CHAPTER 7

# ARRAYS

The C language provides a capability that enables the user to define a set of ordered data items known as an *array*. This chapter describes how arrays can be defined and manipulated in C. In later chapters, we will include further discussions on arrays to illustrate how they work with program functions, structures, character strings, and pointers.

Suppose we had a set of grades that we wished to read into the computer, and suppose that we wished to perform some operations on these grades, such as rank them in ascending order, compute their average, or find their median. In Program 6-2, we were able to calculate the average of a set of grades by simply adding each grade into a cumulative total as each grade was keyed in. However, if we wanted to rank the grades into ascending order, for example, then we would have to do something further. If you think about the process of ranking a set of grades, you will quickly realize that we cannot perform such an operation until each and every grade has been entered. Therefore, using the techniques we have already described, we would read in each grade and store it into a unique variable, perhaps with a sequence of statements such as:

```
printf ("Enter grade 1\n");
scanf ("%d", &grade1);
printf ("Enter grade 2\n");
scanf ("%d", &grade2);
```

Once all of the grades had been entered, we could then proceed to rank them. This could be done by setting up a series of **if** statements to compare each of the values to determine the smallest grade, the next smallest grade, and so on, until the maximum grade had been determined. If you sit down and try to write a program to perform precisely this task, you will soon realize that for any reasonably sized list of grades (where reasonably sized is probably only about 10), the resulting program will be quite large and quite complex. All is not lost, however, as this is one instance when the array comes to the rescue.

In C we can define a variable called **grades**, which represents not a *single* value of a grade but an entire *set of grades*. Each element of the set can then be referenced by means of a number called an *index* number or *subscript*. Where, in mathematics, a subscripted variable $x_i$ refers to the $i$th element $x$ in a set, in C the equivalent notation is

```
x[i]
```

- make a program that will store
- every student have # and use only this number

So the expression

```
grades[5]
```

(read as "**grades** sub 5") refers to element number 5 in the array called **grades**. In C, array elements begin with number 0, so

```
grades[0]
```

actually refers to the first element of the array. (For this reason, it is easier to think of it as referring to element number zero, rather than as referring to the first element.)

An individual array element can be used anywhere that a normal variable could be. For example, we can assign an array value to another variable with a statement such as

```
g = grades[50];
```

This statement takes the value contained in **grades[50]** and assigns it to **g**. More generally, if **i** is declared to be an integer variable, then the statement

```
g = grades[i];
```

will take the value contained in element number **i** of the **grades** array and assign it to **g**. So if **i** were equal to 7 and the above statement were executed, then the value of **grades[7]** would get assigned to **g**.

A value can be stored into an element of an array simply by specifying the array element on the left-hand side of an equals sign. In the statement

```
grades[100] = 95;
```

the value 95 is stored into element number 100 of the **grades** array. The statement

```
grades[i] = g;
```

will have the effect of storing the value of **g** into **grades[i]**.

The ability to represent a collection of related data items by a single array enables us to develop concise and efficient programs. For example, we can very easily sequence through the elements in the array by varying the value of a variable that is used as a subscript into the array. So the **for** loop

```
for ( i = 0;  i < 100;  ++i )
  sum = sum + grades[i];
```

will sequence through the first 100 elements of the array **grades** (elements 0 through 99) and will add the value of each grade into **sum**. When the **for** loop is finished, the variable **sum** will then contain the total of the first 100 values of the **grades** array (assuming **sum** were set to 0 before the loop was entered).

In addition to integer constants, integer-valued expressions can also be used inside the brackets to reference a particular element of an array. So if **low** and **high** were defined as integer variables, then the statement

```
next_value = sorted_data[(low + high) / 2];
```

would assign to the variable **next_value** the value indexed by evaluating the expression **(low + high) / 2**. If **low** were equal to 1 and **high** were equal to 9, then the value of **sorted_data[5]** would be assigned to **next_value**. And if **low** were equal to 1 and **high** were equal to 10 then the value of **sorted_data[5]** would also be referenced, since we know that an integer division of 11 by 2 gives the result of 5.

Just as with variables, arrays must also be declared before they are used. The declaration of an array involves declaring the type of element that will be contained in the array—such as **int**, **float**, or **char**—as well as the maximum number of elements that will be stored inside the array. (The C system needs this latter information in order to determine how much of its memory space to reserve for the particular array.)

As an example, the declaration

```
int  grades[100];
```

declares **grades** to be an array containing 100 integer elements. Valid references to this array may be made by using subscripts from 0 through 99. (But be careful to make sure that valid subscripts are used, since C does not do any checking of array bounds for you. So a reference to element number 150 of array **grades** as declared above would not necessarily cause an error but would most likely cause unwanted, if not unpredictable, program results.)

To declare an array called **averages** that contained 200 floating point elements, the declaration

```
float  averages[200];
```

would be used. This declaration would cause enough space inside the computer's memory to be reserved to contain 200 floating point numbers. Similarly, the declaration

```
int  values[10];
```

would reserve enough space for an array called **values** that could hold up to 10 integer numbers. We could better conceptualize this reserved storage space by referring to Fig. 7-1.

The elements of arrays declared to be of type **int**, **float**, or **char** may be manipulated in the same fashion as can ordinary variables: we can assign values to them, display their values, add to them, subtract from them, and so on. So if the following statements were to appear in a program

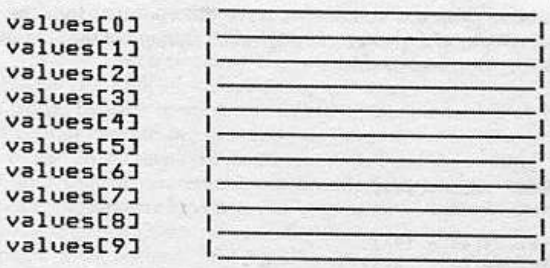


*Fig. 7-1.* The array **values** inside memory.

```
int  values[10];

values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

then the array **values** would contain the numbers as shown in Fig. 7-2 after these statements were executed.

The first assignment statement has the effect of storing the value of 197 into **values[0]**. In a similar fashion, the second and third assignment statements store values of −100 and 350 into **values[2]** and **values[5]**, respectively. The next statement adds the contents of **values[0]** (which is 197) to the contents of **values[5]** (which is 350) and stores the result of 547 in **values[3]**. In the following program statement, 350—the value contained in **values[5]**—is divided by 10 and the result stored into **values[9]**. The last statement decrements the contents of **values[2]**, which has the effect of changing its value from −100 to −101.

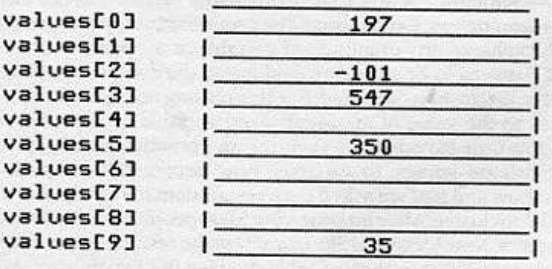


*Fig. 7-2.* The array **values** inside memory.

The above program statements were incorporated into the following program. The **for** loop sequences through each element of the array, displaying its value at the terminal in turn.

**Program 7-1**

```
main ()
  {
    int  values[10];
    int  index;

    values[0] = 197;
    values[2] = -100;
    values[5] = 350;
    values[3] = values[0] + values[5];
    values[9] = values[5] / 10;
    --values[2];

    for ( index = 0;  index < 10;  ++index )
      printf ("values[%d] = %d\n", index, values[index]);
  }
```

**Program 7-1 Output**

```
values[0] = 197
values[1] = 0
values[2] = -101
values[3] = 547
values[4] = 0
values[5] = 350
values[6] = 0
values[7] = 0
values[8] = 0
values[9] = 35
```

The variable **index** assumes the values 0 through 9, since the last valid subscript of an array is always one less than the number of elements (due to that zeroeth element). Since we never assigned values to five of the elements in the array—elements 1, 4 and 6 through 8—the values that are displayed for them are meaningless. Even though the program's output shows these values as zero, the value of any uninitialized variable or array element is simply the value that happens to be sitting around inside the computer's memory at the time that the program is executed. For this reason, no assumption should ever be made as to the value of an uninitialized variable or array element.

It is now time to consider a slightly more practical example. Suppose we took a telephone survey to discover how people felt about a particular television show and that we asked each respondent to rate the show on a scale from 1 to 10, inclusive. After interviewing 5,000 people we accumulated a list of 5,000 numbers. Now we would like to analyze the results. One of the first pieces of data we would like to gather is a table showing the distribution of the ratings. In other words, we would like to know how many people rated the show a 1, how many a 2, and so on up to 10.