `sizeof` operator to a type name like this:

```
printf("\nThe size of a variable of type long is %d bytes.", sizeof(long));
```

The parentheses around the type name following the `sizeof` operator are required. If you leave them out, the code won't compile.

You can also apply the `sizeof` operator to a variable and it will compute the number of bytes occupied by that variable. For example, suppose you declare the variable value with the following statement:

```
double value = 1.0;
```

You can now output the number of bytes occupied by `value` with this statement:

```
printf("\nThe size of value is %d bytes.", sizeof value);
```

Note that no parentheses around the operand for `sizeof` are necessary in this case, but you can include them if you wish. This statement will output the following line:

```
The size of value is 8 bytes.
```

This is because a variable of type `double` occupies 8 bytes in memory. Of course, you can store the size value you get from applying the `sizeof` operator:

```
int value_size = sizeof value;
```

The `sizeof` operator works with arrays too. You can declare an array with the following statement:

```
double values[5] = { 1.5, 2.5, 3.5, 4.5, 5.5 };
```

Now you can output the number of bytes that the array occupies with the following statement:

```
printf("\nThe size of the array, values, is %d bytes.", sizeof values);
```

This will produce the following output:

```
The size of the array, values, is 40 bytes.
```

You can also obtain the number of bytes occupied by a single element of the array with the expression `sizeof values[0]`. This expression will have the value 8. Of course, any legal index value for an element could be used to produce the same result. You can therefore use the `sizeof` operator to calculate the number of elements in an array:

```
int element_count = sizeof values/sizeof values[0];
```

After executing this statement, the variable `element_count` will contain the number of elements in the array `values`.

Because you can apply the `sizeof` operator to a data type, you could have written the previous statement to calculate the number of array elements as follows:

```
int element_count = sizeof values/sizeof(double);
```

This would produce the same result as before because the array is of type `double` and `sizeof(double)` would have produced the number of bytes occupied by a `double` value. Because there is the risk that you might accidentally use the wrong type, it's probably better to use the former statement in practice.

Although the `sizeof` operator doesn't require the use of parentheses when applied to a variable, it's common practice to use them anyway, so the earlier example could be written as follows:

```
int ElementCount = sizeof(values)/sizeof(values[0]);
printf("The size of the array is %d elements ", sizeof(values));
printf("and there are %d elements of %d bytes each",
                                ElementCount, sizeof(values[0]);
```

The output from these statements will be the following:

```
The size of the array is 40 bytes and there are 5 elements of 8 bytes each
```

Multidimensional Arrays

Let's stick to two dimensions for the moment and work our way up. A two-dimensional array can be declared as follows:

```
float carrots[25][50];
```

This declares an array, `carrots`, containing 25 rows of 50 floating-point elements. Similarly, you can declare another two-dimensional array of floating-point numbers with this statement:

```
float numbers[3][5];
```

Like the vegetables in the field, you tend to visualize these arrays as rectangular arrangements because it's convenient to do so. You can visualize this array as having three rows and five columns. They're actually stored in memory sequentially by row, as shown in Figure 5-5.

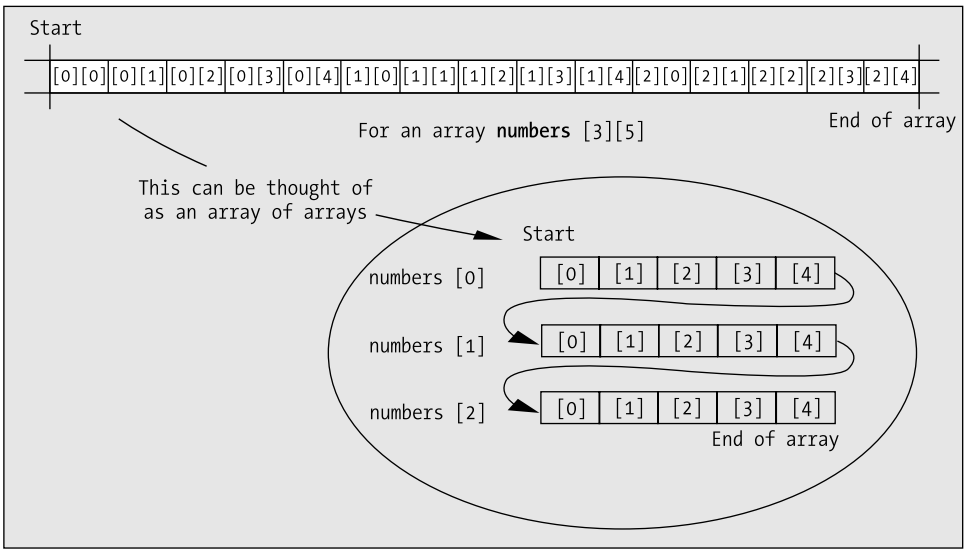


Figure 5-5. *Organization of a 3 × 5 element array in memory*

It's easy to see that the rightmost index varies most rapidly. Figure 5-5 also illustrates how you can envisage a two-dimensional array as a one-dimensional array of elements, in which each element is itself a one-dimensional array. You can view the `numbers` array as a one-dimensional array of three elements, where each element in an array contains five elements of type `float`. The first row of five elements of type `float` is located in memory at an address labeled `numbers[0]`, the next row at `numbers[1]`, and the last row of five elements at `numbers[2]`.

The amount of memory allocated to each element is, of course, dependent on the type of variables that the array contains. An array of type `double` will need more memory to store each element than an array of type `float` or type `int`. Figure 5-6 illustrates how the array `numbers[4][10]` with four rows of ten elements of type `float` is stored.

Because the array elements are of type `float`, which on my machine occupy 4 bytes, the total memory occupied by this array on my computer will be 4 × 10 × 4 bytes, which amounts to a total of 160 bytes.

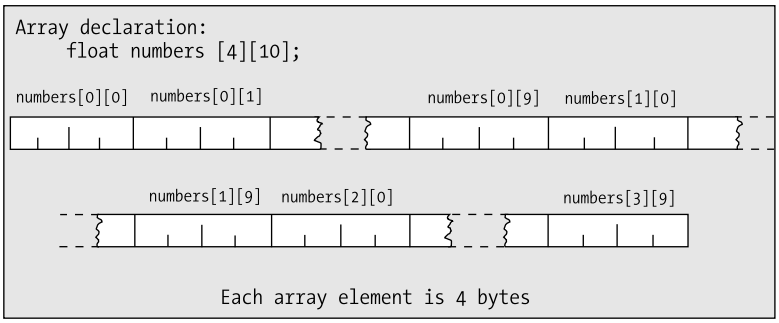


Figure 5-6. *Memory occupied by a 4 × 10 array*

Initializing Multidimensional Arrays

Let's first consider how you initialize a two-dimensional array. The basic structure of the declaration, with initialization, is the same as you've seen before, except that you can optionally put all the initial values for each row between braces {}:

```
int numbers[3][4] = {
                      { 10, 20, 30, 40 },        /* Values for first row  */
                      { 15, 25, 35, 45 },        /* Values for second row */
                      { 47, 48, 49, 50 }         /* Values for third row  */
                    };
```

Each set of values that initializes the elements in a row is between braces, and the whole lot goes between another pair of braces. The values for a row are separated by commas, and each set of values for a row is separated from the next set by a comma.

If you specify fewer initializing values than there are elements in a row, the values will be assigned to elements in sequence, starting with the first in the row. The remaining elements in a row that are left when the initial values have all been assigned will be initialized to 0.

For arrays of three or more dimensions, the process is extended. A three-dimensional array, for example, will have three levels of nested braces, with the inner level containing sets of initializing values for a row:

```
int numbers[2][3][4] = {
                         {                       /* First block of 3 rows  */
                            { 10, 20, 30, 40 },
                            { 15, 25, 35, 45 },
                            { 47, 48, 49, 50 }
                         },
                         {                       /* Second block of 3 rows */
                            { 10, 20, 30, 40 },
                            { 15, 25, 35, 45 },
                            { 47, 48, 49, 50 }
                         }
                       };
```

As you can see, the initializing values are between an outer pair of braces that enclose two blocks of three rows, each between braces. Each row is also between braces, so you have three levels of nested braces for a three-dimensional array. This is true generally; for instance, a six-dimensional array will have six levels of nested braces enclosing the initial values for the elements. You can omit

the braces around the list for each row and the initialization will still work; but including the braces for the row values is much safer because you are much less likely to make a mistake. Of course, if you want to supply fewer initial values than there are elements in a row, you must include the braces around the row values.

TRY IT OUT: MULTIDIMENSIONAL ARRAYS

Let's move away from vegetables and turn to more practical applications. You could use arrays in a program to help you work out your hat size. With this program, just enter the circumference of your head in inches, and your hat size will be displayed:

```
/* Program 5.6 Know your hat size - if you dare...  */
#include <stdio.h>
#include <stdbool.h>

int main(void)

{
   /* The size array stores hat sizes from 6 1/2 to 7 7/8 */
   /* Each row defines one character of a size value so   */
   /* a size is selected by using the same index for each */
   /* the three rows. e.g. Index 2 selects 6 3/4.         */
   char size[3][12] = {                /* Hat sizes as characters */
      {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
      {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
      {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
                     };

  int headsize[12] =                  /* Values in 1/8 inches    */
      {164,166,169,172,175,178,181,184,188,191,194,197};

  float cranium = 0.0;           /* Head circumference in decimal inches */
  int your_head = 0;             /* Headsize in whole eighths            */
  int i = 0;                     /* Loop counter                         */
  bool hat_found = false;        /* Indicates when a hat is found to fit */

  /* Get the circumference of the head */
  printf("\nEnter the circumference of your head above your eyebrows "
       "in inches as a decimal value: ");
  scanf(" %f", &cranium);

  /* Convert to whole eighths of an inch */
  your_head = (int)(8.0*cranium);

  /* Search for a hat size */
  /* A fit is when your_head is greater that one headsize element */
  /* and less than or equal to the next. The size the the second  */
  /* headsize value.                                              */
  for (i = 1 ; i < 12 ; i++)

    /* Find head size in the headsize array */
    if(your_head > headsize[i-1] && your_head <= headsize[i])
```

```
    {
      hat_found = true;
       break;
    }

  if(your_head == headsize[0])  /* Check for min size fit    */
  {
    i = 0;
    hat_found = true;
  }
  if(hat_found)
    printf("\nYour hat size is %c %c%c%c\n",
           size[0][i], size[1][i], (size[1][i]==' ') ? ' ' : '/', size[2][i]);
  /* If no hat was found, the head is too small, or too large    */
  else
  {
    if(your_head < headsize[0])         /* check for too small   */
      printf("\nYou are the proverbial pinhead. No hat for"
                                              " you I'm afraid.\n");
    else                                /* It must be too large  */
      printf("\nYou, in technical parlance, are a fathead."
                                  " No hat for you, I'm afraid.\n");
  }
  return 0;
}
```

Typical output from this program would be this

```
Enter the circumference of your head above your eyebrows in inches as a decimal
 value: 22.5
Your hat size is 7 1/4
```

or possibly this

```
Enter the circumference of your head above your eyebrows in inches as a decimal
 value: 29
You, in technical parlance, are a fathead. No hat for you I'm afraid.
```

How It Works

Before I start discussing this example, I should give you a word of caution. Don't use it to assist large football players to determine their hat size unless they're known for their sense of humor.

The example looks a bit complicated because of the nature of the problem, but it does illustrate using arrays. Let's go through what's happening.

The first declaration in the body of `main()` is as follows:

```
 char size[3][12] = {                /* Hat sizes as characters */
    {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
    {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
    {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
                   };
```

Apart from hats that are designated as "one size fits all" or as small, medium, and large, hats are typically available in sizes from 6 1/2 to 7 7/8 in increments of 1/8. The `size` array shows one way in which you could store such sizes in the program. This array corresponds to 12 possible hat sizes, each of which is made up of three values. For each hat size you store three characters, making it more convenient to output the fractional sizes. The smallest hat size is 6 1/2, so the first three characters corresponding to the first size are in `size[0][0]`, `size[1][0]`, and `size[2][0]`. They contain the characters `'6'`, `'1'`, and `'2'`, representing the size 6 1/2. The biggest hat size is 7 7/8, and it's stored in `size[0][11]`, `size[1][11]`, `size[2][11]`.

You then declare the array `headsize`, which provides the reference head dimensions in this declaration:

```
int headsize[12] =         /* Values in 1/8 inches    */
                  {164,166,169,172,175,178,181,184,188,191,194,197};
```

The values in the array are all whole eighths of an inch. They correspond to the values in the `size` array containing the hat sizes. This means that a head size of 164 eighths of an inch (about 20.5 inches) will give a hat size of 6 1/2, and at the other end of the scale, 197 eighths corresponds to a hat size of 7 7/8.

Notice that the head sizes don't run consecutively. You could get a head size of 171, for example, which doesn't fall into a definite hat size. You need to be aware of this later in the program so that you can decide which is the closest hat size for the head size.

After declaring your arrays, you then declare all the variables you're going to need:

```
float cranium = 0.0;            /* Head circumference in decimal inches */
int your_head = 0;              /* Headsize in whole eighths            */
int i = 0;                      /* Loop counter                         */
bool hat_found = false;         /* Indicates when a hat is found to fit */
```

Notice that `cranium` is declared as type `float`, but the rest are all type `int`. This becomes important later. You declare the variable `hat_found` as type `bool` so you use the symbol `false` to initialize this. The `hat_found` variable will record when you have found a size that fits.

Next, you prompt for your head size to be entered in inches, and the value is stored in the variable `cranium` (remember it's type `float`, so you can store values that aren't whole numbers):

```
printf("\nEnter the circumference of your head above your eyebrows "
    "in inches as a decimal value: ");
scanf(" %f", &cranium);
```

The value stored in `cranium` is then converted into eighths of an inch with this statement:

```
your_head = (int)(8.0*cranium);
```

Because cranium contains the circumference of a head in inches, multiplying by 8.0 results in the number of eighths of an inch that that represents. Thus the value stored in `your_head` will then be in the same units as the values stored in the array `headsize`. Note that you need the cast to type `int` here to avoid a warning message from the compiler. The code will still work if you omit the cast, but the compiler must then insert the cast to type `int`. Because this cast potentially loses information, the compiler will issue a warning. The parentheses around the expression `(8.0*cranium)` are also necessary; without them, you would only cast the value 8.0 to type `int`, not the whole expression.

You use the value stored in `your_head` to find the closest value in the array `headsize` that isn't less than that value:

```
for (i = 1 ; i < 12 ; i++)
  /* Find head size in the headsize array */
  if(your_head > headsize[i-1] && your_head <= headsize[i])
  {
```

```
      hat_found = true;
       break;
    }
```

The process is a simple one and is carried out in this `for` loop. The loop index `i` runs from the second element in the array to the last element. This is because you use `i-1` to index the array in the `if` expression. On each loop iteration, you compare your head size with a pair of successive values stored in the `headsize` array to find the element value that is greater than or equal to your input size with the preceding value less than your input size. The index found will correspond to the hat size that fits.

If your input size corresponds exactly to the size corresponding to the first element in the array, this size will fit but will not be discovered within the loop. You therefore check for this situation with the `if` statement:

```
if(your_head == headsize[0])  /* Check for min size fit    */
{
  i = 0;
  hat_found = true;
}
```

If the size in `your_head` matches that in the first element of the `headsize` array then you have a hat that fits, so you set `i` to 0 and `hat_found` to true.

Next you output the hat size if the value of `hat_found` is true:

```
if(hat_found)
  printf("\nYour hat size is %c %c%c%c\n",
         size[0][i], size[1][i], (size[1][i]==' ') ? ' ' : '/', size[2][i]);
```

As I said, the hat sizes are stored in the array `size` as characters to simplify the outputting of fractions. The `printf()` here uses the conditional operator to decide when to print a blank and when to print a slash (`/`) for the fractional output value. The fifth element of the `headsize` array corresponds to a hat size of exactly 7. You don't want it to print `7 /`; you just want `7`. Therefore, you customize the `printf()` depending on whether the element `size[1][i]` contains `' '`. In this way, you omit the slash for any size where the numerator of the fractional part is a space, so this will still work even if you add new sizes to the array.

Of course, it may be that no hat was found because either the head is too small or too large for the hat sizes available, so the `else` clause for the `if` statement deals with that situation, because the `else` executes if `hat_found` is false:

```
/* If no hat was found, the head is too small, or too large    */
else
{
  if(your_head < headsize[0])         /* check for too small   */
    printf("\nYou are the proverbial pinhead. No hat for"
                                              " you I'm afraid.\n");
  else                                /* It must be too large  */
    printf("\nYou, in technical parlance, are a fathead."
                                " No hat for you, I'm afraid.\n");
}
```

If the value in `your_head` is less than the first `headsize` element, the head is too small for the available hats; otherwise it must be too large.

Remember, when you use this program, if you lie about the size of your head, your hat won't fit. The more mathematically astute, and any hatters reading this book, will appreciate that the hat size is simply the diameter of a notionally circular head. Therefore, if you have the circumference of your head in inches, you can produce your hat size by dividing this value by π.

Designing a Program

Now that you've learned about arrays, let's see how you can apply them in a bigger problem. Let's try writing another game.

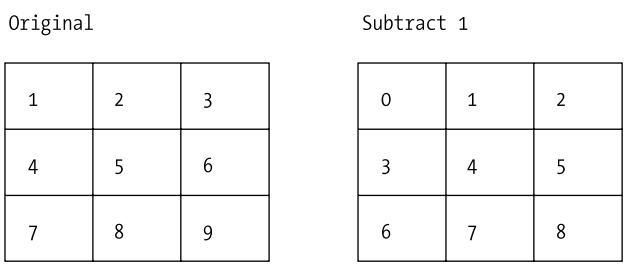
The Problem

The problem you're set is to write a program that allows two people to play tic-tac-toe (also known as noughts and crosses) on the computer.

The Analysis

Tic-tac-toe is played on a 3 × 3 grid of squares. Two players take turns entering either an **X** or an **O** in the grid. The player that first manages to get three of his or her symbols in a line horizontally, vertically, or diagonally is the winner. You know how the game works, but how does that translate into designing your program? You'll need the following:

- *A 3 × 3 grid in which to store the turns of the two players*: That's easy. You can just use a two-dimensional array with three rows of three elements.
- *A way for a square to be selected when a player takes his or her turn*: You can label the nine squares with digits from 1 to 9. A player will just need to enter the number of the square to select it.
- *A way to get the two players to take alternate turns*: You can identify the two players as 1 and 2, with player 1 going first. You can then determine the player number by the number of the turn. On odd-numbered turns it's player 1. On even-numbered turns it's player 2.
- *Some way of specifying where to place the player symbol on the grid and checking to see if it's a valid selection*: A valid selection is a digit from 1 to 9. If you label the first row of squares with 1, 2, and 3, the second row with 4, 5, and 6, and the third row with 7, 8, and 9, you can calculate a row index and a column index from the square number. If you subtract 1 from the player's choice of square number, the square numbers are effectively 0 through 8, as shown in the following image:



Then the expression `choice/3` gives the row number, as you can see here:

Original less 1

0	1	2
3	4	5
6	7	8

Divide by 3

0	0	0
1	1	1
2	2	2

The expression `choice%3` will give the column number:

Original less 1

0	1	2
3	4	5
6	7	8

Remainder after divide by 3

0	1	2
0	1	2
0	1	2

- *A method of finding out if one of the players has won*: After each turn, you'll need to check to see if any row, column, or diagonal in the board grid contains identical symbols. If it does, the last player has won.
- *A way to detect the end of the game*: Because the board has nine squares, a game consists of up to nine turns. The game ends when a winner is discovered, or after nine turns.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

You can first add the code for the main game loop and the code to display the board:

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;                    /* Player number - 1 or 2 */
  int winner = 0;                    /* The winning player     */

  char board[3][3] = {               /* The board */
              {'1','2','3'},         /* Initial values are reference numbers */
              {'4','5','6'},         /* used to select a vacant square for   */
              {'7','8','9'}          /* a turn.                              */
                     };
```

```
  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner                            */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    player = i%2 + 1; /* Select player */

    /* Code to play the game */
  }
  /* Code to output the result */
  return 0;
}
```

Here, you've declared the following variables: `i`, for the loop variable; `player`, which stores the identifier for the current player, 1 or 2; `winner`, which contains the identifier for the winning player; and the array `board`, which is of type `char`. The array is of type `char` because you want to place the symbols `'X'` or `'O'` in the squares. The array is initialized with the characters for the digits that identify the squares. The main game loop continues for as long as the loop condition is `true`. It will be `false` if `winner` contains a value other than 0 (which indicates that a winner has been found) or the loop counter is equal to or greater than 9 (which will be the case when all nine squares on the board have been filled).

When you display the grid in the loop, you use vertical bars and underline characters to delineate the squares. When a player selects a square, the symbol for that player will replace the digit character.

Step 2

Next, you can implement the code for the player to select a square and to ensure that the square is valid:

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;                    /* Player number - 1 or 2           */
  int winner = 0;                    /* The winning player               */
  int choice = 0;                    /* Square selection number for turn */
  int row = 0;                       /* Row index for a square           */
  int column = 0;                    /* Column index for a square        */

  char board[3][3] = {               /* The board */
              {'1','2','3'},         /* Initial values are reference numbers */
              {'4','5','6'},         /* used to select a vacant square for   */
              {'7','8','9'}          /* a turn.                              */
                    };
```

```
  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner                            */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    player = i%2 + 1; /* Select player   */

    /* Get valid player square selection */
    do
    {
      printf("\nPlayer %d, please enter the number of the square "
             "where you want to place your %c: ",
              player,(player==1)?'X':'0');
      scanf("%d", &choice);

      row = --choice/3;                /* Get row index of square    */
      column = choice%3;               /* Get column index of square */
    }while(choice<0 || choice>9 || board[row][column]>'9');

    /* Insert player symbol */
    board[row][column] = (player == 1) ? 'X' : '0';

    /* Code to check for a winner */
  }
  /* Code to output the result */
  return 0;
}
```

You prompt the current player for input in the `do-while` loop and read the square number into the variable `choice` that you declared as type `int`. You'll use this value to compute the row and column index values in the array. The row and column index values are stored in the integer variables `row` and `column`, and you compute these values using the expressions you saw earlier. The `do-while` loop condition verifies that the square selected is valid. There are three possible ways that an invalid choice could be made: the integer entered for the square number could be less than the minimum, 1, or greater than the maximum, 9, or it could select a square that already contains `'X'` or `'0'`. In the latter case, the contents of the square will have a value greater than the character `'9'`, because the character codes for `'X'` and `'0'` are greater than the character code for `'9'`. If the choice that is entered fails on any of these conditions, you just repeat the request to select a square.

Step 3

You can add the code to check for a winning line next. This needs to be executed after every turn:

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;                    /* Player number - 1 or 2              */
  int winner = 0;                    /* The winning player                  */
  int choice = 0;                    /* Square selection number for turn    */
  int row = 0;                       /* Row index for a square              */
  int column = 0;                    /* Column index for a square           */
  int line=0;                        /* Row or column index in checking loop */

  char board[3][3] = {               /* The board */
             {'1','2','3'},          /* Initial values are reference numbers */
             {'4','5','6'},          /* used to select a vacant square for   */
             {'7','8','9'}           /* a turn.                              */
                    };

  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner                            */
  for(int i = 0; i<9 && winner==0; i++)
  {
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    player = i%2 + 1;                /* Select player   */

    /* Get valid player square selection          */
    do
    {
      printf("\nPlayer %d, please enter the number of the square "
             "where you want to place your %c: ",
              player,(player==1)?'X':'O');
      scanf("%d", &choice);

      row = --choice/3;              /* Get row index of square    */
      column = choice%3;             /* Get column index of square */
    }while(choice<0 || choice>9 || board[row][column]>'9');

    /* Insert player symbol */
    board[row][column] = (player == 1) ? 'X' : 'O';

    /* Check for a winning line - diagonals first */
    if((board[0][0]==board[1][1] && board[0][0]==board[2][2]) ||
       (board[0][2]==board[1][1] && board[0][2]==board[2][0]))
      winner = player;
```

```
    else
      /* Check rows and columns for a winning line */
      for(line = 0; line <= 2; line ++)
        if((board[line][0]==board[line][1] && board[line][0]==board[line][2])||
           (board[0][line]==board[1][line] && board[0][line]==board[2][line]))
          winner = player;
  }
  /* Code to output the result */
  return 0;
}
```

To check for a winning line, you can compare one element in the line with the other two to test for equality. If all three are identical, then you have a winning line. You check both diagonals in the board array with the `if` expression, and if either diagonal has identical symbols in all three elements, you set `winner` to the current player. The current player, identified in `player`, must be the winner because he or she was the last to place a symbol on a square. If neither diagonal has identical symbols, you check the rows and the columns in the `else` clause, using a `for` loop. The `for` loop contains one statement, an `if` statement that checks both a row and a column for identical elements. If either is found, `winner` is set to the current player. Each value of the loop variable `line` is used to index a row and a column. Thus the `for` loop will check the row and column corresponding to index value 0, which is the first row and column, then the second row and column, and finally the third row and column corresponding to `line` having the value 2. Of course, if `winner` is set to a value here, the main loop condition will be false, so the loop will end and you'll continue with the code following the main loop.

Step 4

The final task is to display the grid with the final position and to display a message for the result. If `winner` is 0, the game is a draw; otherwise, `winner` contains the player number of the winner:

```
/* Program 5.7 Tic-Tac-Toe */
#include <stdio.h>

int main(void)
{
  int player = 0;                    /* Player number - 1 or 2              */
  int winner = 0;                    /* The winning player                  */
  int choice = 0;                    /* Square selection number for turn    */
  int row = 0;                       /* Row index for a square              */
  int column = 0;                    /* Column index for a square           */
  int line=0;                        /* Row or column index in checking loop */

  char board[3][3] = {               /* The board */
              {'1','2','3'},         /* Initial values are reference numbers */
              {'4','5','6'},         /* used to select a vacant square for   */
              {'7','8','9'}          /* a turn.                              */
                     };

  /* The main game loop. The game continues for up to 9 turns */
  /* As long as there is no winner                            */
  for(int i = 0; i<9 && winner==0; i++)
  {
```

```
    /* Display the board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
     printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

    player = i%2 + 1;                /* Select player */

    /* Get valid player square selection */
    do
    {
      printf("\nPlayer %d, please enter the number of the square "
             "where you want to place your %c: ",
              player,(player==1)?'X':'O');
      scanf("%d", &choice);

      row = --choice/3;               /* Get row index of square    */
      column = choice%3;              /* Get column index of square */
    }while(choice<0 || choice>9 || board[row][column]>'9');

    /* Insert player symbol */
    board[row][column] = (player == 1) ? 'X' : 'O';

    /* Check for a winning line - diagonals first */
    if((board[0][0]==board[1][1] && board[0][0]==board[2][2]) ||
       (board[0][2]==board[1][1] && board[0][2]==board[2][0]))
      winner = player;
    else
      /* Check rows and columns for a winning line */
      for(line = 0; line <= 2; line ++)
        if((board[line][0]==board[line][1] &&
            board[line][0]==board[line][2])||
           (board[0][line]==board[1][line] &&
            board[0][line]==board[2][line]))
          winner = player;
  }
  /* Game is over so display the final board */
  printf("\n\n");
  printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
  printf("---+---+---\n");
  printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
  printf("---+---+---\n");
  printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);

  /* Display result message */
  if(winner == 0)
    printf("\nHow boring, it is a draw\n");
  else
    printf("\nCongratulations, player %d, YOU ARE THE WINNER!\n",
                                                          winner);
  return 0;
}
```

Typical output from this program and a very bad player No. 2 would be as follows:

```
 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

Player 1, please enter your go: 1

 X | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

Player 2, please enter your go: 2

 X | O | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

Player 1, please enter your go: 5


 X | O | 3
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | 9

Player 2, please enter your go: 3


 X | O | O
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | 9

Player 1, please enter your go: 9

 X | O | O
---+---+---
 4 | X | 6
---+---+---
 7 | 8 | X

Congratulations, player 1, YOU ARE THE WINNER!
```

Summary

This chapter explored the ideas behind arrays. An array is a fixed number of elements of the same type and you access any element within the array using the array name and one or more index values. Index values for an array are integer values starting from 0, and there is one index for each array dimension.

Combining arrays with loops provides a very powerful programming capability. Using an array, you can process a large number of data values of the same type within a loop, so the amount of program code you need for the operation is essentially the same, regardless of how many data values there are. You have also seen how you can organize your data using multidimensional arrays. You can structure an array such that each array dimension selects a set of elements with a particular characteristic, such as the data pertaining to a particular time or location. By applying nested loops to multidimensional arrays, you can process all the array elements with a very small amount of code.

Up until now, you've mainly concentrated on processing numbers. The examples haven't really dealt with text to any great extent. You're going to change that in the next chapter, where you're going to write programs that can process and analyze strings of characters.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Downloads area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 5-1. Write a program that will read five values of type `double` from the keyboard and store them in an array. Calculate the reciprocal of each value (the reciprocal of a value `x` is `1.0/x`) and store it in a separate array. Output the values of the reciprocals, and calculate and output the sum of the reciprocals.

Exercise 5-2. Define an array, `data`, with 100 elements of type `double`. Write a loop that will store the following sequence of values in corresponding elements of the array:

```
1/(2*3*4) 1/(4*5*6) 1/(6*7*8) … up to 1/(200*201*202)
```

Write another loop that will calculate the following:

```
data[0]-data[1]+data[2]-data[3]+… -data[99]
```

Multiply the result of this by 4.0, add 3.0, and output the final result. Do you recognize the value that you get?

Exercise 5-3. Write a program that will read five values from the keyboard and store them in an array of type `float` with the name `amounts`. Create two arrays of five elements of type `long` with the names `dollars` and `cents`. Store the whole number part of each value in the `amounts` array in the corresponding element of `dollars` and the fractional part of the amount as a two-digit integer in `cents` (e.g., 2.75 in `amounts[1]` would result in 2 being stored in `dollars[1]` and 75 being stored in `cents[1]`). Output the values from the two arrays of type `long` as monetary amounts (e.g., $2.75).

Exercise 5-4. Define a two-dimensional array, `data[12][5]`, of type `double`. Initialize the elements in the first column with values from 2.0 to 3.0 inclusive in steps of 0.1. If the first element in a row has the value `x`, populate the remaining elements in each row with the values `1/x`, x^2, x^3, and x^4. Output the values in the array with each row on a separate line and with a heading for each column.