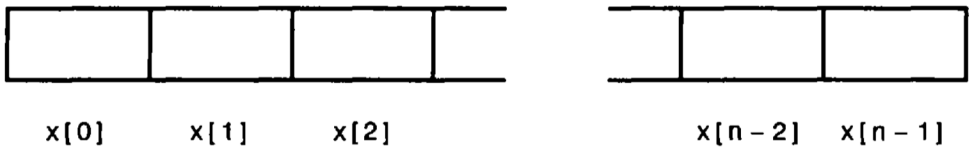
Chapter 9

Arrays

Many applications require the processing of multiple data items that have common characteristics (e.g., a set of numerical data, represented by $x_1, x_2, \ldots, x_n$). In such situations it is often convenient to place the data items into an *array*, where they will all share the same name (e.g., x). The individual data items can be characters, integers, floating-point numbers, etc. However, they must all be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more *subscripts*, with each subscript enclosed in square brackets. Each subscript must be expressed as a nonnegative integer. In an n-element array, the array elements are `x[0]`, `x[1]`, `x[2]`, . . . , `x[n - 1]`, as illustrated in Fig. 9.1. The value of each subscript can be expressed as an integer constant, an integer variable or a more complex integer expression.



x is an n-element, one-dimensional array

Fig. 9.1

The number of subscripts determines the dimensionality of the array. For example, `x[i]` refers to an element in the one-dimensional array x. Similarly, `y[i][j]` refers to an element in the two-dimensional array y. (We can think of a two-dimensional array as a table, where `y[i][j]` is the jth element of the ith row.) Higher-dimensional arrays can be also be formed, by adding additional subscripts in the same manner (e.g., `z[i][j][k]`).

Recall that we have used one-dimensional character arrays earlier in this book, in conjunction with the processing of strings and lines of text. Thus, arrays are not entirely new, even though our previous references to them were somewhat casual. We will now consider arrays in greater detail. In particular, we will discuss the manner in which arrays are defined and processed, the passing of arrays to functions, and the use of multidimensional arrays. Both numerical and character-type arrays will be considered. Initially we will concentrate on one-dimensional arrays, though multidimensional arrays will be considered in Sec. 9.4.

9.1 DEFINING AN ARRAY

Arrays are defined in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression, enclosed in square brackets. The expression is usually written as a positive integer constant.

In general terms, a one-dimensional array definition may be expressed as

storage-class data-type array`[`*expression*`];`

where *storage-class* refers to the storage class of the array, *data-type* is the data type, *array* is the array name, and *expression* is a positive-valued integer expression which indicates the number of array elements. The *storage-class* is optional; default values are `automatic` for arrays that are defined within a function or a block, and `external` for arrays that are defined outside of a function.

EXAMPLE 9.1 Several typical one-dimensional array definitions are shown below.

```
int x[100];
char text[80];
static char message[25];
static float n[12];
```

The first line states that `x` is a 100-element integer array, and the second defines `text` to be an 80-element character array. In the third line, `message` is defined as a static 25-element character array, whereas the fourth line establishes `n` as a static 12-element floating-point array.

It is sometimes convenient to define an array size in terms of a symbolic constant rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size (e.g., within `for` loops as well as in array definitions) can be altered simply by changing the value of the symbolic constant.

EXAMPLE 9.2 Lowercase to Uppercase Text Conversion Here is a complete program that reads in a one-dimensional character array, converts all of the elements to uppercase, and then displays the converted array. Similar programs are shown in Examples 4.4, 6.9, 6.12 and 6.16.

```
/* read in a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define  SIZE  80

main()
{
    char letter[SIZE];
    int count;

    /* read in the line */

    for (count = 0; count < SIZE; ++count)
        letter[count] = getchar();

    /* display the line in upper case */

    for (count = 0; count < SIZE; ++count)
        putchar(toupper(letter[count]));
}
```

Notice that the symbolic constant `SIZE` is assigned a value of 80. This symbolic constant, rather than its value, appears in the array definition and in the two `for` statements. (Remember that *the value of the symbolic constant will be substituted for the constant itself during the compilation process.*) Therefore, in order to alter the program to accommodate a different size array, only the `#define` statement must be changed.

For example, to alter the above program so that it will process a 60-element array, the original `#define` statement is simply replaced by

```
#define  SIZE  60
```

This one change accommodates all of the necessary program alterations; there is no possibility that some required program modification will be overlooked.

Automatic arrays, unlike automatic variables, cannot be initialized. However, external and static array definitions can include the assignment of initial values if desired. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas. The general form is

```
storage-class  data-type  array[expression] = {value 1, value 2, . . ., value n};
```

where *value 1* refers to the value of the first array element, *value 2* refers to the value of the second element, and so on. The appearance of the *expression*, which indicates the number of array elements, is optional when initial values are present.

EXAMPLE 9.3 Shown below are several array definitions that include the assignment of initial values.

```
int digits[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
static float x[6] = {0, 0.25, 0, -0.50, 0, 0};
char color[3] = {'R', 'E', 'D'};
```

Note that x is a static array. The other two arrays (`digits` and `color`) are assumed to be external arrays by virtue of their placement within the program.

The results of these initial assignments, in terms of the individual array elements, are as follows. (Remember that the subscripts in an n-element array range from 0 to n – 1.)

```
digits[0] = 1        x[0] =  0        color[0] = 'R'
digits[1] = 2        x[1] =  0.25     color[1] = 'E'
digits[2] = 3        x[2] =  0        color[2] = 'D'
digits[3] = 4        x[3] = -0.50
digits[4] = 5        x[4] =  0
digits[5] = 6        x[5] =  0
digits[6] = 7
digits[7] = 8
digits[8] = 9
digits[9] = 10
```

All individual array elements that are not assigned explicit initial values will automatically be set to zero. This includes the remaining elements of an array in which some elements have been assigned nonzero values.

EXAMPLE 9.4 Consider the following array definitions.

```
int digits[10] = {3, 3, 3};
static float x[6] = {-0.3, 0 , 0.25};
```

The results, on an element-by-element basis, are as follows.

```
digits[0] = 3        x[0] = -0.3
digits[1] = 3        x[1] =  0
digits[2] = 3        x[2] =  0.25
digits[3] = 0        x[3] =  0
digits[4] = 0        x[4] =  0
digits[5] = 0        x[5] =  0
digits[6] = 0
digits[7] = 0
digits[8] = 0
digits[9] = 0
```

In each case, all of the array elements are automatically set to zero except those that have been explicitly initialized within the array definitions. Note that the repeated values (i.e., 3, 3, 3) must be shown individually.

The array size need not be specified explicitly when initial values are included as a part of an array definition. With a numerical array, the array size will automatically be set equal to the number of initial values included within the definition.

EXAMPLE 9.5 Consider the following array definitions, which are variations of the definitions shown in Examples 9.3 and 9.4.

```
int digits[] = {1, 2, 3, 4, 5, 6};
static float x[] = {0, 0.25, 0, -0.5};
```

Thus, `digits` will be a six-element integer array, and `x` will be a static, four-element floating-point array. The individual elements will be assigned the following values. (*Note the empty brackets in the array declarations.*)

```
digits[0] = 1          x[0] =  0
digits[1] = 2          x[1] =  0.25
digits[2] = 3          x[2] =  0
digits[3] = 4          x[3] = -0.5
digits[4] = 5
digits[5] = 6
```

Strings (i.e., character arrays) are handled somewhat differently, as discussed in Sec. 2.6. In particular, when a string constant is assigned to an external or a static character array as a part of the array definition, the array size specification is usually omitted. The proper array size will be assigned automatically. This will include a provision for the null character `\0`, which is automatically added at the end of every string (see Example 2.26).

EXAMPLE 9.6 Consider the following two character array definitions. Each includes the initial assignment of the string constant `"RED"`. However, the first array is defined as a three-element array, whereas the size of the second array is unspecified.

```
char color[3] = "RED";
char color[] = "RED";
```

The results of these initial assignments are not the same because of the null character, `\0`, which is automatically added at the end of the second string. Thus, the elements of the first array are

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

whereas the elements of the second array are

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```

Thus, the first form is incorrect, since the null character `\0` is not included in the array.

The array definition could also have been written as

```
char color[4] = "RED";
```

This definition is correct, since we are now defining a four-element array which includes an element for the null character. However, many programmers prefer the earlier form, which omits the size specifier.

If a program requires a one-dimensional array *declaration* (because the array is defined elsewhere in the program), the declaration is written in the same manner as the array definition with the following exceptions.

1. The square brackets may be empty, since the array size will have been specified as a part of the array definition. Array declarations are customarily written in this form.
2. Initial values cannot be included in the declaration.

These rules apply to formal argument declarations within functions as well as external variable declarations. However, the rules for defining *multidimensional* formal arguments are more complex (see Sec. 9.4).

EXAMPLE 9.7 Here is a skeletal outline of a two-file C program that makes use of external arrays.

First file:

```
int c[] = {1, 2, 3};              /* external array DEFINITION */

char message[] = "Hello!";        /* external array DEFINITION */

extern void funct1(void);         /* function prototype */

main()
{
    . . . . .
}
```

Second file:

```
extern int c[];                   /* external array DECLARATION */

extern char message[];            /* external array DECLARATION */

extern void funct1(void)          /* function definition */
{
    . . . . .
}
```

This program outline includes two external arrays, `c` and `message`. The first array (`c`) is a three-element integer array that is defined and initialized in the first file. The second array (`message`) is a character array that is also defined and initialized in the first file. The arrays are then *declared* in the second file, because they are global arrays that must be recognized throughout the entire program.

Neither the array definitions in the first file nor the array declarations in the second file include explicit size specifications. Such size specifications are permissible in the first file, but are omitted because of the initialization. Moreover, array size specifications serve no useful purpose within the second file, since the array sizes have already been established.

9.2 PROCESSING AN ARRAY

Single operations which involve entire arrays are not permitted in C. Thus, if `a` and `b` are similar arrays (i.e., same data type, same dimensionality and same size), assignment operations, comparison operations, etc. must be carried out on an element-by-element basis. This is usually accomplished within a loop, where each pass through the loop is used to process one array element. The number of passes through the loop will therefore equal the number of array elements to be processed.

We have already seen several examples in which the individual elements of a character array are processed in one way or another (see Examples 4.4, 4.19, 6.9, 6.12, 6.16, 6.19, 6.20, 6.32, 6.34, 8.3, 8.5 and 9.2). Numerical arrays are processed in much the same manner. In a numerical array, each array element represents a single numerical quantity, as illustrated in the example below.

EXAMPLE 9.8 Deviations About an Average Suppose we want to read a list of n floating-point quantities and then calculate their average, as in Example 6.17. In addition to simply calculating the average, however, we will also compute the *deviation* of each numerical quantity about the average, using the formula

$$d = x_i - avg$$

where x_i represents each of the given quantities, $i = 1, 2, \cdots, n$, and avg represents the calculated average.

In order to solve this problem we must store each of the given quantities in a one-dimensional, floating-point array. This is an essential part of the program. The reason, which must be clearly understood, is as follows.

In all of the earlier examples where we calculated the average of a list of numbers, each number was replaced by its successor in the given list (see Examples 6.10, 6.13, 6.17 and 6.31). Hence each individual number was no longer available for subsequent calculations once the next number had been entered. Now, however, these individual quantities must be retained within the computer in order to calculate their corresponding deviations after the average has been determined. We therefore store them in a one-dimensional array, which we shall call `list`.

Let us define `list` to be a 100-element, floating-point array. However, we need not make use of all 100 elements. Rather, we shall specify the actual number of elements by entering a positive integer quantity (not exceeding 100) for the integer variable n.

Here is the complete C program.

```
/* calculate the average of n numbers,
   then compute the deviation of each number about the average */

#include <stdio.h>

main()

{
    int n, count;
    float avg, d, sum = 0;
    float list[100];

    /* read a value for n */
    printf("\nHow many numbers will be averaged? ");
    scanf("%d", &n);
    printf("\n");

    /* read the numbers and calculate their sum */
    for (count = 0; count < n; ++count)   {
        printf("i = %d   x = ", count + 1);
        scanf("%f", &list[count]);
        sum += list[count];
    }

    /* calculate and display the average */
    avg = sum / n;
    printf("\nThe average is %5.2f\n\n", avg);
```

```
    /* calculate and display the deviations about the average */
    for (count = 0; count < n; ++count)   {
        d = list[count] - avg;
        printf("i = %d   x = %5.2f   d = %5.2f\n", count + 1, list[count], d);
    }
}
```

Note that the second `scanf` function (within the first `for` loop) includes an ampersand (`&`) in front of `list[count]`, since we are entering a single array element rather than an entire array (see Sec. 4.4).

Now suppose the program is executed using the following five numerical quantities: $x_1 = 3, x_2 = -2, x_3 = 12, x_4 = 4.4, x_5 = 3.5$. The interactive session, including the data entry and the calculated results, is shown below. The user's responses are underlined.

```
How many numbers will be averaged? 5
i = 1   x = 3
i = 2   x = -2
i = 3   x = 12
i = 4   x = 4.4
i = 5   x = 3.5

The average is  4.18

i = 1   x =  3.00   d = -1.18
i = 2   x = -2.00   d = -6.18
i = 3   x = 12.00   d =  7.82
i = 4   x =  4.40   d =  0.22
i = 5   x =  3.50   d = -0.68
```

In some applications it may be desirable to assign initial values to the elements of an array. This requires that the array either be defined globally, or locally (within the function) as a static array. The next example illustrates the use of a global array definition.

EXAMPLE 9.9 Deviations About an Average Revisited Let us again calculate the average of a given set of numbers and then compute the deviation of each number about the average, as in Example 9.8. Now, however, let us assign the given numbers to the array within the array definition. To do so, let us move the definition of the array `list` outside of the `main` portion of the program. Thus, `list` will become an external array. Moreover, we will remove the explicit size specification from the definition, since the number of initial values will now determine the array size.

The initial values included in the following program are the same five values that were specified as input data for the previous example. To be consistent, we will also assign an initial value for `n`. This can be accomplished by defining `n` as either an automatic variable within `main`, or as an external variable. We have chosen the latter method, so that all of the initial assignments that might otherwise be entered as input data are grouped together.

Here is the complete program.

```
/* calculate the average of n numbers,
   then compute the deviation of each number about the average */

#include <stdio.h>

int n = 5;
float list[] = {3, -2, 12, 4.4, 3.5};

main()

{
    int count;
    float avg, d, sum = 0;
```

```
    /* calculate and display the average */
    for (count = 0; count < n; ++count)
        sum += list[count];
    avg = sum / n;
    printf("\nThe average is %5.2f\n\n", avg);

    /* calculate and display the deviations about the average */
    for (count = 0; count < n; ++count)   {
        d = list[count] - avg;
        printf("i = %d x = %5.2f   d = %5.2f\n", count + 1, list[count], d);
    }
}
```

Note that this version of the program does not require any input data.

Execution of this program will generate the following output.

```
The average is  4.18

i = 1   x =  3.00   d = -1.18
i = 2   x = -2.00   d = -6.18
i = 3   x = 12.00   d =  7.82
i = 4   x =  4.40   d =  0.22
i = 5   x =  3.50   d = -0.68
```

9.3 PASSING ARRAYS TO FUNCTIONS

An entire array can be passed to a function as an argument. The manner in which the array is passed differs markedly, however, from that of an ordinary variable.

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

Some care is required when writing function prototypes that include array arguments. An empty pair of square brackets must follow the name of each array argument, thus indicating that the argument is an array. If argument names are not included in a function declaration, then an empty pair of square brackets must follow the array argument data type.

EXAMPLE 9.10 The following program outline illustrates the passing of an array from the main portion of the program to a function.

```
float average(int a, float x[]);      /* function prototype */

main()
{
    int n;                            /* variable DECLARATION */
    float avg;                        /* variable DECLARATION */
    float list[100];                  /* array DEFINITION */

    . . . . .

    avg = average(n, list);

    . . . . .
}
```

```
float average(int a, float x[])      /* function DEFINITION */
{
    . . . . .
}
```

Within `main` we see a call to the function `average`. This function call contains two actual arguments — the integer variable `n`, and the one-dimensional, floating-point array `list`. Notice that `list` appears as an ordinary variable within the function call; i.e., the square brackets are not included.

The first line of the function definition includes two formal arguments, `a` and `x`. The formal argument declarations establish `a` as an integer variable and `x` as a one-dimensional, floating-point array. Thus, there is a correspondence between the actual argument `n` and the formal argument `a`. Similarly, there is a correspondence between the actual argument `list` and the formal argument `x`. Note that the size of `x` is not specified within the formal argument declaration.

Note that the function prototype could have been written without argument names, as

```
float average(int, float[]);         /* function declaration */
```

Either form is valid.

We have already discussed the fact that arguments are passed to a function by value when the arguments are ordinary variables (see Sec. 7.5). When an array is passed to a function, however, the values of the array elements *are not* passed to the function. Rather, the array name is interpreted as the *address* of the first array element (i.e., the address of the memory location containing the first array element). This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a *pointer* to the first array element (more about this in the next chapter). Arguments that are passed in this manner are said to be passed *by reference* rather than by value.

When a reference is made to an array element within the function, the value of the element's subscript is added to the value of the pointer to indicate the address of the specified array element. Therefore any array element can be accessed from within the function. Moreover, *if an array element is altered within the function, the alteration will be recognized in the calling portion of the program* (actually, throughout the entire scope of the array).

EXAMPLE 9.11 Here is a simple C program that passes a three-element integer array to a function, where the array elements are altered. The values of the array elements are displayed at three different places in the program, thus illustrating the effects of the alterations.

```
#include <stdio.h>

void modify(int a[]);        /* function prototype */

main()
{
    int count, a[3];         /* array definition */

    printf("\nFrom main, before calling the function:\n");
    for (count = 0; count <= 2; ++count)   {
        a[count] = count + 1;
        printf("a[%d] = %d\n", count, a[count]);
    }

    modify(a);

    printf("\nFrom main, after calling the function:\n");
    for (count = 0; count <=2; ++count)
        printf("a[%d] = %d\n", count, a[count]);
}
```

```
void modify(int a[])          /* function definition */
{
    int count;

    printf("\nFrom the function, after modifying the values:\n");
    for (count = 0; count <= 2; ++count)   {
        a[count] = -9;
        printf("a[%d] = %d\n", count, a[count]);
    }
    return;
}
```

The array elements are assigned the values `a[0] = 1`, `a[1] = 2` and `a[2] = 3` within the first loop appearing in `main`. These values are displayed as soon as they are assigned. The array is then passed to the function `modify`, where each array element is assigned the value –9. These new values are then displayed from within the function. Finally, the values of the array elements are again displayed from `main`, after control has been transferred back to `main` from `modify`.

When the program is executed, the following output is generated.

```
From main, before calling the function:
a[0] = 1
a[1] = 2
a[2] = 3

From the function, after modifying the values:
a[0] = -9
a[1] = -9
a[2] = -9

From main, after calling the function:
a[0] = -9
a[1] = -9
a[2] = -9
```

These results show that the elements of `a` are altered within `main` as a result of the changes that were made within `modify`.

EXAMPLE 9.12 We now consider a variation of the previous program. The present program includes the use of a global variable, and the transfer of both a local variable and an array to the function.

```
#include <stdio.h>

int a = 1;                          /* global variable */
void modify(int b, int c[]);        /* function prototype */

main()
{
    int b = 2;              /* local variable */
    int count, c[3];        /* array definition */

    printf("\nFrom main, before calling the function:\n");
    printf("a = %d   b = %d\n", a, b);
    for (count = 0; count <= 2; ++count)   {
        c[count] = 10 * (count + 1);
        printf("c[%d] = %d\n", count, c[count]);
    }
```

```
    modify(b, c);            /* function access */
    printf("\nFrom main, after calling the function:\n");
    printf("a = %d   b = %d\n", a, b);
    for (count = 0; count <=2; ++count)
        printf("c[%d] = %d\n", count, c[count]);
}

void modify(int b, int c[])         /* function definition */
{
    int count;

    printf("\nFrom the function, after modifying the values:\n");
    a = -999;
    b = -999;
    printf("a = %d   b = %d\n", a, b);
    for (count = 0; count <= 2; ++count)   {
        c[count] = -9;
        printf("c[%d] = %d\n", count, c[count]);
    }
    return;
}
```

When the program is executed, the following output is generated.

```
From main, before calling the function:
a = 1   b = 2
c[0] = 10
c[1] = 20
c[2] = 30

From the function, after modifying the values:
a = -999   b = -999
c[0] = -9
c[1] = -9
c[2] = -9

From main, after calling the function:
a = -999   b = 2
c[0] = -9
c[1] = -9
c[2] = -9
```

We now see that the value of a and the elements of c are altered within `main` as a result of the changes that were made in `modify`. However, the change made to b is confined to the function, as expected. (Compare with the results obtained in the last example, and in Example 7.12.)

The ability to alter an array globally within a function provides a convenient mechanism for moving multiple data items back and forth between the function and the calling portion of the program. Simply pass the array to the function and then alter its elements within the function. Or, if the original array must be preserved, copy the array (element-by-element) within the calling portion of the program, pass the copy to the function, and perform the alterations. You should exercise some caution in altering an array within a function, however, since it is very easy to unintentionally alter the array outside of the function.

EXAMPLE 9.13 Reordering a List of Numbers Consider the well-known problem of rearranging (i.e., *sorting*) a list of n integer quantities into a sequence of increasing values. Let us write a sorting program in such a manner that unnecessary storage will not be used. Therefore the program will contain only one array—a one-dimensional, integer array called `x`, which will be rearranged one element at a time.

The rearrangement will begin by scanning the entire array for the smallest number. This number will then be interchanged with the first number in the array, thus placing the smallest number at the top of the list. Next the remaining $n - 1$ numbers will be scanned for the smallest, which will be exchanged with the second number. The remaining $n - 2$ numbers will then be scanned for the smallest, which will be interchanged with the third number, and so on, until the entire array has been rearranged. The complete rearrangement will require a total of $n - 1$ passes through the array, though the length of each scan will become progressively smaller with each pass.

In order to find the smallest number within each pass, we sequentially compare each number in the array, `x[i]`, with the starting number, `x[item]`, where `item` is an integer variable that is used to identify a particular array element. If `x[i]` is smaller than `x[item]`, then we interchange the two numbers; otherwise we leave the two numbers in their original positions. Once this procedure has been applied to the entire array, the first number in the array will be the smallest. We then repeat the entire procedure $n - 2$ times, for a total of $n - 1$ passes (`item` = 0, 1, . . . , $n - 2$).

The only remaining question is how the two numbers are actually interchanged. To carry out the interchange, we first temporarily save the value of `x[item]` for future reference. Then we assign the current value of `x[i]` to `x[item]`. Finally, we assign the *original* value of `x[item]`, which has temporarily been saved, to `x[i]`. The interchange is now complete.

The strategy described above can be written in C as follows.

```
/* reorder all array elements */
for (item = 0; item < n - 1; ++item)
    /* find the smallest of all remaining elements */
    for (i = item + 1; i < n; ++i)
        if (x[i] < x[item])   {
           /* interchange two elements */
           temp = x[item];
           x[item] = x[i];
           x[i] = temp;
        }
```

We are assuming that `item` and `i` are integer variables that are used as counters, and that `temp` is an integer variable that is used to temporarily store the value of `x[item]`.

It is now a simple matter to add the required variable and array definitions, and the required input/output statements. Here is a complete C program.

```
/* reorder a one-dimensional, integer array from smallest to largest */

#include <stdio.h>

#define  SIZE  100

void reorder(int n, int x[]);

main()
{
    int i, n, x[SIZE];

    /* read in a value for n */
    printf("\nHow many numbers will be entered? ");
    scanf("%d", &n);
    printf("\n");

    /* read in the list of numbers */
```

```
    for (i = 0; i < n; ++i)   {
        printf("i = %d   x = ", i + 1);
        scanf("%d", &x[i]);
    }

    /* reorder all array elements */
    reorder(n, x);

    /* display the reordered list of numbers */
    printf("\n\nReordered List of Numbers:\n\n");
    for (i = 0; i < n; ++i)
        printf("i = %d   x = %d\n", i + 1, x[i]);
}


void reorder(int n, int x[])    /* rearrange the list of numbers */

{
    int i, item, temp;

    for (item = 0; item < n - 1; ++item)
        /* find the smallest of all remaining elements */
        for (i = item + 1; i < n; ++i)
            if (x[i] < x[item])   {
               /* interchange two elements */
               temp = x[item];
               x[item] = x[i];
               x[i] = temp;
            }
    return;
}
```

In this program x is defined initially as a 100-element integer array. (Notice the use of the symbolic constant SIZE to define the size of x.) A value for n is first read into the computer, followed by numerical values for the first n elements of x (i.e., x[0], x[1], . . . , x[n - 1]). Following the data input, n and x are passed to the function reorder, where the first n elements of x are rearranged into ascending order. The reordered elements of x are then displayed from main at the conclusion of the program.

The declaration for reorder appearing in main is written as a function prototype, as a matter of good programming practice. Notice the manner in which the function arguments are written. In particular, note that the second argument is identified as an integer array by the empty square brackets that follow the array name, i.e., int x[]. The square brackets are a required part of this argument specification.

Now suppose that the program is used to reorder the following six numbers: 595 78 −1505 891 −29 −7. The program will generate the following interactive dialog. (The user's responses are underlined, as usual.)

```
How many numbers will be entered? 6

i = 1   x = 595
i = 2   x = 78
i = 3   x = -1505
i = 4   x = 891
i = 5   x = -29
i = 6   x = -7


Reordered list of numbers:
```

```
i = 1   x = -1505
i = 2   x = -29
i = 3   x = -7
i = 4   x = 78
i = 5   x = 595
i = 6   x = 891
```

It should be mentioned that *the* `return` *statement cannot be used to return an array*, since `return` can pass only a *single-valued* expression back to the calling portion of the program. Therefore, if the elements of an array are to be passed back to the calling portion of the program, the array must either be defined as an external array whose scope includes both the function and the calling portion of the program, or it must be passed to the function as a formal argument.

EXAMPLE 9.14 A Piglatin Generator Piglatin is an encoded form of English that is often used by children as a game. A piglatin word is formed from an English word by transposing the first sound (usually the first letter) to the end of the word, and then adding the letter "a". Thus, the word "dog" becomes "ogda," "computer" becomes "omputerca," "piglatin" becomes "iglatinpa" (or "igpa atinla," if spelled as two separate words), and so on.

Let us write a C program that will accept a line of English text and then print out the corresponding text in piglatin. We will assume that each textual message can be typed on one 80-column line, with a single blank space between successive words. (Actually, we will require that the *piglatin* message not exceed 80 characters. Therefore the original message must be somewhat less than 80 characters, since the corresponding piglatin message will be lengthened by the addition of the letter "a" after each word.) For simplicity, we will transpose only the first letter (not the first sound) of each word. Also, we will ignore any special consideration that might be given to capital letters and to punctuation marks.

We will use two character arrays in this program. One array will contain the original line of English text, and the other will contain the translated piglatin.

The overall computational strategy will be straightforward, consisting of the following major steps.

1. Initialize both arrays by assigning blank spaces to all of the elements.
2. Read in an entire line of text (several words).
3. Determine the number of words in the line (by counting the number of single blank spaces that are followed by a nonblank space).
4. Rearrange the words into piglatin, on a word-by-word basis, as follows:
   (*a*) Locate the end of the word.
   (*b*) Transpose the first letter to the end of the word and then add an "a."
   (*c*) Locate the beginning of the next word.
5. Display the entire line of piglatin.

We will continue this procedure repetitively, until the computer reads a line of text whose first three letters are "end" (or "END").

In order to implement this strategy we will make use of two markers, called `m1` and `m2`, respectively. The first marker (`m1`) will indicate the position of the beginning of a particular word within the original line of text. The second marker (`m2`) will indicate the end of the word. Note that the character in the column preceding column number `m1` will be a blank space (except for the first word). Also, note that the character in the column beyond column number `m2` will be a blank space.

This program lends itself to the use of a function for carrying out each of the major tasks. Before discussing the individual functions, however, we define the following program variables.

`english` = a one-dimensional character array that represents the original line of text

`piglatin` = a one-dimensional character array that represents the new line of text (i.e., the piglatin)

`words` = an integer variable that indicates the number of words in the given line of text

`n` = an integer variable that is used as a word counter (`n` = 1, 2, . . . , `words`)

`count` = an integer variable that is used as a character counter within each line (`count` = 0, 1, 2, . . . , 79)

We will also make use of the integer variables `m1` and `m2` discussed earlier.

Now let us return to the overall program outline presented above. The first step, array initialization, can be carried out in a straightforward manner with the following function.

```
/* initialize the character arrays with blank spaces */

void initialize(char english[], char piglatin[])

{
    int count;

    for (count = 0; count < 80; ++count)
        english[count] = piglatin[count] = ' ';
    return;
}
```

Step 2 can also be carried out with a simple function. This procedure will contain a `while` loop that will continue to read characters from the keyboard until an end of line is detected. This sequence of characters will become the elements of the character array `english`. Here is the complete function.

```
/* read one line of English text */

void readinput(char english[])

{
    int count = 0;
    char c;

    while ((c = getchar()) != '\n')   {
          english[count] = c;
          ++count;
    }
    return;
}
```

Step 3 of the overall outline is equally straightforward. We simply scan the original line for occurrences of single blank characters followed by nonblank characters. The word counter (`words`) is then incremented each time a single blank character is encountered. Here is the word-count routine.

```
/* scan the English text and determine the number of words */

int countwords(char english[])

{
    int count, words = 1;

    for (count = 0; count < 79; ++count)
    if (english[count] == ' ' && english[count + 1] != ' ')
       ++words;
    return (words);
}
```

Now consider step 4 (rearrange the English text into piglatin), which is really the heart of the program. The logic for carrying this out is rather involved since it requires three separate, though related, operations. We must first identify the end of each word by finding the first blank space beyond `m1`. We then assign the characters that make up the word to the character array `piglatin`, with the first character at the end of the word. Finally, we must reset the initial marker, to identify the beginning of the next word.

The logic must be handled carefully, since the new line of text will be longer than the original line (because of the latter "a" added at the end). Hence, the characters in the first piglatin word will occupy locations `m1` to `m2+1`. The characters in the second word will occupy locations `m1+1` to `m2+2` (note that these are new values for `m1` and `m2`), and so on. These rules can be generalized as follows.

First, for word number *n*, transfer all characters except the first from the original line to the new line. This can be accomplished by writing

```
for (count = m1; count < m2; ++count)
    piglatin[count + (n - 1)] = english[count + 1];
```

The last two characters (i.e., the first character in the original word plus the letter "a") can then be added in the following manner.

```
piglatin[m2 + (n - 1)] = english[m1];
piglatin[m2 + n] = 'a';
```

We then reset the value of `m1`, i.e.,

```
m1 = m2 + 2;
```

in preparation for the next word. This entire group of calculations is repeated for each word in the original line.

Here is the function that accomplishes all of this.

```
/* convert each word into piglatin */

void convert(int words, char english[], char piglatin[])

{
    int n, count;
    int m1 = 0;          /* marker -> beginning of word */
    int m2;              /* marker -> end of word       */

    /* convert each word */
    for (n = 1; n <= words; ++n)   {

        /* locate the end of the current word */
        count = m1;
        while (english[count] != ' ')
              m2 = count++;

        /* transpose the first letter and add 'a' */
        for (count = m1; count < m2; ++count)
            piglatin[count + (n - 1)] = english[count + 1];
        piglatin[m2 + (n - 1)] = english[m1];
        piglatin[m2 + n] = 'a';

        /* reset the initial marker */
        m1 = m2 + 2;
    }
    return;
}
```

Step 5 (display the piglatin) requires little more than a `for` loop. The complete function can be written as

```
/* display the line of text in piglatin */

void writeoutput(char piglatin[])
{
    int count = 0;

    for (count = 0; count < 80; ++count)
        putchar(piglatin[count]);
    printf("\n");
    return;
}
```

Now consider the main portion of the program. This is nothing more than a group of definitions and declarations, an initial message, a `do - while` loop that allows for repetitious program execution (until the word "end" is detected, in either upper or lowercase, as the first word in the english text), and a closing message. The `do - while` loop can be made to continue indefinitely by using the test `(words >= 0)` at the end of the loop. Since `words` is assigned an initial value of 1 and its value does not decrease, the test will always be true.

The complete program is shown below.

```
/* convert English to piglatin, one line at a time */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void initialize(char english[], char piglatin[]);
void readinput(char english[]);
int countwords(char english[]);
void convert(int words, char english[], char piglatin[]);
void writeoutput(char piglatin[]);

main()
{
    char english[80], piglatin[80];
    int words;

    printf("Welcome to the Piglatin Generator\n\n");
    printf("Type \'END\' when finished\n\n");

    do {  /* process a new line of text */

          initialize(english, piglatin);
          readinput(english);

          /* test for stopping condition */
          if (toupper(english[0]) == 'E' &&
              toupper(english[1]) == 'N' &&
              toupper(english[2]) == 'D')   break;

          /* count the number of words in the line */
          words = countwords(english);

          /* convert english into piglatin */
          convert(words, english, piglatin);
          writeoutput(piglatin);
    }
    while (words >= 0);

    printf("\naveHa aa icena ayda  (Have a nice day)\n");
}
```

```
/* initialize the character arrays with blank spaces */
void initialize(char english[], char piglatin[])
{
    int count;

    for (count = 0; count < 80; ++count)
        english[count] = piglatin[count] = ' ';
    return;
}

/* read one line of English text */
void readinput(char english[])
{
    int count = 0;
    char c;

    while ((c = getchar()) != '\n')   {
          english[count] = c;
          ++count;
    }
    return;
}

/* scan the English text and determine the number of words */
int countwords(char english[])
{
    int count, words = 1;

    for (count = 0; count < 79; ++count)
        if (english[count] == ' ' && english[count + 1] != ' ')
           ++words;
    return (words);
}

/* convert each word into piglatin */
void convert(int words, char english[], char piglatin[])
{
    int n, count;
    int m1 = 0;          /* marker -> beginning of word */
    int m2;              /* marker -> end of word       */

    /* convert each word */
    for (n = 1; n <= words; ++n)   {

        /* locate the end of the current word */
        count = m1;
        while (english[count] != ' ')
              m2 = count++;

        /* transpose the first letter and add 'a' */
        for (count = m1; count < m2; ++count)
            piglatin[count + (n - 1)] = english[count + 1];
        piglatin[m2 + (n - 1)] = english[m1];
        piglatin[m2 + n] = 'a';
```

```
            /* reset the initial marker */
            m1 = m2 + 2;
        }
        return;
    }

    /* display the line of text in piglatin */

    void writeoutput(char piglatin[])

    {
        int count = 0;

        for (count = 0; count < 80; ++count)
            putchar(piglatin[count]);
        printf("\n");
        return;
    }
```

Notice that each function requires at least one array as an argument. In `countwords` and `writeoutput`, the array arguments simply provide input to the functions. In `convert`, however, one array argument provides input to the function and the other provides output to `main`. And in `initialize` and `readinput`, the arrays represent information that is returned to `main`.

The function declarations within `main` are written as full function prototypes. Note that each array argument is identified by an empty pair of square brackets following the array name.

Now consider what happens when the program is executed. Here is a typical interactive session, in which the user's entries are underlined.

```
Welcome to the Piglatin Generator

Type 'END' when finished

C is a popular structured programming language
Ca sia aa opularpa tructuredsa rogrammingpa anguagela

baseball is the great American pastime.
aseballba sia heta reatga mericanAa astime,pa

though there are many who prefer football
houghta hereta reaa anyma howa referpa ootballfa

please do not sneeze in the computer room
leasepa oda otna neezesa nia heta omputerca oomra

end

aveHa aa icena ayda  (Have a nice day)
```

The program does not include any special accommodations for punctuation marks, uppercase letters, or double-letter sounds (e.g., "th" or "sh"). These refinements are left as exercises for the reader.

9.4 MULTIDIMENSIONAL ARRAYS

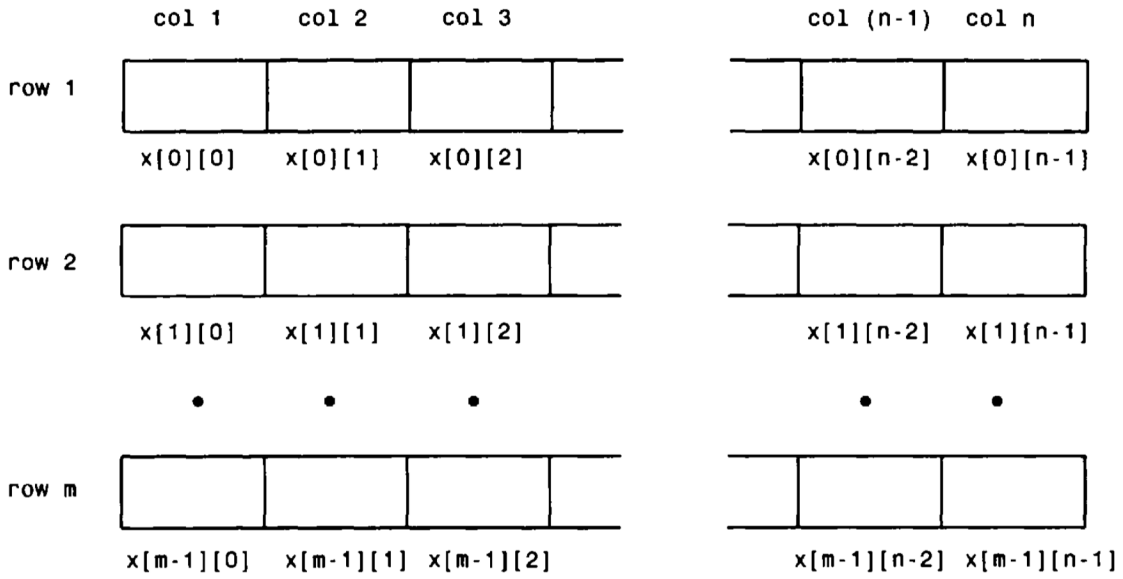
Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on.

In general terms, a multidimensional array definition can be written as

storage-class data-type array[*expression 1*][*expression 2*] . . .[*expression n*];

where *storage-class* refers to the storage class of the array, *data-type* is its data type, *array* is the array name, and *expression 1, expression 2, . . ., expression n* are positive-valued integer expressions that indicate the number of array elements associated with each subscript. Remember that the *storage-class* is optional; the default values are *automatic* for arrays that are defined inside of a function, and *external* for arrays defined outside of a function.

We have already seen that an n-element, one-dimensional array can be thought of as a *list* of values, as illustrated in Fig. 9.1. Similarly, an m × n, two-dimensional array can be thought of as a *table* of values having m rows and n columns, as illustrated in Fig. 9.2. Extending this idea, a three-dimensional array can be visualized as a *set* of tables (e.g., a book in which each page is a table), and so on.



x is a m × n, two-dimensional array

Fig. 9.2

EXAMPLE 9.15 Several typical multidimensional array definitions are shown below.

```
float table[50][50];

char page[24][80];

static double records[100][66][255];

static double records[L][M][N];
```

The first line defines `table` as a floating-point array having 50 rows and 50 columns (hence 50 × 50 = 2500 elements), and the second line establishes `page` as a character array with 24 rows and 80 columns (24 × 80 = 1920 elements). The third array can be thought of as a set of 100 static, double-precision tables, each having 66 lines and 255 columns (hence 100 × 66 × 255 = 1,683,000 elements).

The last definition is similar to the preceding definition except that the array size is defined by the symbolic constants `L`, `M` and `N`. Thus, the values assigned to these symbolic constants will determine the actual size of the array.

Some care must be given to the order in which initial values are assigned to multidimensional array elements. (Remember, *only external and static arrays can be initialized.*) The rule is that the last (rightmost) subscript increases most rapidly, and the first (leftmost) subscript increases least rapidly. Thus, the elements of a two-dimensional array will be assigned by rows; i.e., the elements of the first row will be assigned, then the elements of the second row, and so on.

EXAMPLE 9.16 Consider the following two-dimensional array definition.

```
int values[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Note that `values` can be thought of as a table having 3 rows and 4 columns (4 elements per row). Since the initial values are assigned by rows (i.e., last subscript increasing most rapidly), the results of this initial assignment are as follows.

```
values[0][0] = 1    values[0][1] = 2    values[0][2] = 3    values[0][3] = 4
values[1][0] = 5    values[1][1] = 6    values[1][2] = 7    values[1][3] = 8
values[2][0] = 9    values[2][1] = 10   values[2][2] = 11   values[2][3] = 12
```

Remember that the first subscript ranges from 0 to 2, and the second subscript ranges from 0 to 3.

The natural order in which the initial values are assigned can be altered by forming groups of initial values enclosed within braces (i.e., { . . . }). The values within each innermost pair of braces will be assigned to those array elements whose last subscript changes most rapidly. In a two-dimensional array, for example, the values within an inner pair of braces will be assigned to the elements of a row, since the second (column) subscript increases most rapidly. If there are too few values within a pair of braces, the remaining elements of that row will be assigned zeros. However, the number of values within each pair of braces cannot exceed the defined row size.

EXAMPLE 9.17 Here is a variation of the two-dimensional array definition presented in the last example.

```
int values[3][4] = {
                      {1, 2, 3, 4},
                      {5, 6, 7, 8},
                      {9, 10, 11, 12}
                   };
```

This definition results in the same initial assignments as in the last example. Thus, the four values in the first inner pair of braces are assigned to the array elements in the first row, the values in the second inner pair of braces are assigned to the array elements in the second row, etc. Note that an outer pair of braces is required, containing the inner pairs.

Now consider the following two-dimensional array definition.

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

This definition assigns values only to the first three elements in each row. Therefore, the array elements will have the following initial values.

```
values[0][0] = 1    values[0][1] = 2    values[0][2] = 3    values[0][3] = 0
values[1][0] = 4    values[1][1] = 5    values[1][2] = 6    values[1][3] = 0
values[2][0] = 7    values[2][1] = 8    values[2][2] = 9    values[2][3] = 0
```

Notice that the last element in each row is assigned a value of zero.

If the preceding array definition is written as

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

then three of the array elements will again be assigned zeros, though the order of the assignments will be different. In particular, the array elements will have the following initial values.

```
values[0][0] = 1    values[0][1] = 2    values[0][2] = 3    values[0][3] = 4
values[1][0] = 5    values[1][1] = 6    values[1][2] = 7    values[1][3] = 8
values[2][0] = 9    values[2][1] = 0    values[2][2] = 0    values[2][3] = 0
```

Now the initial values are assigned with the last subscript increasing most rapidly, on a row-by-row basis, until all of the initial values have been assigned. Without the inner pairs of braces, however, the initial values cannot be grouped for assignment to specific rows.

Finally, consider the array definition

```
int values[3][4] = {
                       {1, 2, 3, 4, 5},
                       {6, 7, 8, 9, 10},
                       {11, 12, 13, 14, 15}
                   };
```

This will result in a compilation error, since the number of values in each inner pair of braces (five values in each pair) exceeds the defined array size (four elements in each row).

The use of embedded groups of initial values can be generalized to higher dimensional arrays.

EXAMPLE 9.18 Consider the following three-dimensional array definition.

```
int t[10][20][30] = {
                        {                       /* table 1 */
                            {1, 2, 3, 4},       /* row 1 */
                            {5, 6, 7, 8},       /* row 2 */
                            {9, 10, 11, 12}     /* row 3 */
                        },
                        {                       /* table 2 */
                            {21, 22, 23, 24},   /* row 1 */
                            {25, 26, 27, 28},   /* row 2 */
                            {29, 30, 31, 32}    /* row 3 */
                        }
                    };
```

Think of this array as a collection of 10 tables, each having 20 rows and 30 columns. The groups of initial values will result in the assignment of the following nonzero values in the first two tables.

```
t[0][0][0] = 1    t[0][0][1] = 2    t[0][0][2] = 3    t[0][0][3] = 4
t[0][1][0] = 5    t[0][1][1] = 6    t[0][1][2] = 7    t[0][1][3] = 8
t[0][2][0] = 9    t[0][2][1] = 10   t[0][2][2] = 11   t[0][2][3] = 12

t[1][0][0] = 21   t[1][0][1] = 22   t[1][0][2] = 23   t[1][0][3] = 24
t[1][1][0] = 25   t[1][1][1] = 26   t[1][1][2] = 27   t[1][1][3] = 28
t[1][2][0] = 29   t[1][2][1] = 30   t[1][2][2] = 31   t[1][2][3] = 32
```

All of the remaining array elements will be assigned zeros.

Multidimensional arrays are processed in the same manner as one-dimensional arrays, on an element-by-element basis. However, some care is required when passing multidimensional arrays to a function. In particular, the formal argument declarations within the function definition *must* include explicit size specifications in all of the subscript positions *except the first*. These size specifications must be consistent

with the corresponding size specifications in the calling program. The first subscript position may be written as an empty pair of square brackets, as with a one-dimensional array. The corresponding function prototypes must be written in the same manner.

EXAMPLE 9.19 Adding Two Tables of Numbers Suppose we want to read two tables of integers into the computer, calculate the sums of the corresponding elements, i.e.,

```
c[i][j] = a[i][j] + b[i][j]
```

and then display the new table containing these sums. We will assume that all of the tables contain the same number of rows and columns, not exceeding 20 rows and 30 columns.

Let us make use of the following variable and array definitions.

`a, b, c` = two-dimensional arrays, each having the same number of rows and the same number of columns, not exceeding 20 rows and 30 columns

`nrows` = an integer variable indicating the actual number of rows in each table

`ncols` = an integer variable indicating the actual number of columns in each table

`row` = an integer counter that indicates the row number

`col` = an integer counter that indicates the column number

The program will be modularized by writing separate functions to read in an array, calculate the sum of the array elements, and display an array. Let us call these functions `readinput`, `computesums` and `writeoutput`, respectively.

The logic within each function is quite straightforward. Here is a complete C program for carrying out the computation.

```
/* calculate the sum of the elements in two tables of integers */

#include <stdio.h>

#define MAXROWS  20
#define MAXCOLS  30

void readinput(int a[][MAXCOLS], int nrows, int ncols);
void computesums(int a[][MAXCOLS], int b[][MAXCOLS],
                 int c[][MAXCOLS], int nrows, int ncols);
void writeoutput(int c[][MAXCOLS], int nrows, int ncols);


main()

{
    int nrows, ncols;

    /* array definitions */
    int a[MAXROWS][MAXCOLS], b[MAXROWS][MAXCOLS], c[MAXROWS][MAXCOLS];

    printf("How many rows? ");
    scanf("%d", &nrows);
    printf("How many columns? ");
    scanf("%d", &ncols);

    printf("\n\nFirst table:\n");
    readinput(a, nrows, ncols);

    printf("\n\nSecond table:\n");
    readinput(b, nrows, ncols);

    computesums(a, b, c, nrows, ncols);
```

```
    printf("\n\nSums of the elements:\n\n");
    writeoutput(c, nrows, ncols);
}

/* read in a table of integers */

void readinput(int a[][MAXCOLS], int m, int n)

{
    int row, col;

    for (row = 0; row < m; ++row)   {
        printf("\nEnter data for row no. %2d\n", row + 1);
        for (col = 0; col < n; ++col)
            scanf("%d", &a[row][col]);
    }
    return;
}

/* add the elements of two integer tables */

void computesums(int a[][MAXCOLS], int b[][MAXCOLS],
                 int c[][MAXCOLS], int m, int n)

{
    int row, col;

    for (row = 0; row < m; ++row)
        for (col = 0; col < n; ++col)
            c[row][col] = a[row][col] + b[row][col];
    return;
}

/* display a table of integers */

void writeoutput(int a[][MAXCOLS], int m, int n)

{
    int row, col;

    for (row = 0; row < m; ++row)   {
        for (col = 0; col < n; ++col)
            printf("%4d", a[row][col]);
        printf("\n");
    }
    return;
}
```

The array definitions are expressed in terms of the symbolic constants MAXROWS and MAXCOLS, whose values are specified as 20 and 30, respectively, at the beginning of the program.

Notice the manner in which the formal argument declarations are written within each function definition. For example, the first line of function readinput is written as

```
void readinput(int a[][MAXCOLS], int m, int n)
```

The array name, a, is followed by two pairs of square brackets. The first pair is empty, because the number of rows need not be specified explicitly. However, the second pair contains the symbolic constant MAXCOLS, which provides an explicit

size specification for the number of columns. The array names appearing in the other function definitions (i.e., in functions `computesums` and `writeoutput`) are written in the same manner.

Also, notice the function prototypes at the beginning of the program. Each prototype is analogous to the first line of the corresponding function definition. In particular, each array name is followed by two pairs of brackets, the first of which is empty. The second pair of brackets contains the size specification for the number of columns, as required.

Now suppose the program is used to sum the following two tables of numbers.

First table				*Second table*			
1	2	3	4	10	11	12	13
5	6	7	8	14	15	16	17
9	10	11	12	18	19	20	21

Execution of the program will generate the following dialog. (The user's responses are underlined, as usual.)

```
How many rows? 3
How many columns? 4

First table:

Enter data for row no.   1
1 2 3 4

Enter data for row no.   2
5 6 7 8

Enter data for row no.   3
9 10 11 12

Second table:

Enter data for row no.   1
10 11 12 13

Enter data for row no.   2
14 15 16 17

Enter data for row no.   3
18 19 20 21

Sums of the elements:

11  13  15  17
19  21  23  25
27  29  31  33
```

Some C compilers are unable to pass sizeable multidimensional arrays to functions. In such situations it may be possible to redesign the program so that the multidimensional arrays are defined as external (global) arrays. Hence, the arrays need not be passed to functions as arguments. This strategy will not always work, however, because some programs (such as the program shown in the last example) use the same function to process different arrays. Problems of this type can usually be circumvented through the use of pointers, as discussed in the next chapter.

9.5 ARRAYS AND STRINGS

We have already seen that a string can be represented as a one-dimensional character-type array. Each character within the string will be stored within one element of the array. Some problems require that the

characters within a string be processed individually (e.g., the piglatin generator shown in Example 9.14). However, there are many other problems which require that strings be processed as complete entities. Such problems can be simplified considerably through the use special string-oriented library functions.

For example, most C compilers include library functions that allow strings to be compared, copied or *concatenated* (i.e., combined, one behind another). Other library functions permit operations on individual characters within strings; e.g., they allow individual characters to be located within strings, and so on. The following example illustrates the use of some of these library functions.

EXAMPLE 9.20 Reordering a List of Strings Suppose we wish to enter a list of strings into the computer, rearrange them into alphabetical order, and then display the rearranged list. The strategy for doing this is very similar to that shown in Example 9.13, where we rearranged a list of numbers into ascending order. Now, however, there is the additional complication of comparing entire strings, rather than single numerical values. We will therefore store the strings within a two-dimensional character array. Each string will be stored in a separate row within the array.

To simplify the computation, let us make use of the library functions `strcmp` and `strcpy`. These functions are used to compare two strings and to copy one string to another, respectively. (Some compilers also include the `strcmpi` function, which is a variation of the more common `strcmp`. The use of `strcmpi` is sometimes more convenient, since it does not distinguish between upper- and lowercase. However, it is not supported by the ANSI standard.)

The `strcmp` function accepts two strings as arguments and returns an integer value, depending upon the relative order of the two strings, as follows:

1. A negative value if the first string precedes the second string alphabetically.
2. A value of zero if the first string and the second string are identical (disregarding case).
3. A positive value if the second string precedes the first string alphabetically.

Therefore, if `strcmp(`*`string1, string2`*`)` returns a positive value, it would indicate that *`string2`* must be moved, placing it ahead of *`string1`* in order to alphabetize the two strings properly.

The `strcpy` function also accepts two strings as arguments. Its first argument is generally an identifier that represents a string. The second argument can be a string constant or an identifier representing another string. The function copies the value of *`string2`* to *`string1`*. Hence, it effectively causes one string to be assigned to another.

The complete program is very similar to the numerical reordering program presented in Example 9.13. Now, however, we will allow the program to accept an unspecified number of strings, until a string is entered whose first three characters are END (in either upper- or lowercase). The program will count the strings as they are entered, ignoring the last string, which contains END.

Here is the entire program.

```
/* sort a list of strings alphabetically using a two-dimensional character array */

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

void reorder(int n, char x[][12]);     /* function prototype */

main()
{
    int i, n = 0;
    char x[10][12];

    printf("Enter each string on a separate line below\n\n");
    printf("Type \'END\' when finished\n\n");

    /* read in the list of strings */
    do  {
            printf("string %d: ", n + 1);
            scanf("%s", x[n]);
        }  while (strcmp(x[n++], "END"));
```

```
    /* adjust the value of n */
    n--;
    /* reorder the list of strings */
    reorder(n, x);
    /* display the reordered list of strings */
    printf("\n\nReordered List of Strings:\n");
    for (i = 0; i < n; ++i)
        printf("\nstring %d: %s", i + 1, x[i]);
}

void reorder(int n, char x[][12])    /* rearrange the list of strings */
{
    char temp[12];
    int i, item;
    for (item = 0; item < n - 1; ++item)
        /* find the lowest of all remaining strings */
        for (i = item + 1; i < n; ++i)
            if (strcmp(x[item], x[i]) > 0)   {
                /* interchange the two strings */
                strcpy(temp, x[item]);
                strcpy(x[item], x[i]);
                strcpy(x[i], temp);
            }
    return;
}
```

The `strcmp` function appears in two different places within this program: in `main`, when testing for a stopping condition, and in `rearrange`, when testing for the need to interchange two strings. The actual string interchange is carried out using `strcpy`.

The dialog resulting from a typical execution of the program is shown below. The user's responses are underlined, as usual.

```
Enter each string on a separate line below

Type 'END' when finished

string 1: PACIFIC
string 2: ATLANTIC
string 3: INDIAN
string 4: CARIBBEAN
string 5: BERING
string 6: BLACK
string 7: RED
string 8: NORTH
string 9: BALTIC
string 10: CASPIAN
string 11: END
```