Table 4.1 **Basic Data Types**

Type	Constant Examples	printf chars
`char`	`'a', '\n'`	`%c`
`_Bool`	`0`, `1`	`%i`, `%u`
`short int`	—	`%hi`, `%hx`, `%ho`
`unsigned short int`	—	`%hu`, `%hx`, `%ho`
`int`	`12`, `-97`, `0xFFE0`, `0177`	`%i`, `%x`, `%o`
`unsigned int`	`12u`, `100U`, `0XFFu`	`%u`, `%x`, `%o`
`long int`	`12L`, `-2001`, `0xffffL`	`%li`, `%lx`, `%lo`
`unsigned long int`	`12UL`, `100ul`, `0xffeeUL`	`%lu`, `%lx`, `%lo`
`long long int`	`0xe5e5e5e5LL`, `500ll`	`%lli`, `%llx`, `&llo`
`unsigned long long int`	`12ull`, `0xffeeULL`	`%llu`, `%llx`, `%llo`
`float`	`12.34f`, `3.1e-5f, 0x1.5p10,0x1P-1`	`%f`, `%e`, `%g, %a`
`double`	`12.34`, `3.1e-5, 0x.1p3`	`%f`, `%e`, `%g, %a`
`long double`	`12.341`, `3.1e-5l`	`%Lf`, `$Le`, `%Lg`

Working with Arithmetic Expressions

In C, just as in virtually all programming languages, the plus sign (+) is used to add two values, the minus sign (-) is used to subtract two values, the asterisk (*) is used to multiply two values, and the slash (/) is used to divide two values. These operators are known as *binary* arithmetic operators because they operate on two values or terms.

You have seen how a simple operation such as addition can be performed in C. Program 4.2 further illustrates the operations of subtraction, multiplication, and division. The last two operations performed in the program introduce the notion that one operator can have a higher priority, or *precedence*, over another operator. In fact, each operator in C has a precedence associated with it. This precedence is used to determine how an expression that has more than one operator is evaluated: The operator with the higher precedence is evaluated first. Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the *associative* property of an operator. Appendix A provides a complete list of operator precedences and their rules of association.

Program 4.2 **Using the Arithmetic Operators**

```
// Illustrate the use of various arithmetic operators

#include <stdio.h>

int main (void)
{
```

Program 5.9 **Output**

```
Enter your number.
13579
97531
```

Program 5.9 **Output (Rerun)**

```
Enter your number.
0
0
```

As you can see from the program's output, when 0 is keyed into the program, the program correctly displays the digit 0.

The `break` Statement

Sometimes when executing a loop, it becomes desirable to leave the loop as soon as a certain condition occurs (for instance, you detect an error condition, or you reach the end of your data prematurely). The `break` statement can be used for this purpose. Execution of the `break` statement causes the program to immediately exit from the loop it is executing, whether it's a `for`, `while`, or `do` loop. Subsequent statements in the loop are skipped, and execution of the loop is terminated. Execution continues with whatever statement follows the loop.

If a `break` is executed from within a set of nested loops, only the innermost loop in which the `break` is executed is terminated.

The format of the `break` statement is simply the keyword `break` followed by a semicolon:

```
break;
```

The `continue` Statement

The `continue` statement is similar to the `break` statement except it doesn't cause the loop to terminate. Rather, as its name implies, this statement causes the loop in which it is executed to be continued. At the point that the `continue` statement is executed, any statements in the loop that appear *after* the `continue` statement are automatically skipped. Execution of the loop otherwise continues as normal.

The `continue` statement is most often used to bypass a group of statements inside a loop based upon some condition, but to otherwise continue execution of the loop. The format of the `continue` statement is simply

```
continue;
```

Don't use the `break` or `continue` statements until you become very familiar with writing program loops and gracefully exiting from them. These statements are too easy to abuse and can result in programs that are hard to follow.

The notation $M_{3,2}$ refers to the value `20`, which is found in the 3rd row, 2nd column of the matrix. In a similar fashion, $M_{4,5}$ refers to the element contained in the 4th row, 5th column: the value `6`.

In C, you can use an analogous notation when referring to elements of a two-dimensional array. However, because C likes to start numbering things at zero, the 1st row of the matrix is actually row 0, and the 1st column of the matrix is column 0. The preceding matrix would then have row and column designations, as shown in Table 7.3.

Table 7.3 **A 4 x 5 Matrix in C**

Column (j) / **Row (i)**	*0*	*1*	*2*	*3*	*4*
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

Whereas in mathematics the notation $M_{i,j}$ is used, in C the equivalent notation is

```
M[i][j]
```

Remember, the first index number refers to the row number, whereas the second index number references the column. So the statement

```
sum = M[0][2] + M[2][4];
```

adds the value contained in row 0, column 2—which is `-3`—to the value contained in row 2, column 4—which is `14`—and assigns the result of `11` to the variable `sum`.

Two-dimensional arrays are declared the same way that one-dimensional arrays are; thus

```
int  M[4][5];
```

declares the array `M` to be a two-dimensional array consisting of 4 rows and 5 columns, for a total of 20 elements. Each position in the array is defined to contain an integer value.

Two-dimensional arrays can be initialized in a manner analogous to their one-dimensional counterparts. When listing elements for initialization, the `values` are listed by row. Brace pairs are used to separate the list of initializers for one row from the next. So to define and initialize the array `M` to the elements listed in Table 7.3, a statement such as the following can be used:

```
int  M[4][5] = {
                    { 10,  5, -3, 17, 82 },
                    {  9,  0,  0,  8, -7 },
                    { 32, 20,  1,  0, 14 },
                    {  0,  0,  8,  7,  6 }
               };
```

Pay particular attention to the syntax of the preceding statement. Note that commas are required after each brace that closes off a row, except in the case of the final row. The use of the inner pairs of braces is actually optional. If not supplied, initialization proceeds by row. Thus, the preceding statement could also have been written as follows:

```
int  M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
                 20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

As with one-dimensional arrays, it is not required that the entire array be initialized. A statement such as

```
int  M[4][5] = {
                   { 10,  5, -3 },
                   {  9,  0,  0 },
                   { 32, 20,  1 },
                   {  0,  0,  8 }
                };
```

only initializes the first three elements of each row of the matrix to the indicated values. The remaining values are set to `0`. Note that, in this case, the inner pairs of braces *are required* to force the correct initialization. Without them, the first two rows and the first two elements of the 3rd row would have been initialized instead. (Verify to yourself that this is the case.)

Subscripts can also be used in the initialization list, in a like manner to single-dimensional arrays. So the declaration

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

initializes the three indicated elements of `matrix` to the specified values. The unspecified elements are set to zero by default.

Variable-Length Arrays[1]

This section discusses a feature in the language that enables you to work with arrays in your programs without having to give them a constant size.

In the examples in this chapter, you have seen how the size of an array is declared to be of a specific size. The C language allows you to declare arrays of a variable size. For example, Program 7.3 only calculates the first 15 Fibonacci numbers. But what if you want to calculate 100 or even 500 Fibonacci numbers? Or, what if you want to have the user specify the number of Fibonacci numbers to generate? Study Program 7.8 to see one method for resolving this problem.

1. As of this writing, full support for variable-length arrays was not offered by all compilers. You might want to check your compiler's documentation before you use this feature.

Program 8.9 **Continued**

```
int  minimum (int  values[10])
{
    int  minValue, i;

    minValue = values[0];

    for ( i = 1;  i < 10;  ++i )
        if ( values[i] < minValue )
           minValue = values[i];

    return minValue;
}

int main (void)
{
    int  scores[10], i, minScore;
    int  minimum (int  values[10]);

    printf ("Enter 10 scores\n");

    for ( i = 0;  i < 10;  ++i )
        scanf ("%i", &scores[i]);

    minScore = minimum (scores);
    printf ("\nMinimum score is %i\n", minScore);

    return 0;
}
```

Program 8.9 **Output**

```
Enter 10 scores
69
97
65
87
69
86
78
67
92
90

Minimum score is 65
```

Program 8.10 **Revising the Function to Find the Minimum Value in an Array**

```
// Function to find the minimum value in an array

#include <stdio.h>

int  minimum (int  values[], int  numberOfElements)
{
    int  minValue, i;

    minValue = values[0];

    for ( i = 1;  i < numberOfElements;  ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
    int  array1[5] = { 157, -28, -37, 26, 10 };
    int  array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    int  minimum (int  values[], int  numberOfElements);

    printf ("array1 minimum: %i\n", minimum (array1, 5));
    printf ("array2 minimum: %i\n", minimum (array2, 7));

    return 0;
}
```

Program 8.10 **Output**

```
array1 minimum: -37
array2 minimum: 1
```

This time, the function `minimum` is defined to take two arguments: first, the array whose minimum you want to find and second, the number of elements in the array. The open and close brackets that immediately follow `values` in the function header serve to inform the C compiler that `values` is an array of integers. As stated previously, the compiler really doesn't need to know how large it is.

The formal parameter `numberOfElements` replaces the constant `10` as the upper limit inside the `for` statement. So the `for` statement sequences through the array from `values[1]` through the last element of the array, which is `values[numberOfElements - 1]`.

An entire multidimensional array can be passed to a function in the same way that a single-dimensional array can: You simply list the name of the array. For example, if the matrix `measured_values` is declared to be a two-dimensional array of integers, the C statement

```
scalarMultiply (measured_values, constant);
```

can be used to invoke a function that multiplies each element in the matrix by the value of `constant`. This implies, of course, that the function itself can change the values contained inside the `measured_values` array. The discussion of this topic for single-dimensional arrays also applies here: An assignment made to any element of the formal parameter array inside the function makes a permanent change to the array that was passed to the function.

When declaring a single-dimensional array as a formal parameter inside a function, you learned that the actual dimension of the array is not needed; simply use a pair of empty brackets to inform the C compiler that the parameter is, in fact, an array. This does not totally apply in the case of multidimensional arrays. For a two-dimensional array, the number of rows in the array can be omitted, but the declaration *must* contain the number of columns in the array. So the declarations

```
int  array_values[100][50]
```

and

```
int  array_values[][50]
```

are both valid declarations for a formal parameter array called `array_values` containing 100 rows by 50 columns; but the declarations

```
int  array_values[100][]
```

and

```
int  array_values[][]
```

are not because the number of columns in the array *must* be specified.

In Program 8.13, you define a function `scalarMultiply`, which multiplies a two-dimensional integer array by a scalar integer value. Assume for purposes of this example that the array is dimensioned 3 x 5. The `main` routine calls the `scalarMultiply` routine twice. After each call, the array is passed to the `displayMatrix` routine to display the contents of the array. Pay careful attention to the nested `for` loops that are used in both `scalarMultiply` and `displayMatrix` to sequence through each element of the two-dimensional array.

Program 8.13 **Using Multidimensional Arrays and Functions**

```
#include <stdio.h>

int main (void)
{
```

Program 8.13 **Continued**

```
    void  scalarMultiply (int  matrix[3][5], int  scalar);
    void  displayMatrix (int  matrix[3][5]);
    int   sampleMatrix[3][5] =
          {
              {  7, 16, 55, 13, 12 },
              { 12, 10, 52,  0,  7 },
              { -2,  1,  2,  4,  9 }
          };

    printf ("Original matrix:\n");
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, 2);

    printf ("\nMultiplied by 2:\n");
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, -1);

    printf ("\nThen multiplied by -1:\n");
    displayMatrix (sampleMatrix);

    return 0;
}

// Function to multiply a 3 x 5 array by a scalar

void  scalarMultiply (int  matrix[3][5], int  scalar)
{
    int  row, column;

    for ( row = 0;  row < 3;  ++row )
        for ( column = 0;  column < 5;  ++column )
            matrix[row][column]  *=  scalar;
}


void  displayMatrix (int  matrix[3][5])
{
    int   row, column;

    for ( row = 0;  row < 3;  ++row) {
        for ( column = 0;  column < 5;  ++column )
            printf ("%5i", matrix[row][column]);
```

Program 8.13 **Continued**

```
        printf ("\n");
    }
}
```

Program 8.13 **Output**

```
Original matrix:
     7  16   55   13   12
    12  10   52    0    7
    -2   1    2    4    9

Multiplied by 2:
    14  32  110   26   24
    24  20  104    0   14
    -4   2    4    8   18

Then multiplied by -1:
   -14  -32 -110  -26  -24
   -24  -20 -104    0  -14
     4   -2   -4   -8  -18
```

The `main` routine defines the matrix `sampleValues` and then calls the `displayMatrix` function to display its initial values at the terminal. Inside the `displayMatrix` routine, notice the nested `for` statements. The first or outermost `for` statement sequences through each row in the matrix, so the value of the variable `row` varies from `0` through `2`. For each value of `row`, the innermost `for` statement is executed. This `for` statement sequences through each column of the particular row, so the value of the variable `column` ranges from `0` through `4`.

The `printf` statement displays the value contained in the specified row and column using the format characters `%5i` to ensure that the elements line up in the display. After the innermost `for` loop has finished execution—meaning that an entire row of the matrix has been displayed—a newline character is displayed so that the next row of the matrix is displayed on the next line of the terminal.

The first call to the `scalarMultiply` function specifies that the `sampleMatrix` array is to be multiplied by 2. Inside the function, a simple set of nested `for` loops is set up to sequence through each element in the array. The element contained in `matrix[row][column]` is multiplied by the value of `scalar` in accordance with the use of the assignment operator `*=`. After the function returns to the `main` routine, the `displayMatrix` function is once again called to display the contents of the `sampleMatrix` array. The program's output verifies that each element in the array has, in fact, been multiplied by 2.

The `scalarMultiply` function is called a second time to multiply the now modified elements of the `sampleMatrix` array by –1. The modified array is then displayed by a final call to the `displayMatrix` function, and program execution is then complete.

Multidimensional Variable-Length Arrays and Functions

You can take advantage of the variable-length array feature in the C language and write functions that can take multidimensional arrays of varying sizes. For example, Program 8.13 can been rewritten so that the `scalarMultiply` and `displayMatrix` functions can accept matrices containing any number of rows and columns, which can be passed as arguments to the functions. See Program 8.13A.

Program 8.13A **Multidimensional Variable-Length Arrays**

```
#include <stdio.h>

int main (void)
{

    void  scalarMultiply (int nRows, int nCols,
                          int  matrix[nRows][nCols], int  scalar);
    void  displayMatrix (int nRows, int nCols, int  matrix[nRows][nCols]);
    int   sampleMatrix[3][5] =
          {
              {  7, 16, 55, 13, 12 },
              { 12, 10, 52,  0,  7 },
              { -2,  1,  2,  4,  9 }
          };

    printf ("Original matrix:\n");
    displayMatrix (3, 5, sampleMatrix);

    scalarMultiply (3, 5, sampleMatrix, 2);
    printf ("\nMultiplied by 2:\n");
    displayMatrix (3, 5, sampleMatrix);

    scalarMultiply (3, 5, sampleMatrix, -1);
    printf ("\nThen multiplied by -1:\n");
    displayMatrix (3, 5, sampleMatrix);

    return 0;
}

// Function to multiply a matrix by a scalar

void  scalarMultiply (int nRows, int nCols,
                      int  matrix[nRows][nCols], int  scalar)
```

Program 8.13A **Continued**

```
{
    int  row, column;

    for ( row = 0;  row < nRows;  ++row )
        for ( column = 0;  column < nCols;  ++column )
            matrix[row][column]  *=  scalar;
}


void  displayMatrix (int nRows, int nCols, int  matrix[nRows][nCols])
{
    int   row, column;

    for ( row = 0;  row < nRows;  ++row) {
        for ( column = 0;  column < nCols;  ++column )
            printf ("%5i", matrix[row][column]);

        printf ("\n");
    }
}
```

Program 8.13A **Output**

```
Original matrix:
     7  16   55   13   12
    12  10   52    0    7
    -2   1    2    4    9

Multiplied by 2:
    14  32  110   26   24
    24  20  104    0   14
    -4   2    4    8   18

Then multiplied by -1:
   -14  -32 -110  -26  -24
   -24  -20 -104    0  -14
     4   -2   -4   -8  -18
```

The function declaration for `scalarMultiply` looks like this:

```
void  scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols], int  scalar)
```

The rows and columns in the matrix, `nRows`, and `nCols`, must be listed as arguments *before* the matrix itself so that the compiler knows about these parameters before it encounters the declaration of matrix in the argument list. If you try it this way instead:

```
void  scalarMultiply (int matrix[nRows][nCols], int nRows, int nCols, int  scalar)
```

9

Working with Structures

Chapter 7, "Working with Arrays," introduced the array that permits you to group elements of the same type into a single logical entity. To reference an element in the array, all that is necessary is that the name of the array be given together with the appropriate subscript.

The C language provides another tool for grouping elements together. This falls under the name of *structures* and forms the basis for the discussions in this chapter. As you will see, the structure is a powerful concept that you will use in many C programs that you develop.

Suppose you want to store a date—for example 9/25/04—inside a program, perhaps to be used for the heading of some program output, or even for computational purposes. A natural method for storing the date is to simply assign the month to an integer variable called `month`, the day to an integer variable called `day`, and the year to an integer variable called `year`. So the statements

```
int  month = 9, day = 25, year = 2004;
```

work just fine. This is a totally acceptable approach. But suppose your program also needs to store the date of purchase of a particular item, for example. You can go about the same procedure of defining three more variables such as `purchaseMonth`, `purchaseDay`, and `purchaseYear`. Whenever you need to use the purchase date, these three variables could then be explicitly accessed.

Using this method, you must keep track of three separate variables for each date that you use in the program—variables that are logically related. It would be much better if you could somehow group these sets of three variables together. This is precisely what the structure in C allows you to do.

A Structure for Storing the Date

You can define a structure called `date` in the C language that consists of three components that represent the month, day, and year. The syntax for such a definition is rather straightforward, as follows:

```
struct  date
{
    int  month;
    int  day;
    int  year;
};
```

The `date` structure just defined contains three integer *members* called `month`, `day`, and `year`. The definition of `date` in a sense defines a new type in the language in that variables can subsequently be declared to be of type `struct date`, as in the declaration

```
struct date  today;
```

You can also declare a variable `purchaseDate` to be of the same type by a separate declaration, such as

```
struct date  purchaseDate;
```

Or, you can simply include the two declarations on the same line, as in

```
struct date  today, purchaseDate;
```

Unlike variables of type `int`, `float`, or `char`, a special syntax is needed when dealing with structure variables. A member of a structure is accessed by specifying the variable name, followed by a period, and then the member name. For example, to set the value of the `day` in the variable `today` to `25`, you write

```
today.day = 25;
```

Note that there are no spaces permitted between the variable name, the period, and the member name. To set the `year` in `today` to `2004`, the expression

```
today.year = 2004;
```

can be used. Finally, to test the value of `month` to see if it is equal to `12`, a statement such as

```
if  ( today.month == 12 )
   nextMonth = 1;
```

does the trick.

Try to determine the effect of the following statement.

```
if  ( today.month == 1  &&  today.day == 1 )
  printf ("Happy New Year!!!\n");
```

Program 9.1 incorporates the preceding discussions into an actual C program.

Program 9.1 **Illustrating a Structure**

```
// Program to illustrate a structure

#include <stdio.h>

int main (void)
{
    struct  date
    {
        int  month;
        int  day;
        int  year;
    };

    struct date  today;

    today.month = 9;
    today.day = 25;
    today.year = 2004;

    printf ("Today's date is %i/%i/%.2i.\n", today.month, today.day,
              today.year % 100);

    return 0;
}
```

Program 9.1 **Output**

```
Today's date is 9/25/04.
```

The first statement inside `main` defines the structure called `date` to consist of three integer members called `month`, `day`, and `year`. In the second statement, the variable `today` is declared to be of type `struct date`. The first statement simply defines what a `date` structure looks like to the C compiler and causes no storage to be reserved inside the computer. The second statement declares a variable to be of type `struct date` and, therefore, *does* cause memory to be reserved for storing the three integer values of the variable `today`. Be certain you understand the difference between defining a structure and declaring variables of the particular structure type.

After `today` has been declared, the program then proceeds to assign values to each of the three members of `today`, as depicted in Figure 9.1.

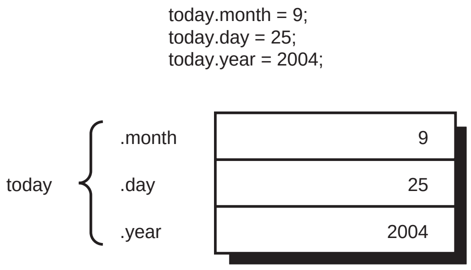

Figure 9.1. Assigning values to a structure variable.

After the assignments have been made, the values contained inside the structure are displayed by an appropriate `printf` call. The remainder of `today.year` divided by 100 is calculated prior to being passed to the `printf` function so that just 04 is displayed for the year. Recall that the format characters `%.2i` are used to specify that two integer digits are to be displayed with zero fill. This ensures that you get the proper display for the last two digits of the year.

Using Structures in Expressions

When it comes to the evaluation of expressions, structure members follow the same rules as ordinary variables do in the C language. So division of an integer structure member by another integer is performed as an integer division, as in

```
century = today.year / 100 + 1;
```

Suppose you want to write a simple program that accepts today's date as input and displays tomorrow's date to the user. Now, at first glance, this seems a perfectly simple task to perform. You can ask the user to enter today's date and then proceed to calculate tomorrow's date by a series of statements, such as

```
tomorrow.month = today.month;
tomorrow.day   = today.day + 1;
tomorrow.year  = today.year;
```

Of course, the preceding statements work just fine for the majority of dates, but the following two cases are not properly handled:

1. If today's date falls at the end of a month.
2. If today's date falls at the end of a year (that is, if today's date is December 31).

One way to determine easily if today's date falls at the end of a month is to set up an array of integers that corresponds to the number of days in each month. A lookup inside the array for a particular month then gives the number of days in that month. So the statement

```
int  daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

defines an array called `daysPerMonth` containing 12 integer elements. For each month `i`, the value contained in `daysPerMonth[i - 1]` corresponds to the number of days in that particular month. Therefore, the number of days in April, which is the fourth month of the year, is given by `daysPerMonth[3]`, which is equal to 30. (You could define the array to contain 13 elements, with `daysPerMonth[i]` corresponding to the number of days in month `i`. Access into the array could then be made directly based on the month number, rather than on the month number minus 1. The decision of whether to use 12 or 13 elements in this case is strictly a matter of personal preference.)

If it is determined that today's date falls at the end of the month, you can calculate tomorrow's date by simply adding 1 to the month number and setting the value of the day equal to `1`.

To solve the second problem mentioned earlier, you must determine if today's date is at the end of a month and if the month is 12. If this is the case, then tomorrow's day and month must be set equal to `1` and the year appropriately incremented by 1.

Program 9.2 asks the user to enter today's date, calculates tomorrow's date, and displays the results.

Program 9.2 **Determining Tomorrow's Date**

```
// Program to determine tomorrow's date

#include <stdio.h>

int main (void)
{
    struct  date
    {
         int  month;
         int  day;
         int  year;
     };

     struct date  today, tomorrow;

     const int  daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                     31, 31, 30, 31, 30, 31 };

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &today.month, &today.day, &today.year);

     if  ( today.day != daysPerMonth[today.month - 1] ) {
         tomorrow.day = today.day + 1;
         tomorrow.month = today.month;
         tomorrow.year = today.year;
     }
```

Program 9.2 **Continued**

```
     else if ( today.month == 12 ) {    // end of year
         tomorrow.day = 1;
         tomorrow.month = 1;
         tomorrow.year = today.year + 1;
     }
     else {                             // end of month
         tomorrow.day = 1;
         tomorrow.month = today.month + 1;
         tomorrow.year = today.year;
     }

     printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
            tomorrow.day, tomorrow.year % 100);

     return 0;
}
```

Program 9.2 **Output**

```
Enter today's date (mm dd yyyy): 12 17 2004
Tomorrow's date is 12/18/04.
```

Program 9.2 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 12 31 2005
Tomorrow's date is 1/1/06.
```

Program 9.2 **Output (Second Rerun)**

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 3/1/04.
```

If you look at the program's output, you quickly notice that there seems to be a mistake somewhere: The day after February 28, 2004 is listed as March 1, 2004 and *not* as February 29, 2004. The program forgot about leap years! You fix this problem in the following section. First, you need to analyze the program and its logic.

After the `date` structure is defined, two variables of type `struct date`, `today` and `tomorrow`, are declared. The program then asks the user to enter today's date. The three integer values that are entered are stored into `today.month`, `today.day`, and `today.year`, respectively. Next, a test is made to determine if the day is at the end of the month, by comparing `today.day` to `daysPerMonth[today.month - 1]`. If it is not the end of the month, tomorrow's date is calculated by simply adding 1 to the day and setting tomorrow's month and year equal to today's month and year.

If today's date does fall at the end of the month, another test is made to determine if it is the end of the year. If the month equals 12, meaning that today's date is December 31, tomorrow's date is set equal to January 1 of the next year. If the month does not equal 12, tomorrow's date is set to the first day of the following month (of the same year).

After tomorrow's date has been calculated, the values are displayed to the user with an appropriate `printf` statement call, and program execution is complete.

Functions and Structures

Now, you can return to the problem that was discovered in the previous program. Your program thinks that February always has 28 days, so naturally when you ask it for the day after February 28, it always displays March 1 as the answer. You need to make a special test for the case of a leap year. If the year is a leap year, and the month is February, the number of days in that month is 29. Otherwise, the normal lookup inside the `daysPerMonth` array can be made.

A good way to incorporate the required changes into Program 9.2 is to develop a function called `numberOfDays` to determine the number of days in a month. The function would perform the leap year test and the lookup inside the `daysPerMonth` array as required. Inside the `main` routine, all that has to be changed is the `if` statement, which compares the value of `today.day` to `daysPerMonth[today.month - 1]`. Instead, you could now compare the value of `today.day` to the value returned by your `numberOfDays` function.

Study Program 9.3 carefully to determine what is being passed to the `numberOfDays` function as an argument.

Program 9.3 **Revising the Program to Determine Tomorrow's Date**

```
// Program to determine tomorrow's date

#include <stdio.h>
#include <stdbool.h>

struct  date
{
    int  month;
    int  day;
    int  year;
};

int main (void)
{
```

Program 9.3 **Continued**

```
     struct date  today, tomorrow;
     int  numberOfDays (struct date d);

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &today.month, &today.day, &today.year);

     if  ( today.day != numberOfDays (today) ) {
         tomorrow.day = today.day + 1;
         tomorrow.month = today.month;
         tomorrow.year = today.year;
     }
     else if ( today.month == 12 ) {    // end of year
         tomorrow.day = 1;
         tomorrow.month = 1;
         tomorrow.year = today.year + 1;
     }
     else {                             // end of month
          tomorrow.day = 1;
          tomorrow.month = today.month + 1;
          tomorrow.year = today.year;
     }

     printf ("Tomorrow's date is %i/%i/%.2i.\n",tomorrow.month,
                 tomorrow.day, tomorrow.year % 100);

     return 0;
}

// Function to find the number of days in a month

int  numberOfDays  (struct date  d)
{
    int   days;
    bool  isLeapYear (struct date  d);
    const int   daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true &&  d.month == 2 )
         days = 29;
    else
         days = daysPerMonth[d.month - 1];

    return days;
}
```

Program 9.3 **Continued**

```
// Function to determine if it's a leap year

bool  isLeapYear (struct date  d)
{
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                 d.year % 400 == 0 )
        leapYearFlag = true;   // It's a leap year
    else
        leapYearFlag = false;  // Not a leap year

    return leapYearFlag;
}
```

Program 9.3 **Output**

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 2/29/04.
```

Program 9.3 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 2 28 2005
Tomorrow's date is 3/1/05.
```

The first thing that catches your eye in the preceding program is the fact that the definition of the `date` structure appears first and outside of any function. This makes the definition known throughout the file. Structure definitions behave very much like variables—if a structure is defined within a particular function, only that function knows of its existence. This is a *local* structure definition. If you define the structure outside of any function, that definition is *global*. A global structure definition allows any variables that are subsequently defined in the program (either inside or outside of a function) to be declared to be of that structure type.

Inside the `main` routine, the prototype declaration

```
int  numberOfDays (struct date d);
```

informs the C compiler that the `numberOfDays` function returns an integer value and takes a single argument of type `struct date`.

Instead of comparing the value of `today.day` against the value `daysPerMonth[today.month - 1]`, as was done in the preceding example, the statement

```
if  ( today.day != numberOfDays (today) )
```

is used. As you can see from the function call, you are specifying that the structure `today` is to be passed as an argument. Inside the `numberOfDays` function, the appropriate declaration must be made to inform the system that a structure is expected as an argument:

```
int  numberOfDays  (struct date  d)
```

As with ordinary variables, and unlike arrays, any changes made by the function to the values contained in a structure argument have no effect on the original structure. They affect only the copy of the structure that is created when the function is called.

The `numberOfDays` function begins by determining if it is a leap year and if the month is February. The former determination is made by calling another function called `isLeapYear`. You learn about this function shortly. From reading the `if` statement

```
if ( isLeapYear (d) == true   &&  d.month == 2 )
```

you can assume that the `isLeapYear` function returns `true` if it is a leap year and returns `false` if it is not a leap year. This is directly in line with our discussions of Boolean variables back in Chapter 6, "Making Decisions." Recall that the standard header file `<stdbool.h>` defines the values `bool`, `true`, and `false` for you, which is why this file is included at the beginning of Program 9.3.

An interesting point to be made about the previous `if` statement concerns the choice of the function name `isLeapYear`. This name makes the `if` statement extremely readable and implies that the function is returning some kind of yes/no answer.

Getting back to the program, if the determination is made that it is February of a leap year, the value of the variable `days` is set to 29; otherwise, the value of `days` is found by indexing the `daysPerMonth` array with the appropriate month. The value of `days` is then returned to the `main` routine, where execution is continued as in Program 9.2.

The `isLeapYear` function is straightforward enough—it simply tests the year contained in the `date` structure given as its argument and returns `true` if it is a leap year and `false` if it is not.

As an exercise in producing a better-structured program, take the entire process of determining tomorrow's date and relegate it to a separate function. You can call the new function `dateUpdate` and have it take as its argument today's date. The function then calculates tomorrow's date and *returns* the new date back to us. Program 9.4 illustrates how this can be done in C.

Program 9.4 **Revising the Program to Determine Tomorrow's Date, Version 2**

```
// Program to determine tomorrow's date

#include <stdio.h>
#include <stdbool.h>

struct  date
{
    int  month;
    int  day;
```

Program 9.4 **Continued**

```
    int  year;
};

// Function to calculate tomorrow's date

struct date  dateUpdate (struct date  today)
{
    struct date  tomorrow;
    int  numberOfDays (struct date  d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 )  {   // end of year
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                             // end of month
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    return tomorrow;
}

// Function to find the number of days in a month

int  numberOfDays  (struct date  d)
{
     int  days;
     bool isLeapYear (struct date  d);
     const int  daysPerMonth[12] =
       { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

     if ( isLeapYear   &&  d.month == 2 )
         days = 29;
     else
         days = daysPerMonth[d.month - 1];

     return days;
}
```

Program 9.4 **Continued**

```
// Function to determine if it's a leap year

bool  isLeapYear (struct date  d)
{
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                 d.year % 400 == 0 )
        leapYearFlag = true;   // It's a leap year
    else
        leapYearFlag = false;  // Not a leap year

    return leapYearFlag;
}

int main (void)
{
     struct date  dateUpdate (struct date  today);
     struct date  thisDay, nextDay;

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &thisDay.month, &thisDay.day,
                &thisDay.year);

     nextDay = dateUpdate (thisDay);

     printf ("Tomorrow's date is %i/%i/%.2i.\n",nextDay.month,
                nextDay.day, nextDay.year % 100);

     return 0;
}
```

Program 9.4 **Output**

```
Enter today's date (mm dd yyyy): 2 28 2008
Tomorrow's date is 2/29/08.
```

Program 9.4 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 2 22 2005
Tomorrow's date is 2/23/05.
```

Inside `main`, the statement

```
next_date = dateUpdate (thisDay);
```

After the time has been entered, the program calls the `timeUpdate` function, passing along the `currentTime` as the argument. The result returned by the function is assigned to the `struct time` variable `nextTime`, which is then displayed with an appropriate `printf` call.

The `timeUpdate` function begins execution by "bumping" the time in `now` by one second. A test is then made to determine if the number of seconds has reached 60. If it has, the seconds are reset to 0 and the minutes are increased by 1. Another test is then made to see if the number of minutes has now reached 60, and if it has, the minutes are reset to 0 and the hour is increased by 1. Finally, if the two preceding conditions are satisfied, a test is then made to see if the hour is equal to 24; that is, if it is precisely midnight. If it is, the hour is reset to 0. The function then returns the value of `now`, which contains the updated time, back to the calling routine.

Initializing Structures

Initializing structures is similar to initializing arrays—the elements are simply listed inside a pair of braces, with each element separated by a comma.

To initialize the `date` structure variable `today` to July 2, 2005, the statement

```
struct date  today = { 7, 2, 2005 };
```

can be used. The statement

```
struct time  this_time = { 3, 29, 55 };
```

defines the `struct time` variable `this_time` and sets its value to 3:29:55 a.m. As with other variables, if `this_time` is a local structure variable, it is initialized each time the function is entered. If the structure variable is made static (by placing the keyword `static` in front of it), it is only initialized once at the start of program execution. In either case, the initial values listed inside the curly braces must be constant expressions.

As with the initialization of an array, fewer values might be listed than are contained in the structure. So the statement

```
struct time  time1 = { 12, 10 };
```

sets `time1.hour` to `12` and `time1.minutes` to `10` but gives no initial value to `time1.seconds`. In such a case, its default initial value is undefined.

You can also specify the member names in the initialization list. In that case, the general format is

```
.member = value
```

This method enables you to initialize the members in any order, or to only initialize specified members. For example,

```
struct time time1 = { .hour = 12, .minutes = 10 };
```

sets the `time1` variable to the same initial values as shown in the previous example. The statement

```
struct date today = { .year = 2004 };
```

sets just the year member of the date structure variable `today` to `2004`.

Compound Literals

You can assign one or more values to a structure in a single statement using what is know as *compound literals*. For example, assuming that `today` has been previously declared as a `struct date` variable, the assignment of the members of `today` as shown in Program 9.1 can also be done in a single statement as follows:

```
today = (struct date) { 9, 25, 2004 };
```

Note that this statement can appear anywhere in the program; it is not a declaration statement. The type cast operator is used to tell the compiler the type of the expression, which in this case is `struct date`, and is followed by the list of values that are to be assigned to the members of the structure, in order. These values are listed in the same way as if you were initializing a structure variable.

You can also specify values using the *`.member`* notation like this:

```
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

The advantage of using this approach is that the arguments can appear in any order. Without explicitly specifying the member names, they must be supplied in the order in which they are defined in the structure.

The following example shows the `dateUpdate` function from Program 9.4 rewritten to take advantage of compound literals:

```
// Function to calculate tomorrow's date – using compound literals

struct date  dateUpdate (struct date  today)
{
    struct date  tomorrow;
    int  numberOfDays (struct date  d);

    if ( today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.month, today.day + 1, today.year };
    else if ( today.month == 12 )      // end of year
        tomorrow = (struct date) { 1, 1, today.year + 1 };
    else                              // end of month
        tomorrow = (struct date) { today.month + 1, 1, today.year };


    return tomorrow;
}
```

Whether you decide to use compound literals in your programs is up to you. In this case, the use of compound literals makes the `dateUpdate` function easier to read.

Compound literals can be used in other places where a valid structure expression is allowed. This is a perfectly valid, albeit totally impractical example of such a use:

```
nextDay =  dateUpdate ((struct date) { 5, 11, 2004} );
```

The `dateUpdate` function expects an argument of type `struct date`, which is precisely the type of compound literal that is supplied as the argument to the function.

Arrays of Structures

You have seen how useful the structure is in enabling you to logically group related elements together. With the `time` structure, for instance, it is only necessary to keep track of one variable, instead of three, for each time that is used by the program. So, to handle 10 different times in a program, you only have to keep track of 10 different variables, instead of 30.

An even better method for handling the 10 different times involves the combination of two powerful features of the C programming language: structures and arrays. C does not limit you to storing simple data types inside an array; it is perfectly valid to define an *array of structures*. For example,

```
struct time  experiments[10];
```

defines an array called `experiments`, which consists of 10 elements. Each element inside the array is defined to be of type `struct time`. Similarly, the definition

```
struct date  birthdays[15];
```

defines the array `birthdays` to contain 15 elements of type `struct date`. Referencing a particular structure element inside the array is quite natural. To set the second birthday inside the `birthdays` array to August 8, 1986, the sequence of statements

```
birthdays[1].month = 8;
birthdays[1].day   = 8;
birthdays[1].year  = 1986;
```

works just fine. To pass the entire `time` structure contained in `experiments[4]` to a function called `checkTime`, the array element is specified:

```
checkTime (experiments[4]);
```

As is to be expected, the `checkTime` function declaration must specify that an argument of type `struct time` is expected:

```
void checkTime (struct time  t0)
{
     .
     .
     .
}
```

Initialization of arrays containing structures is similar to initialization of multidimensional arrays. So the statement

```
struct time  runTime [5] =
     {  {12, 0, 0},  {12, 30, 0},  {13, 15, 0} };
```

sets the first three times in the array `runTime` to 12:00:00, 12:30:00, and 13:15:00. The inner pairs of braces are optional, meaning that the preceding statement can be equivalently expressed as

```
struct time  runTime[5] =
     { 12, 0, 0, 12, 30, 0, 13, 15, 0 };
```

The following statement

```
struct time runTime[5] =
     { [2] = {12, 0, 0} };
```

initializes just the third element of the array to the specified value, whereas the statement

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

sets just the hours and minutes of the second element of the `runTime` array to `12` and `30`, respectively.

Program 9.6 sets up an array of time structures called `testTimes`. The program then calls your `timeUpdate` function from Program 9.5. To conserve space, the `timeUpdate` function is not included in this program listing. However, a comment statement is inserted to indicate where in the program the function could be included.

In Program 9.6, an array called `testTimes` is defined to contain five different times. The elements in this array are assigned initial values that represent the times 11:59:59, 12:00:00, 1:29:59, 23:59:59, and 19:12:27, respectively. Figure 9.2 can help you to understand what the `testTimes` array actually looks like inside the computer's memory. A particular `time` structure stored in the `testTimes` array is accessed by using the appropriate index number 0–4. A particular member (`hour`, `minutes`, or `seconds`) is then accessed by appending a period followed by the member name.

For each element in the `testTimes` array, Program 9.6 displays the time as represented by that element, calls the `timeUpdate` function from Program 9.5, and then displays the updated time.

Program 9.6 **Illustrating Arrays of Structures**

```
//  Program to illustrate arrays of structures

#include <stdio.h>

struct  time
{
```

Program 9.6 **Continued**

```
    int  hour;
    int  minutes;
    int  seconds;
};

int main (void)
{
    struct time  timeUpdate (struct time  now);
    struct time  testTimes[5] =
        {  { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
           { 23, 59, 59 }, { 19, 12, 27 }};
    int  i;

    for ( i = 0;  i < 5;  ++i )  {
        printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
            testTimes[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate (testTimes[i]);

        printf (" ...one second later it's %.2i:%.2i:%.2i\n",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);
    }

    return 0;
}

//  ***** Include the timeUpdate function here  *****
```

Program 9.6 **Output**

```
Time is 11:59:59 ...one second later it's 12:00:00
Time is 12:00:00 ...one second later it's 12:00:01
Time is 01:29:59 ...one second later it's 01:30:00
Time is 23:59:59 ...one second later it's 00:00:00
Time is 19:12:27 ...one second later it's 19:12:28
```

The concept of an array of structures is a very powerful and important one in C. Make certain you understand it fully before you move on.

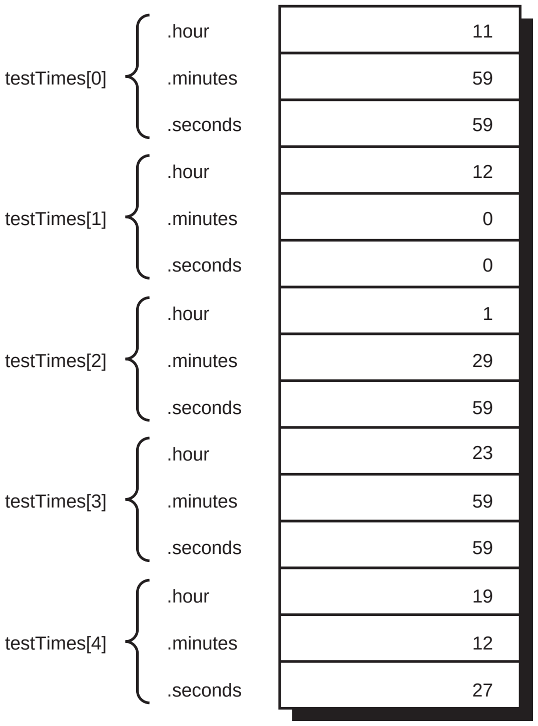

Figure 9.2 The array `testTimes` in memory.

Structures Containing Structures

C provides you with an enormous amount of flexibility in defining structures. For instance, you can define a structure that itself contains other structures as one or more of its members, or you can define structures that contain arrays.

You have seen how it is possible to logically group the month, day, and year into a structure called `date` and how to group the hour, minutes, and seconds into a structure called `time`. In some applications, you might have the need to logically group both a date and a time together. For example, you might need to set up a list of events that are to occur at a particular date and time.

What the preceding discussion implies is that you want to have a convenient means for associating *both* the date and the time together. You can do this in C by defining a new structure, called, for example, `dateAndTime`, which contains as its members two elements: the date and the time.

```
struct dateAndTime
{
    struct date   sdate;
    struct time   stime;
};
```

The first member of this structure is of type `struct date` and is called `sdate`. The second member of the `dateAndTime` structure is of type `struct time` and is called `stime`. This definition of a `dateAndTime` structure requires that a `date` structure and a `time` structure have been previously defined to the compiler.

Variables can now be defined to be of type `struct dateAndTime`, as in

```
struct dateAndTime  event;
```

To reference the `date` structure of the variable `event`, the syntax is the same:

```
event.sdate
```

So, you could call your `dateUpdate` function with this date as the argument and assign the result back to the same place by a statement such as

```
event.sdate = dateUpdate (event.sdate);
```

You can do the same type of thing with the `time` structure contained within your `dateAndTime` structure:

```
event.stime = timeUpdate (event.stime);
```

To reference a particular member *inside* one of these structures, a period followed by the member name is tacked on the end:

```
event.sdate.month = 10;
```

This statement sets the `month` of the `date` structure contained within `event` to October, and the statement

```
++event.stime.seconds;
```

adds one to the `seconds` contained within the `time` structure.

The `event` variable can be initialized in the expected manner:

```
struct dateAndTime  event =
          { { 2, 1, 2004 }, { 3, 30, 0 } };
```

This sets the date in the variable `event` to February 1, 2004, and sets the time to 3:30:00.

Of course, you can use members' names in the initialization, as in

```
struct dateAndTime event =
          { { .month = 2, .day = 1, .year = 2004 },
            { .hour = 3, .minutes = 30, .seconds = 0 }
          };
```

Naturally, it is possible to set up an array of `dateAndTime` structures, as is done with the following declaration:

```
struct dateAndTime  events[100];
```

The array `events` is declared to contain 100 elements of type `struct dateAndTime`. The fourth `dateAndTime` contained within the array is referenced in the usual way as `events[3]`, and the *i*th date in the array can be sent to your `dateUpdate` function as follows:

```
events[i].sdate = dateUpdate (events[i].sdate);
```

To set the first time in the array to noon, the series of statements

```
events[0].stime.hour    = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

can be used.

Structures Containing Arrays

As the heading of this section implies, it is possible to define structures that contain arrays as members. One of the most common applications of this type is setting up an array of characters inside a structure. For example, suppose you want to define a structure called `month` that contains as its members the number of days in the month as well as a three-character abbreviation for the month name. The following definition does the job:

```
struct  month
{
    int    numberOfDays;
    char   name[3];
};
```

This sets up a `month` structure that contains an integer member called `numberOfDays` and a character member called `name`. The member `name` is actually an array of three characters. You can now define a variable to be of type `struct month` in the normal fashion:

```
struct month  aMonth;
```

You can set the proper fields inside `aMonth` for January with the following sequence of statements:

```
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

Or, you can initialize this variable to the same values with the following statement:

```
struct month  aMonth = { 31, { 'J', 'a', 'n' } };
```

To go one step further, you can set up 12 month structures inside an array to represent each month of the year:

```
struct month  months[12];
```

Program 9.7 illustrates the `months` array. Its purpose is simply to set up the initial values inside the array and then display these values at the terminal.

It might be easier for you to conceptualize the notation that is used to reference particular elements of the `months` array as defined in the program by examining Figure 9.3.

Program 9.7 **Illustrating Structures and Arrays**

```
// Program to illustrate structures and arrays

#include <stdio.h>

int main (void)
{
    int  i;

    struct  month
    {
        int    numberOfDays;
        char   name[3];
    };

    const struct month  months[12] =
     { { 31, {'J', 'a', 'n'} },  { 28, {'F', 'e', 'b'} },
       { 31, {'M', 'a', 'r'} },  { 30, {'A', 'p', 'r'} },
       { 31, {'M', 'a', 'y'} },  { 30, {'J', 'u', 'n'} },
       { 31, {'J', 'u', 'l'} },  { 31, {'A', 'u', 'g'} },
       { 30, {'S', 'e', 'p'} },  { 31, {'O', 'c', 't'} },
       { 30, {'N', 'o', 'v'} },  { 31, {'D', 'e', 'c'} } };

     printf ("Month    Number of Days\n");
     printf ("-----    --------------\n");

     for ( i = 0;  i < 12;  ++i )
         printf (" %c%c%c           %i\n",
             months[i].name[0], months[i].name[1],
             months[i].name[2], months[i].numberOfDays);

     return 0;
}
```

Program 9.7 **Output**

```
Month   Number of Days
-----   ---------------
 Jan          31
 Feb          28
 Mar          31
 Apr          30
 May          31
 Jun          30
 Jul          31
 Aug          31
 Sep          30
 Oct          31
 Nov          30
 Dec          31
```

As you can see in Figure 9.3, the notation

```
months[0]
```

refers to the *entire* `month` structure contained in the first location of the `months` array. The type of this expression is `struct month`. Therefore, when passing `months[0]` to a function as an argument, the corresponding formal parameter inside the function must be declared to be of type `struct month`.

Going one step further, the expression

```
months[0].numberOfDays
```

refers to the `numberOfDays` member of the `month` structure contained in `months[0]`. The type of this expression is `int`. The expression

```
months[0].name
```

references the three-character array called `name` inside the `month` structure of `months[0]`. If passing this expression as an argument to a function, the corresponding formal parameter is declared to be an array of type `char`.

Finally, the expression

```
months[0].name[0]
```

references the first character of the `name` array contained in `months[0]` (the character `'J'`).

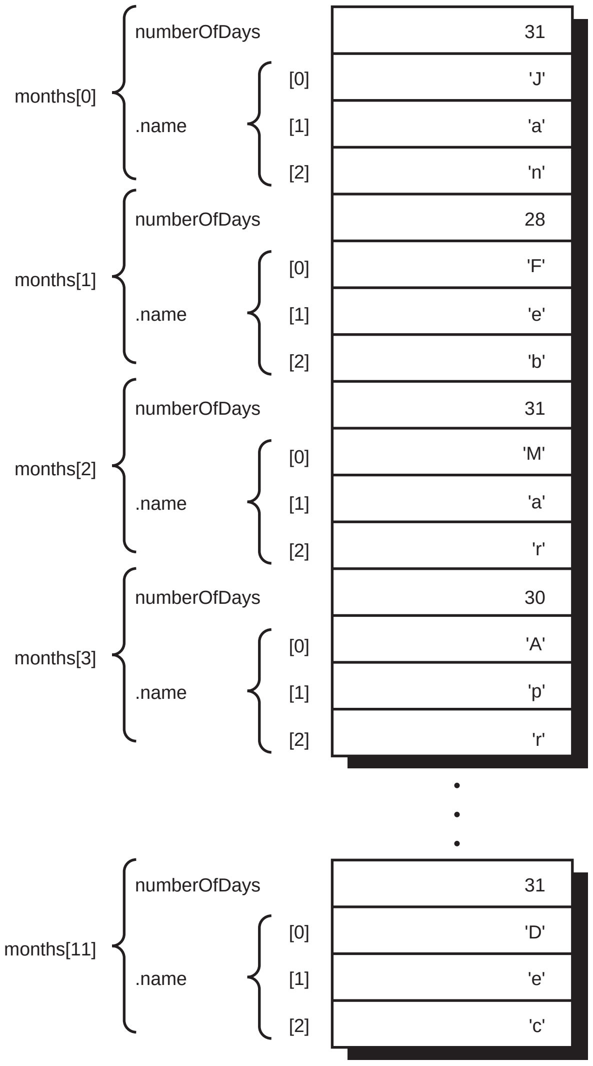

Figure 9.3 The array `months`.

Structure Variants

You do have some flexibility in defining a structure. First, it is valid to declare a variable to be of a particular structure type at the same time that the structure is defined. This is done simply by including the variable name (or names) before the terminating semicolon of the structure definition. For example, the statement

11

Pointers

IN THIS CHAPTER, YOU EXAMINE one of the most sophisticated features of the C programming language: *pointers*. In fact, the power and flexibility that C provides in dealing with pointers serve to set it apart from many other programming languages. Pointers enable you to effectively represent complex data structures, to change values passed as arguments to functions, to work with memory that has been allocated "dynamically" (see Chapter 17, "Miscellaneous and Advanced Features"), and to more concisely and efficiently deal with arrays.

To understand the way in which pointers operate, it is first necessary to understand the concept of *indirection*. You are familiar with this concept from your everyday life. For example, suppose you need to buy a new ink cartridge for your printer. In the company that you work for, all purchases are handled by the Purchasing department. So, you call Jim in Purchasing and ask him to order the new cartridge for you. Jim, in turn, calls the local supply store to order the cartridge. This approach to obtain your new cartridge is actually an indirect one because you are not ordering the cartridge directly from the supply store yourself.

This same notion of indirection applies to the way pointers work in C. A pointer provides an indirect means of accessing the value of a particular data item. And just as there are reasons why it makes sense to go through the Purchasing department to order new cartridges (you don't have to know which particular store the cartridges are being ordered from, for example), so are there good reasons why, at times, it makes sense to use pointers in C.

a way (noun)

Defining a Pointer Variable

But enough talk—it's time to see how pointers actually work. Suppose you define a variable called `count` as follows:

```
int  count = 10;
```

Un puntero es un intermediario, como el señor de recursos humanos que ayuda con las compras, entonces la variable de declaracion de la variable y su puntero deben ser iguales, sin embargo puede que se permita que se declaren distinto aunque esto sea incorrecto y peligroso

You can define another variable, called `int_pointer`, that can be used to enable you to indirectly access the value of `count` by the declaration

```
int  *int_pointer;
```

The asterisk defines to the C system that the variable `int_pointer` is of type *pointer to* `int`. This means that `int_pointer` is used in the program to indirectly access the value of one or more integer values. toda variable tiene una dirección de memoria pero no un puntero asociado

You have seen how the `&` operator was used in the `scanf` calls of previous programs. This unary operator, known as the *address* operator, is used to make a pointer to an object in C. So, if `x` is a variable of a particular type, the expression `&x` is a pointer to that variable. The expression `&x` can be assigned to any pointer variable, if desired, that has been declared to be a pointer to the same type as `x`.

Therefore, with the definitions of `count` and `int_pointer` as given, you can write a statement such as

```
int_pointer = &count;
```

relación entre el puntero int_pointer y &count

to set up the indirect reference between `int_pointer` and `count`. The address operator has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a *pointer* to the variable `count`. The link that has been made between `int_pointer` and `count` is conceptualized in Figure 11.1. The directed line illustrates the idea that `int_pointer` does not directly contain the value of `count`, but a pointer to the variable `count`. Un puntero permite escribir codigo que oepera sobre memoria, no sobre nombres

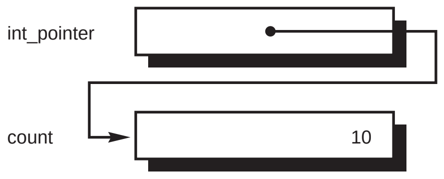

Figure 11.1 Pointer to an integer.

To reference the contents of `count` through the pointer variable `int_pointer`, you use the *indirection* operator, which is the asterisk `*`. So, if `x` is defined as type `int`, the statement

```
x = *int_pointer;
```

assigns the value that is indirectly referenced through `int_pointer` to the variable `x`. Because `int_pointer` was previously set pointing to `count`, this statement has the effect of assigning the value contained in the variable `count`—which is `10`—to the variable `x`.

The previous statements have been incorporated into Program 11.1, which illustrates the two fundamental pointer operators: the address operator, `&`, and the indirection operator, `*`.

La funcion recibe copias del valor, no acceso a la memoria original, entonces las copias que se generan dentro de la función no modifican la variable original (no puedo modificar una variable desde una funcion sino trabajo con punteros)

Al modificar una dirección estamos modificando directamente la memoria (y por lo tanto somos capaces de modificar variables desde funciones

Program 11.1 **Illustrating Pointers**

```
// Program to illustrate pointers

#include <stdio.h>

int main (void)
{
     int   count = 10, x;
     int   *int_pointer;

     int_pointer = &count;
     x = *int_pointer;

     printf ("count = %i, x = %i\n", count, x);

     return 0;
}
```

en resumen los punteros permiten modificar el valor de una variable, esto se debe a porque estamos tocando directamente la direccion de la variable, y esto permite modificar variables desde dentro de una función

“No quiero ser mejor que otros que no juegan el mismo juego.Quiero saber hasta dónde llego cuando el juego es limpio.”

Program 11.1 **Output**

```
count = 10, x = 10
```

The variables `count` and `x` are declared to be integer variables in the normal fashion. On the next line, the variable `int_pointer` is declared to be of type “pointer to `int`.” Note that the two lines of declarations could have been combined into the single line

```
int  count = 10, x, *int_pointer;
```

Next, the address operator is applied to the variable `count`. This has the effect of creating a pointer to this variable, which is then assigned by the program to the variable `int_pointer`.

Execution of the next statement in the program,

```
x = *int_pointer;
```

proceeds as follows: The indirection operator tells the C system to treat the variable `int_pointer` as containing a pointer to another data item. This pointer is then used to access the desired data item, whose type is specified by the declaration of the pointer variable. Because you told the compiler that `int_pointer` points to integers when you declared the variable, the compiler knows that the value referenced by the expression `*int_pointer` is an integer. And because you set `int_pointer` to point to the integer variable `count` in the previous program statement, it is the value of `count` that is indirectly accessed by this expression.

You should realize that Program 11.1 is a manufactured example of the use of pointers and does not show a practical use for them in a program. Such motivation is presented shortly, after you have become familiar with the basic ways in which pointers can be defined and manipulated in a program.

Program 11.2 illustrates some interesting properties of pointer variables. Here, a pointer to a character is used.

Program 11.2 **More Pointer Basics**

```
// Further examples of pointers

#include <stdio.h>

int main (void)
{
     char  c = 'Q';
     char  *char_pointer = &c;

     printf ("%c %c\n", c, *char_pointer);

     c = '/';
     printf ("%c %c\n", c, *char_pointer);

     *char_pointer = '(';
     printf ("%c %c\n", c, *char_pointer);

     return 0;
}
```

Program 11.2 **Output**

```
Q Q
/ /
( (
```

The character variable `c` is defined and initialized to the character `'Q'`. In the next line of the program, the variable `char_pointer` is defined to be of type "pointer to `char`," meaning that whatever value is stored inside this variable should be treated as an indirect reference (pointer) to a character. Notice that you can assign an initial value to this variable in the normal fashion. The value that you assign to `char_pointer` in the program is a pointer to the variable `c`, which is obtained by applying the address operator to the variable `c`. (Note that this initialization generates a compiler error if `c` had been defined *after* this statement because a variable must always be declared *before* its value can be referenced in an expression.)

The declaration of the variable `char_pointer` and the assignment of its initial value could have been equivalently expressed in two separate statements as

```
char  *char_pointer;
char_pointer = &c;
```

(and *not* by the statements

```
char  *char_pointer;
*char_pointer = &c;
```

as might be implied from the single-line declaration).

Always remember, that the value of a pointer in C is meaningless until it is set pointing to something.

The first `printf` call simply displays the contents of the variable `c` and the contents of the variable that is referenced by `char_pointer`. Because you set `char_pointer` to point to the variable `c`, the value that is displayed is the contents of `c`, as verified by the first line of the program's output.

In the next line of the program, the character `'/'` is assigned to the character variable `c`. Because `char_pointer` still points to the variable `c`, displaying the value of `*char_pointer` in the subsequent `printf` call correctly displays this new value of `c` at the terminal. This is an important concept. Unless the value of `char_pointer` is changed, the expression `*char_pointer` *always* accesses the value of `c`. So, as the value of `c` changes, so does the value of `*char_pointer`.

The previous discussion can help you to understand how the program statement that appears next in the program works. Unless `char_pointer` is changed, the expression `*char_pointer` always references the value of `c`. Therefore, in the expression

```
*char_pointer = '(';
```

you are assigning the left parenthesis character to `c`. More formally, the character `'('` is assigned to the variable that is pointed to by `char_pointer`. You know that this variable is `c` because you placed a pointer to `c` in `char_pointer` at the beginning of the program.

The preceding concepts are the key to your understanding of the operation of pointers. Please review them at this point if they still seem a bit unclear.

Using Pointers in Expressions

In Program 11.3, two integer pointers, `p1` and `p2`, are defined. Notice how the value referenced by a pointer can be used in an arithmetic expression. If `p1` is defined to be of type "pointer to integer," what conclusion do you think can be made about the use of `*p1` in an expression?

Program 11.3 **Using Pointers in Expressions**

```
// More on pointers

#include <stdio.h>

int main (void)
{
```

Program 11.3 **Continued**

```
    int  i1, i2;
    int  *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

    return 0;
}
```

Program 11.3 **Output**

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

After defining the integer variables `i1` and `i2` and the integer pointer variables `p1` and `p2`, the program then assigns the value `5` to `i1` and stores a pointer to `i1` inside `p1`. Next, the value of `i2` is calculated with the following expression:

```
i2 = *p1 / 2 + 10;
```

In As implied from the discussions of Program 11.2, if a pointer `px` points to a variable `x`, and `px` has been defined to be a pointer to the same data type as is `x`, then use of `*px` in an expression is, in all respects, identical to the use of `x` in the same expression.

Because in Program 11.3 the variable `p1` is defined to be an integer pointer, the preceding expression is evaluated using the rules of integer arithmetic. And because the value of `*p1` is `5` (`p1` points to `i1`), the final result of the evaluation of the preceding expression is `12`, which is the value that is assigned to `i2`. (The pointer reference operator `*` has higher precedence than the arithmetic operation of division. In fact, this operator, as well as the address operator `&`, has higher precedence than *all* binary operators in C.)

In the next statement, the value of the pointer `p1` is assigned to `p2`. This assignment is perfectly valid and has the effect of setting `p2` to point to the same data item to which `p1` points. Because `p1` points to `i1`, after the assignment statement has been executed, `p2` *also* points to `i1` (and you can have as many pointers to the same item as you want in C).

The `printf` call verifies that the values of `i1`, `*p1`, and `*p2` are all the same (`5`) and that the value of `i2` was set to `12` by the program.

Working with Pointers and Structures

You have seen how a pointer can be defined to point to a basic data type, such as an `int` or a `char`. But pointers can also be defined to point to structures. In Chapter 9, "Working with Structures," you defined your `date` structure as follows:

```
struct date
{
     int  month;
     int  day;
     int  year;
};
```

Just as you defined variables to be of type `struct date`, as in

```
struct date   todaysDate;
```

so can you define a variable to be a pointer to a `struct date` variable:

```
struct date  *datePtr;
```

The variable `datePtr`, as just defined, then can be used in the expected fashion. For example, you can set it to point to `todaysDate` with the assignment statement

```
datePtr = &todaysDate;
```

After such an assignment has been made, you then can indirectly access any of the members of the `date` structure pointed to by `datePtr` in the following way:

```
(*datePtr).day = 21;
```

This statement has the effect of setting the day of the `date` structure pointed to by `datePtr` to 21. The parentheses are required because the structure member operator `.` has higher precedence than the indirection operator `*`.

To test the value of `month` stored in the `date` structure pointed to by `datePtr`, a statement such as

```
if  ( (*datePtr).month == 12  )
          ...
```

can be used.

Pointers to structures are so often used in C that a special operator exists in the language. The structure pointer operator `->`, which is the dash followed by the greater than sign, permits expressions that would otherwise be written as

```
(*x).y
```

to be more clearly expressed as

```
x->y
```

So, the previous `if` statement can be conveniently written as

```
if  ( datePtr->month == 12 )
     ...
```

Program 9.1, the first program that illustrated structures, was rewritten using the concept of structure pointers, as shown in Program 11.4.

Program 11.4 **Using Pointers to Structures**

```
//  Program to illustrate structure pointers

#include <stdio.h>

int main (void)
{
     struct date
     {
          int  month;
          int  day;
          int  year;
     };

     struct date  today, *datePtr;

     datePtr = &today;

     datePtr->month = 9;
     datePtr->day = 25;
     datePtr->year = 2004;

     printf ("Today's date is %i/%i/%.2i.\n",
               datePtr->month, datePtr->day, datePtr->year % 100);

     return 0;
}
```

Program 11.4 **Output**

```
Today's date is 9/25/04.
```

Figure 11.2 depicts how the variables `today` and `datePtr` would look after all of the assignment statements from the preceding program have been executed.

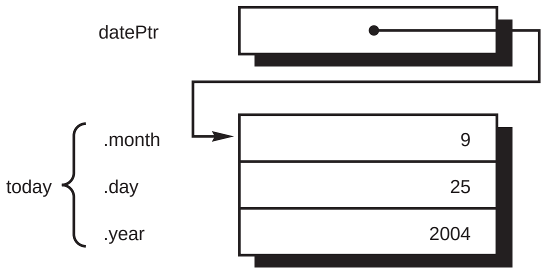

Figure 11.2 Pointer to a structure.

Once again, it should be pointed out that there is no real motivation shown here as to why you should even bother using a structure pointer when it seems as though you can get along just fine without it (as you did in Program 9.1). You will discover the motivation shortly.

Structures Containing Pointers

Naturally, a pointer also can be a member of a structure. In the structure definition

```
struct  intPtrs
{
     int  *p1;
     int  *p2;
};
```

a structure called `intPtrs` is defined to contain two integer pointers, the first one called `p1` and the second one `p2`. You can define a variable of type `struct intPtrs` in the usual way:

```
struct intPtrs  pointers;
```

The variable `pointers` can now be used in the normal fashion, remembering that `pointers` itself is *not* a pointer, but a structure variable that has two pointers as its members.

Program 11.5 shows how the `intPtrs` structure can be handled in a C program.

Program 11.5 **Using Structures Containing Pointers**

```
// Function to use structures containing pointers

#include <stdio.h>

int main (void)
{
     struct  intPtrs
     {
          int  *p1;
          int  *p2;
     };

     struct intPtrs  pointers;
     int  i1 = 100, i2;

     pointers.p1 = &i1;
     pointers.p2 = &i2;
     *pointers.p2 = -97;
```

Program 11.5 **Continued**

```
    printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
    printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
    return 0;
}
```

Program 11.5 **Output**

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

After the variables have been defined, the assignment statement

```
pointers.p1 = &i1;
```

sets the `p1` member of `pointers` pointing to the integer variable `i1`, whereas the next statement

```
pointers.p2 = &i2;
```

sets the `p2` member pointing to `i2`. Next, `-97` is assigned to the variable that is pointed to by `pointers.p2`. Because you just set this to point to `i2`, `-97` is stored in `i2`. No parentheses are needed in this assignment statement because, as mentioned previously, the structure member operator `.` has higher precedence than the indirection operator. Therefore, the pointer is correctly referenced from the structure *before* the indirection operator is applied. Of course, parentheses could have been used just to play it safe, as at times it can be difficult to try to remember which of two operators has higher precedence.

The two `printf` calls that follow each other in the preceding program verify that the correct assignments were made.

Figure 11.3 has been provided to help you understand the relationship between the variables `i1`, `i2`, and `pointers` after the assignment statements from Program 11.5 have been executed. As you can see in Figure 11.3, the `p1` member points to the variable `i1`, which contains the value `100`, whereas the `p2` member points to the variable `i2`, which contains the value `-97`.

Linked Lists

The concepts of pointers to structures and structures containing pointers are very powerful ones in C, for they enable you to create sophisticated data structures, such as *linked lists*, *doubly linked lists*, and *trees*.

Suppose you define a structure as follows:

```
struct entry
{
    int            value;
    struct entry   *next;
};
```

Pointers and Functions

Pointers and functions get along quite well together. That is, you can pass a pointer as an argument to a function in the normal fashion, and you can also have a function return a pointer as its result.

The first case cited previously, passing pointer arguments, is straightforward enough: The pointer is simply included in the list of arguments to the function in the normal fashion. So, to pass the pointer `list_pointer` from the previous program to a function called `print_list`, you can write

```
print_list (list_pointer);
```

Inside the `print_list` routine, the formal parameter must be declared to be a pointer to the appropriate type:

```
void  print_list  (struct entry  *pointer)
{
     ...
}
```

The formal parameter `pointer` can then be used in the same way as a normal pointer variable. One thing worth remembering when dealing with pointers that are sent to functions as arguments: The value of the pointer is copied into the formal parameter when the function is called. Therefore, any change made to the formal parameter by the function does *not* affect the pointer that was passed to the function. But here's the catch: Although the pointer cannot be changed by the function, the data elements that the pointer references *can* be changed! Program 11.8 helps clarify this point.

Program 11.8 **Using Pointers and Functions**

```
// Program to illustrate using pointers and functions

#include <stdio.h>

void test (int  *int_pointer)
{
     *int_pointer = 100;
}

int main (void)
{
     void test (int  *int_pointer);
     int  i = 50, *p = &i;

     printf ("Before the call to test i = %i\n", i);
```

Program 11.8 **Continued**

```
     test (p);
     printf ("After the call to test i = %i\n", i);

     return 0;
}
```

Program 11.8 **Output**

```
Before the call to test i = 50
After the call to test i = 100
```

The function `test` is defined to take as its argument a pointer to an integer. Inside the function, a single statement is executed to set the integer pointed to by `int_pointer` to the value `100`.

The `main` routine defines an integer variable `i` with an initial value of `50` and a pointer to an integer called `p` that is set to point to `i`. The program then displays the value of `i` and calls the `test` function, passing the pointer `p` as the argument. As you can see from the second line of the program's output, the `test` function did, in fact, change the value of `i` to `100`.

Now consider Program 11.9.

Program 11.9 **Using Pointers to Exchange Values**

```
// More on pointers and functions

#include <stdio.h>

void  exchange (int * const pint1, int * const pint2)
{
     int  temp;

     temp = *pint1;
     *pint1 = *pint2;
     *pint2 = temp;
}

int main (void)
{
     void  exchange (int * const pint1, int * const pint2);
     int   i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

     printf ("i1 = %i, i2 = %i\n", i1, i2);
```

Program 11.9 **Continued**

```
     exchange (p1, p2);
     printf ("i1 = %i, i2 = %i\n", i1, i2);

     exchange (&i1, &i2);
     printf ("i1 = %i, i2 = %i\n", i1, i2);

     return 0;
}
```

Program 11.9 **Output**

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

The purpose of the `exchange` function is to interchange the two integer values pointed to by its two arguments. The function header

```
void  exchange (int * const pint1, int * const pint2)
```

says that the `exchange` function takes two integer pointers as arguments, and that the pointers will not be changed by the function (the use of the keyword `const`).

The local integer variable `temp` is used to hold one of the integer values while the exchange is made. Its value is set equal to the integer that is pointed to by `pint1`. The integer pointed to by `pint2` is then copied into the integer pointed to by `pint1`, and the value of `temp` is then stored in the integer pointed to by `pint2`, thus making the exchange complete.

The `main` routine defines integers `i1` and `i2` with values of `-5` and `66`, respectively. Two integer pointers, `p1` and `p2`, are then defined and are set to point to `i1` and `i2`, respectively. The program then displays the values of `i1` and `i2` and calls the `exchange` function, passing the two pointers, `p1` and `p2`, as arguments. The `exchange` function exchanges the value contained in the integer pointed to by `p1` with the value contained in the integer pointed to by `p2`. Because `p1` points to `i1`, and `p2` to `i2`, the values of `i1` and `i2` end up getting exchanged by the function. The output from the second `printf` call verifies that the exchange worked properly.

The second call to `exchange` is a bit more interesting. This time, the arguments that are passed to the function are pointers to `i1` and `i2` that are manufactured right on the spot by applying the address operator to these two variables. Because the expression `&i1` produces a pointer to the integer variable `i1`, this is right in line with the type of argument that your function expects for the first argument (a pointer to an integer). The same applies for the second argument as well. And as can be seen from the program's output, the `exchange` function did its job and switched the values of `i1` and `i2` back to their original values.

You should realize that without the use of pointers, you could not have written your `exchange` function to exchange the value of two integers because you are limited to returning only a single value from a function and because a function cannot permanently change the value of its arguments. Study Program 11.9 in detail. It illustrates with a small example the key concepts to be understood when dealing with pointers in C.

Program 11.10 shows how a function can return a pointer. The program defines a function called `findEntry` whose purpose is to search through a linked list to find a specified value. When the specified value is found, the program returns a pointer to the entry in the list. If the desired value is not found, the program returns the null pointer.

Program 11.10 **Returning a Pointer from a Function**

```
#include <stdio.h>

struct entry
{
     int  value;
     struct entry   *next;
};

struct entry  *findEntry (struct entry  *listPtr, int match)
{
     while  ( listPtr != (struct entry *) 0 )
          if ( listPtr->value == match )
               return (listPtr);
          else
               listPtr = listPtr->next;

     return (struct entry *) 0;
}

int main (void)
{
    struct entry  *findEntry (struct entry  *listPtr, int match);
    struct entry  n1, n2, n3;
    struct entry  *listPtr, *listStart = &n1;

    int search;

    n1.value = 100;
    n1.next =  &n2;

    n2.value = 200;
    n2.next =  &n3;

    n3.value = 300;
    n3.next =  0;
```

Program 11.10 **Continued**

```
    printf ("Enter value to locate: ");
    scanf ("%i", &search);

    listPtr = findEntry (listStart, search);

    if ( listPtr != (struct entry *) 0 )
        printf ("Found %i.\n", listPtr->value);
    else
        printf ("Not found.\n");

    return 0;
}
```

Program 11.10 **Output**

```
Enter value to locate: 200
Found 200.
```

Program 11.10 **Output (Rerun)**

```
Enter value to locate: 400
Not found.
```

Program 11.10 **Output (Second Rerun)**

```
Enter value to locate: 300
Found 300.
```

The function header

```
struct entry  *findEntry (struct entry  *listPtr, int match)
```

specifies that the function `findEntry` returns a pointer to an `entry` structure and that it takes such a pointer as its first argument and an integer as its second. The function begins by entering a `while` loop to sequence through the elements of the list. This loop is executed until either `match` is found equal to one of the `value` entries in the list (in which case the value of `listPtr` is immediately returned) or until the null pointer is reached (in which case the `while` loop is exited and a null pointer is returned).

After setting up the list as in previous programs, the `main` routine asks the user for a value to locate in the list and then calls the `findEntry` function with a pointer to the start of the list (`listStart`) and the value entered by the user (`search`) as arguments. The pointer that is returned by `findEntry` is assigned to the `struct entry` pointer variable `listPtr`. If `listPtr` is not null, the `value` member pointed to by `listPtr` is displayed. This should be the same as the value entered by the user. If `listPtr` is null, then a "Not found." message is displayed.

The program's output verifies that the values `200` and `300` were correctly located in the list, and the value `400` was not found because it did not, in fact, exist in the list.

The pointer that is returned by the `findEntry` function in the program does not seem to serve any useful purpose. However, in more practical situations, this pointer might be used to access other elements contained in the particular entry of the list. For example, you could have a linked list of your dictionary entries from Chapter 10, "Character Strings." Then, you could call the `findEntry` function (or rename it `lookup` as it was called in that chapter) to search the linked list of dictionary entries for the given word. The pointer returned by the `lookup` function could then be used to access the `definition` member of the entry.

Organizing the dictionary as a linked list has several advantages. Inserting a new word into the dictionary is easy: After determining where in the list the new entry is to be inserted, it can be done by simply adjusting some pointers, as illustrated earlier in this chapter. Removing an entry from the dictionary is also simple. Finally, as you learn in Chapter 17, this approach also provides the framework that enables you to dynamically expand the size of the dictionary.

However, the linked list approach for the organization of the dictionary does suffer from one major drawback: You cannot apply your fast binary search algorithm to such a list. This algorithm only works with an array of elements that can be directly indexed. Unfortunately, there is no faster way to search your linked list other than by a straight, sequential search because each entry in the list can only be accessed from the previous one.

One way to glean the benefits of easy insertion and removal of elements, as well as fast search time, is by using a different type of data structure known as a *tree*. Other approaches, such as using *hash tables*, are also feasible. The reader is respectfully referred elsewhere—such as to *The Art of Computer Programming, Volume 3, Sorting and Searching* (Donald E. Knuth, Addison-Wesley)—for discussion of these types of data structures, which can be easily implemented in C with the techniques already described.

Pointers and Arrays

One of the most common uses of pointers in C is as pointers to arrays. The main reasons for using pointers to arrays are ones of notational convenience and of program efficiency. Pointers to arrays generally result in code that uses less memory and executes faster. The reason for this will become apparent through our discussions in this section.

If you have an array of 100 integers called `values`, you can define a pointer called `valuesPtr`, which can be used to access the integers contained in this array with the statement

```
int  *valuesPtr;
```

When you define a pointer that is used to point to the elements of an array, you don't designate the pointer as type "pointer to array"; rather, you designate the pointer as pointing to the type of element that is contained in the array.

If you have an array of characters called `text`, you could similarly define a pointer to be used to point to elements in `text` with the statement

```
char  *textPtr;
```

To set `valuesPtr` to point to the first element in the `values` array, you simply write

```
valuesPtr = values;
```

The address operator is not used in this case because the C compiler treats the appearance of an array name without a subscript as a pointer to the array. Therefore, simply specifying `values` without a subscript has the effect of producing a pointer to the first element of `values` (see Figure 11.9).

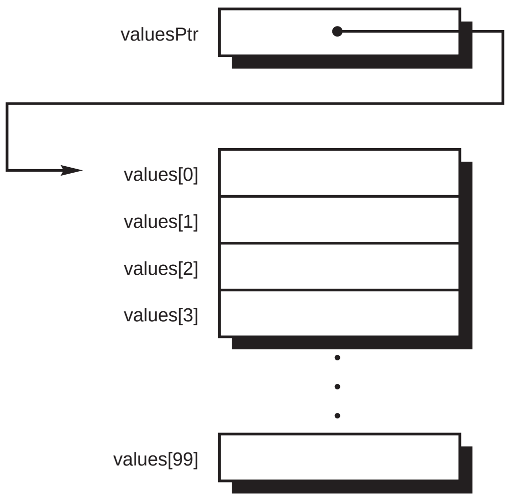

Figure 11.9 Pointer to an array element.

An equivalent way of producing a pointer to the start of `values` is to apply the address operator to the first element of the array. Thus, the statement

```
valuesPtr = &values[0];
```

can be used to serve the same purpose as placing a pointer to the first element of `values` in the pointer variable `valuesPtr`.

To set textPtr to point to the first character inside the text array, either the statement

```
textPtr = text;
```

or

```
textPtr = &text[0];
```

can be used. Whichever statement you choose to use is strictly a matter of taste.

The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array. If valuesPtr is as previously defined and is set pointing to the first element of values, the expression

```
*valuesPtr
```

can be used to access the first integer of the values array, that is, values[0]. To reference values[3] through the valuesPtr variable, you can add 3 to valuesPtr and then apply the indirection operator:

```
*(valuesPtr + 3)
```

In general, the expression

```
*(valuesPtr + i)
```

can be used to access the value contained in values[i].

So, to set values[10] to 27, you could obviously write the expression

```
values[10] = 27;
```

or, using valuesPtr, you could write

```
*(valuesPtr + 10) = 27;
```

To set valuesPtr to point to the second element of the values array, you can apply the address operator to values[1] and assign the result to valuesPtr:

```
valuesPtr = &values[1];
```

If valuesPtr points to values[0], you can set it to point to values[1] by simply adding 1 to the value of valuesPtr:

```
valuesPtr += 1;
```

This is a perfectly valid expression in C and can be used for pointers to *any* data type.

So, in general, if a is an array of elements of type x, px is of type "pointer to x," and i and n are integer constants or variables, the statement

```
px = a;
```

sets px to point to the first element of a, and the expression

```
*(px + i)
```

subsequently references the value contained in a[i]. Furthermore, the statement

```
px += n;
```

sets `px` to point `n` elements farther in the array, *no matter what type of element is contained in the array*.

The increment and decrement operators `++` and `--` are particularly handy when dealing with pointers. Applying the increment operator to a pointer has the same effect as adding one to the pointer, while applying the decrement operator has the same effect as subtracting one from the pointer. So, if `textPtr` is defined as a `char` pointer and is set pointing to the beginning of an array of `char`s called `text`, the statement

```
++textPtr;
```

sets `textPtr` pointing to the next character in `text`, which is `text[1]`. In a similar fashion, the statement

```
--textPtr;
```

sets `textPtr` pointing to the previous character in `text`, assuming, of course, that `textPtr` was not pointing to the beginning of `text` prior to the execution of this statement.

It is perfectly valid to compare two pointer variables in C. This is particularly useful when comparing two pointers in the same array. For example, you can test the pointer `valuesPtr` to see if it points past the end of an array containing 100 elements by comparing it to a pointer to the last element in the array. So, the expression

```
valuesPtr > &values[99]
```

is TRUE (nonzero) if `valuesPtr` is pointing past the last element in the `values` array, and is FALSE (zero) otherwise. Recall from previous discussions that you can replace the preceding expression with its equivalent

```
valuesPtr > values + 99
```

because `values` used without a subscript is a pointer to the beginning of the `values` array. (Remember, it's the same as writing `&values[0]`.)

Program 11.11 illustrates pointers to arrays. The `arraySum` function calculates the sum of the elements contained in an array of integers.

Program 11.11 **Working with Pointers to Arrays**

```
// Function to sum the elements of an integer array

#include <stdio.h>

int  arraySum (int  array[], const int  n)
{
     int  sum = 0, *ptr;
     int  * const arrayEnd = array + n;

     for ( ptr = array;  ptr < arrayEnd;  ++ptr )
          sum += *ptr;
```

Program 11.11 **Continued**

```
     return sum;
}

int main (void)
{
     int  arraySum (int  array[], const int  n);
     int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

     printf ("The sum is %i\n", arraySum (values, 10));

     return 0;
}
```

Program 11.11 **Output**

```
The sum is 21
```

Inside the `arraySum` function, the constant integer pointer `arrayEnd` is defined and set pointing immediately after the last element of `array`. A `for` loop is then set up to sequence through the elements of `array`. The value of `ptr` is set to point to the beginning of `array` when the loop is entered. Each time through the loop, the element of `array` pointed to by `ptr` is added into `sum`. The value of `ptr` is then incremented by the `for` loop to set it pointing to the next element in `array`. When `ptr` points past the end of `array`, the `for` loop is exited, and the value of `sum` is returned to the calling routine.

A Slight Digression About Program Optimization

It is pointed out that the local variable `arrayEnd` was not actually needed by the function because you could have explicitly compared the value of `ptr` to the end of the array inside the `for` loop:

```
for ( ...; pointer <= array + n; ... )
```

The sole motivation for using `arrayEnd` was one of optimization. Each time through the `for` loop, the looping conditions are evaluated. Because the expression `array + n` is never changed from within the loop, its value is constant throughout the execution of the `for` loop. By evaluating it once *before* the loop is entered, you save the time that would otherwise be spent reevaluating this expression each time through the loop. Although there is virtually no savings in time for a 10-element array, especially if the `arraySum` function is called only once by the program, there could be a more substantial savings if this function were heavily used by a program for summing large-sized arrays, for example.

The other issue to be discussed about program optimization concerns the very use of pointers themselves in a program. In the `arraySum` function discussed earlier, the expression `*ptr` is used inside the `for` loop to access the elements in the array. Formerly, you would have written your `arraySum` function with a `for` loop that used an index variable, such as `i`, and then would have added the value of `array[i]` into `sum` inside the loop. In general, the process of indexing an array takes more time to execute than does the process of accessing the contents of a pointer. In fact, this is one of the main reasons why pointers are used to access the elements of an array—the code that is generated is generally more efficient. Of course, if access to the array is not generally sequential, pointers accomplish nothing, as far as this issue is concerned, because the expression `*(pointer + j)` takes just as long to execute as does the expression `array[j]`.

Is It an Array or Is It a Pointer?

Recall that to pass an array to a function, you simply specify the name of the array, as you did previously with the call to the `arraySum` function. You should also remember that to produce a pointer to an array, you need only specify the name of the array. This implies that in the call to the `arraySum` function, what was passed to the function was actually a *pointer* to the array `values`. This is precisely the case and explains why you are able to change the elements of an array from within a function.

But if it is indeed the case that a pointer to the array is passed to the function, then you might wonder why the formal parameter inside the function isn't declared to be a pointer. In other words, in the declaration of `array` in the `arraySum` function, why isn't the declaration

```
int  *array;
```

used? Shouldn't all references to an array from within a function be made using pointer variables?

To answer these questions, recall the previous discussion about pointers and arrays. As mentioned, if `valuesPtr` points to the same type of element as contained in an array called `values`, the expression `*(valuesPtr + i)` is in all ways equivalent to the expression `values[i]`, assuming that `valuesPtr` has been set to point to the beginning of `values`. What follows from this is that you also can use the expression `*(values + i)` to reference the `i`th element of the array `values`, and, in general, if `x` is an array of any type, the expression `x[i]` can always be equivalently expressed in C as `*(x + i)`.

As you can see, pointers and arrays are intimately related in C, and this is why you can declare `array` to be of type "array of `int`s" inside the `arraySum` function *or* to be of type "pointer to `int`." Either declaration works just fine in the preceding program—try it and see.

If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array. This more correctly reflects the use of the array by the function. Similarly, if you are using the argument as a pointer to the array, declare it to be of type pointer.

Realizing now that you could have declared `array` to be an `int` pointer in the preceding program example, and then could have subsequently used it as such, you can eliminate the variable `ptr` from the function and use `array` instead, as shown in Program 11.12.

Program 11.12 **Summing the Elements of an Array**

```
// Function to sum the elements of an integer array  Ver. 2

#include <stdio.h>

int  arraySum (int  *array, const int  n)
{
     int  sum = 0;
     int  * const arrayEnd = array + n;

     for (  ; array < arrayEnd;  ++array )
          sum += *array;

     return sum;
}



int main (void)
{
     int  arraySum (int  *array, const int  n);
     int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

     printf ("The sum is %i\n", arraySum (values, 10));

     return 0;
}
```

Program 11.12 **Output**

```
The sum is 21
```

The program is fairly self-explanatory. The first expression inside the `for` loop was omitted because no value had to be initialized before the loop was started. One point worth repeating is that when the `arraySum` function is called, a pointer to the `values` array is passed, where it is called `array` inside the function. Changes to the value of `array` (as opposed to the values referenced by `array`) do not in any way affect the contents of the `values` array. So, the increment operator that is applied to `array` is just incrementing a pointer to the array `values`, and not affecting its contents. (Of course, you know that you *can* change values in the array if you want to, simply by assigning values to the elements referenced by the pointer.)

Program 11.14 **Output**

```
A string to be copied.
So is this.
```

Operations on Pointers

As you have seen in this chapter, you can add or subtract integer values from pointers. Furthermore, you can compare two pointers to see if they are equal or not, or if one pointer is less than or greater than another pointer. The only other operation that is permitted on pointers is the subtraction of two pointers of the same type. The result of subtracting two pointers in C is the number of elements contained between the two pointers. So, if `a` points to an array of elements of any type and `b` points to another element somewhere farther along in the same array, the expression `b - a` represents the number of elements between these two pointers. For example, if `p` points to some element in an array `x`, the statement

```
n = p - x;
```

has the effect of assigning to the variable `n` (assumed here to be an integer variable) the index number of the element inside `x` to which `p` points.[4] Therefore, if `p` is set pointing to the hundredth element in `x` by a statement such as

```
p = &x[99];
```

the value of `n` after the previous subtraction is performed is `99`.

As a practical application of this newly learned fact about pointer subtraction, take a look at a new version of the `stringLength` function from Chapter 10.

In Program 11.15, the character pointer `cptr` is used to sequence through the characters pointed to by `string` until the null character is reached. At that point, `string` is subtracted from `cptr` to obtain the number of elements (characters) contained in the string. The program's output verifies that the function is working correctly.

Program 11.15 **Using Pointers to Find the Length of a String**

```
// Function to count the characters in a string – Pointer version

#include <stdio.h>

int  stringLength (const char  *string)
{
     const char  *cptr = string;
```

4. The actual type of signed integer that is produced by subtracting two pointers (for example, `int`, `long int`, or `long long int`) is `ptrdiff_t`, which is defined in the standard header file `<stddef.h>`.

Program 11.15 **Continued**

```
     while ( *cptr )
          ++cptr;
     return  cptr - string;
}

int main (void)
{
     int  stringLength (const char  *string);

     printf ("%i  ", stringLength ("stringLength test"));
     printf ("%i  ", stringLength (""));
     printf ("%i\n", stringLength ("complete"));

     return 0;
}
```

Program 11.15 **Output**

```
17  0  8
```

Pointers to Functions

Of a slightly more advanced nature, but presented here for the sake of completeness, is the notion of a pointer to a function. When working with pointers to functions, the C compiler needs to know not only that the pointer variable points to a function, but also the type of value returned by that function as well as the number and types of its arguments. To declare a variable `fnPtr` to be of type "pointer to function that returns an `int` and that takes no arguments," the declaration

```
int  (*fnPtr) (void);
```

can be written. The parentheses around `*fnPtr` are required because otherwise the C compiler treats the preceding statement as the declaration of a function called `fnPtr` that returns a pointer to an `int` (because the function call operator `()` has higher precedence than the pointer indirection operator `*`).

To set your function pointer pointing to a specific function, you simply assign the name of the function to it. So, if `lookup` is a function that returns an `int` and that takes no arguments, the statement

```
fnPtr = lookup;
```

stores a pointer to this function inside the function pointer variable `fnPtr`. Writing a function name without a subsequent set of parentheses is treated in an analogous way to writing an array name without a subscript. The C compiler automatically produces a pointer to the specified function. An ampersand is permitted in front of the function name, but it's not required.

If the `lookup` function has not been previously defined in the program, it is necessary to declare the function before the preceding assignment can be made. So, a statement such as

```
int  lookup (void);
```

is needed before a pointer to this function can be assigned to the variable `fnPtr`.

You can call the function that is indirectly referenced through a pointer variable by applying the function call operator to the pointer, listing any arguments to the function inside the parentheses. For example,

```
entry = fnPtr ();
```

calls the function pointed to by `fnPtr`, storing the returned value inside the variable `entry`.

One common application for pointers to functions is in passing them as arguments to other functions. The standard C library uses this, for example, in the function `qsort`, which performs a "quicksort" on an array of data elements. This function takes as one of its arguments a pointer to a function that is called whenever `qsort` needs to compare two elements in the array being sorted. In this manner, `qsort` can be used to sort arrays of any type, as the actual comparison of any two elements in the array is made by a user-supplied function, and not by the `qsort` function itself. Appendix B, "The Standard C Library," goes into more detail about `qsort` and contains an actual example of its use.

Another common application for function pointers is to create what is known as *dispatch* tables. You can't store functions themselves inside the elements of an array. However, it is valid to store function *pointers* inside an array. Given this, you can create tables that contain pointers to functions to be called. For example, you might create a table for processing different commands that will be entered by a user. Each entry in the table could contain both the command name and a pointer to a function to call to process that particular command. Now, whenever the user enters a command, you can look up the command inside the table and invoke the corresponding function to handle it.

Pointers and Memory Addresses

Before ending this discussion of pointers in C, you should note the details of how they are actually implemented. A computer's memory can be conceptualized as a sequential collection of storage cells. Each cell of the computer's memory has a number, called an *address*, associated with it. Typically, the first address of a computer's memory is numbered 0. On most computer systems, a "cell" is called a *byte*.

The computer uses memory for storing the instructions of your computer program, and also for storing the values of the variables that are associated with a program. So, if you declare a variable called `count` to be of type `int`, the system assigns location(s) in memory to hold the value of `count` while the program is executing. This location might be at address 500, for example, inside the computer's memory.

Luckily, one of the advantages of higher-level programming languages such as C is that you don't need to concern yourself with the particular memory addresses that are assigned to variables—they are automatically handled by the system. However, the knowledge that a unique memory address is associated with each variable will help you to understand the way pointers operate.

When you apply the address operator to a variable in C, the value that is generated is the actual address of that variable inside the computer's memory. (Obviously, this is where the address operator gets its name.) So, the statement

```
intPtr = &count;
```

assigns to `intPtr` the address in the computer's memory that has been assigned to the variable `count`. So, if `count` is located at address 500 and contains the value `10`, this statement assigns the value `500` to `intPtr`, as depicted in Figure 11.11.

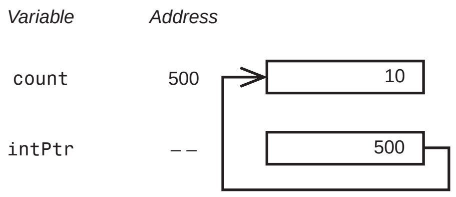

Figure 11.11 Pointers and memory addresses.

The address of `intPtr` is shown in Figure 11.11 as `--` because its actual value is irrelevant for this example.

Applying the indirection operator to a pointer variable, as in the expression

```
*intPtr
```

has the effect of treating the value contained in the pointer variable as a memory address. The value stored at that memory address is then fetched and interpreted in accordance with the type declared for the pointer variable. So, if `intPtr` is of type pointer to `int`, the value stored in the memory address given by `*intPtr` is interpreted as an integer by the system. In our example, the value stored at memory address 500 is fetched and interpreted as an integer. The result of the expression is 10, and is of type `int`.

Storing a value in a location reference by a pointer, as in

```
*intPtr = 20;
```

proceeds in a like manner. The contents of `intPtr` is fetched and treated as a memory address. The specified integer value is then stored at that address. In the preceding statement, the integer value of `20` is, therefore, stored at memory address 500.

At times, system programmers must access particular locations inside the computer's memory. In such cases, this knowledge of the way that pointer variables operate proves helpful.

As you can see from this chapter, the pointer is a very powerful construct in C. The flexibility in defining pointer variables extends beyond those illustrated in this chapter. For example, you can define a pointer to a pointer, and even a pointer to a pointer to a pointer. These types of constructs are beyond the scope of this book, although they are simply logical extensions of everything you've learned about pointers in this chapter.

The topic of pointers is probably the hardest for novices to grasp. You should reread any sections of this chapter that still seem unclear before proceeding. Solving the exercises that follow will also help you to understand the material.

Exercises

1. Type in and run the 15 programs presented in this chapter. Compare the output produced by each program with the output presented after each program in the text.
2. Write a function called `insertEntry` to insert a new entry into a linked list. Have the procedure take as arguments a pointer to the list entry to be inserted (of type `struct entry` as defined in this chapter), and a pointer to an element in the list *after* which the new `entry` is to be inserted.
3. The function developed in exercise 2 only inserts an element after an existing element in the list, thereby preventing you from inserting a new entry at the front of the list. How can you use this same function and yet overcome this problem? (*Hint:* Think about setting up a special structure to point to the beginning of the list.)
4. Write a function called `removeEntry` to remove an `entry` from a linked list. The sole argument to the procedure should be a pointer to the list. Have the function remove the entry *after* the one pointed to by the argument. (Why can't you remove the entry pointed to by the argument?) You need to use the special structure you set up in exercise 3 to handle the special case of removing the first element from the list.
5. A *doubly linked list* is a list in which each entry contains a pointer to the preceding entry in the list as well as a pointer to the next entry in the list. Define the appropriate structure definition for a doubly linked list entry and then write a small program that implements a small doubly linked list and prints out the elements of the list.
6. Develop `insertEntry` and `removeEntry` functions for a doubly linked list that are similar in function to those developed in previous exercises for a singly linked list. Why can your `removeEntry` function now take as its argument a direct pointer to the entry to be removed from the list?
7. Write a pointer version of the `sort` function from Chapter 8, "Working with Functions." Be certain that pointers are exclusively used by the function, including index variables in the loops.

The ## Operator

This operator is used in macro definitions to join two *tokens* together. It is preceded (or followed) by the name of a parameter to the macro. The preprocessor takes the actual argument to the macro that is supplied when the macro is invoked and creates a single token out of that argument and whatever token follows (or precedes) the `##`.

Suppose, for example, you have a list of variables `x1` through `x100`. You can write a macro called `printx` that simply takes as its argument an integer value `1` through `100` and that displays the corresponding `x` variable as shown:

```
#define  printx(n)   printf ("%i\n", x ## n)
```

The portion of the define that reads

```
x ## n
```

says to take the tokens that occur before and after the `##` (the letter `x` and the argument `n`, respectively) and make a single token out of them. So the call

```
printx (20);
```

is expanded into

```
printf ("%i\n", x20);
```

The `printx` macro can even use the previously defined `printint` macro to get the variable name as well as its value displayed:

```
#define  printx(n)   printint(x ## n)
```

The invocation

```
printx (10);
```

first expands into

```
printint (x10);
```

and then into

```
printf ("x10" " = %i\n", x10);
```

and finally into

```
printf ("x10 = %i\n", x10);
```

The `#include` Statement

After you have programmed in C for a while, you will find yourself developing your own set of macros that you will want to use in each of your programs. But instead of having to type these macros into each new program you write, the preprocessor enables you to collect all your definitions into a separate file and then *include* them in your program, using the `#include` statement. These files normally end with the characters `.h` and are referred to as *header* or *include* files.

Suppose you are writing a series of programs for performing various metric conversions. You might want to set up some defines for all of the constants that you need to perform your conversions:

```
#define  INCHES_PER_CENTIMETER   0.394
#define  CENTIMETERS_PER_INCH    1 / INCHES_PER_CENTIMETER

#define  QUARTS_PER_LITER        1.057
#define  LITERS_PER_QUART        1 / QUARTS_PER_LITER

#define  OUNCES_PER_GRAM         0.035
#define  GRAMS_PER_OUNCE         1 / OUNCES_PER_GRAM
   ...
```

Suppose you entered the previous definitions into a separate file on the system called `metric.h`. Any program that subsequently needed to use any of the definitions contained in the `metric.h` file could then do so by simply issuing the preprocessor directive

```
#include "metric.h"
```

This statement must appear before any of the defines contained in `metric.h` are referenced and is typically placed at the beginning of the source file. The preprocessor looks for the specified file on the system and effectively copies the contents of the file into the program at the precise point that the `#include` statement appears. So, any statements inside the file are treated just as if they had been directly typed into the program at that point.

The double quotation marks around the include filename instruct the preprocessor to look for the specified file in one or more file directories (typically first in the same directory that contains the source file, but the actual places the preprocessor searches are system dependent). If the file isn't located, the preprocessor automatically searches other *system* directories as described next.

Enclosing the filename within the characters `<` and `>` instead, as in

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

causes the preprocessor to look for the include file in the special system include file directory or directories. Once again, these directories are system dependent. On Unix systems (including Mac OS X systems), the system include file directory is `/usr/include`, so the standard header file `stdio.h` can be found in `/usr/include/stdio.h`.

To see how include files are used in an actual program example, type the six defines given previously into a file called `metric.h`. Then type in and run Program 13.3.

Program 13.3 **Using the `#include` Statement**

```
/* Program to illustrate the use of the #include statement
   Note: This program assumes that definitions are
   set up in a file called metric.h             */
```

Program 13.3 **Continued**

```
#include <stdio.h>
#include "metric.h"

int main (void)
{
    float  liters, gallons;

    printf ("*** Liters to Gallons ***\n\n");
    printf ("Enter the number of liters: ");
    scanf ("%f", &liters);

    gallons = liters * QUARTS_PER_LITER / 4.0;
    printf ("%g liters = %g gallons\n", liters, gallons);

    return 0;
}
```

Program 13.3 **Output**

```
*** Liters to Gallons ***

Enter the number of liters: 55.75
55.75 liters = 14.73 gallons.
```

The preceding example is a rather simple one because it only shows a single defined value (`QUARTS_PER_LITER`) being referenced from the included file `metric.h`. Nevertheless, the point is well made: After the definitions have been entered into `metric.h`, they can be used in any program that uses an appropriate `#include` statement.

One of the nicest things about the include file capability is that it enables you to centralize your definitions, thus ensuring that all programs reference the same value. Furthermore, errors discovered in one of the values contained in the include file need only be corrected in that one spot, thus eliminating the need to correct each and every program that uses the value. Any program that references the incorrect value simply needs to be recompiled and does not have to be edited.

You can actually put anything you want in an include file—not just `#define` statements, as might have been implied. Using include files to centralize commonly used preprocessor definitions, structure definitions, prototype declarations, and global variable declarations is good programming technique.

One last point to be made about include files in this chapter: Include files can be nested. That is, an include file can itself include another file, and so on.

System Include Files

It was noted that the include file `<stddef.h>` contains a define for `NULL` and is often used for testing to see whether a pointer has a null value. Earlier in this chapter, it was also noted that the header file `<math.h>` contains the definition `M_PI`, which is set to an approximation for the value of π.

The `<stdio.h>` header file contains information about the I/O routines contained in the standard I/O library. This header file is described in more detail in Chapter 16, "Input and Output Operations in C." You should include this file whenever you use any I/O library routine in your program.

Two other useful system include files are `<limits.h>` and `<float.h>`. The first file, `<limits.h>`, contains system-dependent values that specify the sizes of various character and integer data types. For instance, the maximum size of an `int` is defined by the name `INT_MAX` inside this file. The maximum size of an `unsigned long int` is defined by `ULONG_MAX`, and so on.

The `<float.h>` header file gives information about floating-point data types. For example, `FLT_MAX` specifies the maximum floating-point number, and `FLT_DIG` specifies the number of decimal digits of precision for a `float` type.

Other system include files contain prototype declarations for various functions stored inside the system library. For example, the include file `<string.h>` contains prototype declarations for the library routines that perform character string operations, such as copying, comparing, and concatenating.

For more details on these header files, consult Appendix B.

Conditional Compilation

The C preprocessor offers a feature known as *conditional compilation*. Conditional compilation is often used to create one program that can be compiled to run on different computer systems. It is also often used to switch on or off various statements in the program, such as debugging statements that print out the values of various variables or trace the flow of program execution.

The `#ifdef`, `#endif`, `#else`, and `#ifndef` Statements

You were shown earlier in this chapter how you could make the `rotate` function from Chapter 12 more portable. You saw there how the use of a define would help in this regard. The definition

```
#define  kIntSize  32
```

was used to isolate the dependency on the specific number of bits contained in an `unsigned int`. It was noted in several places that this dependency does not have to be made at all because the program can itself determine the number of bits stored inside an `unsigned int`.

Unfortunately, a program sometimes must rely on system-dependent parameters—on a filename, for example—that might be specified differently on different systems or on a particular feature of the operating system.

If you had a large program that had many such dependencies on the particular hardware and/or software of the computer system (and this should be minimized as much as possible), you might end up with many defines whose values would have to be changed when the program was moved to another computer system.

You can help reduce the problem of having to change these defines when the program is moved and can incorporate the values of these defines for each different machine into the program by using the conditional compilation capabilities of the preprocessor. As a simple example, the statements

```
#ifdef  UNIX
#   define  DATADIR    "/uxn1/data"
#else
#   define  DATADIR    "\usr\data"
#endif
```

have the effect of defining `DATADIR` to `"/uxn1/data"` if the symbol `UNIX` has been previously defined and to `"\usr\data"` otherwise. As you can see here, you are allowed to put one or more spaces after the # that begins a preprocessor statement.

The `#ifdef`, `#else`, and `#endif` statements behave as you would expect. If the symbol specified on the `#ifdef` line has been already defined—through a `#define` statement or through the command line when the program is compiled—then lines that follow up to a `#else`, `#elif`, or `#endif` are processed by the compiler; otherwise, they are ignored.

To define the symbol `UNIX` to the preprocessor, the statement

```
#define  UNIX    1
```

or even just

```
#define  UNIX
```

suffices. Most compilers also permit you to define a name to the preprocessor when the program is compiled by using a special option to the compiler command. The `gcc` command line

```
gcc -D UNIX program.c
```

defines the name `UNIX` to the preprocessor, causing all `#ifdef UNIX` statements inside `program.c` to evaluate as TRUE (note that the `-D UNIX` must be typed *before* the program name on the command line). This technique enables names to be defined *without* having to edit the source program.

A value can also be assigned to the defined name on the command line. For example,

```
gcc -D GNUDIR=/c/gnustep program.c
```

invokes the `gcc` compiler, defining the name `GNUDIR` to be the text `/c/gnustep`.

Avoiding Multiple Inclusion of Header Files

The `#ifndef` statement follows along the same lines as the `#ifdef`. This statement is used the same way the `#ifdef` statement is used, except that it causes the subsequent lines to be processed if the indicated symbol is *not* defined. This statement is often used to avoid multiple inclusion of a file in a program. For example, inside a header file, if you want to make certain it is included only once in a program, you can define a unique identifier that can be tested later. Consider the sequence of statements:

```
#ifndef _MYSTDIO_H
#define _MYSTDIO_H
  ...
#endif /* _MYSTDIO_H */
```

Suppose you typed this into a file called `mystdio.h`. If you included this file in your program with a statement like this:

```
#include "mystdio.h"
```

the `#ifndef` inside the file would test whether `_MYSTDIO_H` were defined. Because it wouldn't be, the lines between the `#ifndef` and the matching `#endif` would be included in the program. Presumably, this would contain all of the statements that you want included in your program from this header file. Notice that the very next line in the header file defines `_MYSTDIO_H`. If an attempt were made to again include the file in the program, `_MYSTDIO_H` would be defined, so the statements that followed (up to the `#endif`, which presumably is placed at the very end of your header file) would not be included in the program, thus avoiding multiple inclusion of the file in the program.

This method as shown is used in the system header files to avoid their multiple inclusion in your programs. Take a look at some and see!

The `#if` and `#elif` Preprocessor Statements

The `#if` preprocessor statement offers a more general way of controlling conditional compilation. The `#if` statement can be used to test whether a constant expression evaluates to nonzero. If the result of the expression is nonzero, subsequent lines up to a `#else`, `#elif`, or `#endif` are processed; otherwise, they are skipped. As an example of how this might be used, assume you define the name `OS`, which is set to `1` if the operating system is Macintosh OS, to `2` if the operating system is Windows, to `3` if the operating system is Linux, and so on. You could write a sequence of statements to conditionally compile statements based upon the value of `OS` as follows:

```
#if    OS == 1  /* Mac OS */
  ...
#elif  OS == 2  /* Windows */
    ...
#elif  OS == 3  /* Linux  */
    ...
#else
    ...
#endif
```

With most compilers, you can assign a value to the name `OS` on the command line using the `-D` option discussed earlier. The command line

```
gcc -D OS=2 program.c
```

15

Working with Larger Programs

THE PROGRAMS THAT HAVE BEEN ILLUSTRATED throughout this book have all been very small and relatively simple. Unfortunately, the programs that you will have to develop to solve your particular problems will probably be neither as small nor as simple. Learning the proper techniques for dealing with such programs is the topic of this chapter. As you will see, C provides all the features necessary for the efficient development of large programs. In addition, you can use several utility programs—which are briefly mentioned in this chapter—that make working with large projects easier.

Dividing Your Program into Multiple Files

In every program that you've seen so far, it was assumed that the entire program was entered into a single file—presumably via some text editor, such as `emacs`, `vim`, or some Windows-based editor—and then compiled and executed. In this single file, all the functions that the program used were included—except, of course, for the system functions, such as `printf` and `scanf`. Standard header files such as `<stdio.h>` and `<stdbool.h>` were also included for definitions and function declarations. This approach works fine when dealing with small programs—that is, programs that contain up to 100 statements or so. However, when you start dealing with larger programs, this approach no longer suffices. As the number of statements in the program increases, so does the time it takes to edit the program and to subsequently recompile it. Not only that, large programming applications frequently require the efforts of more than one programmer. Having everyone work on the same source file, or even on their own copy of the same source file, is unmanageable.

C supports the notion of modular programming in that it does not require that all the statements for a particular program be contained in a single file. This means that you can enter your code for a particular module into one file, for another module into a different file, and so on. Here, the term *module* refers either to a single function or to a number of related functions that you choose to group logically.

If you're working with a windows-based project management tool, such as Metrowerks' CodeWarrior, Microsoft Visual Studio, or Apple's Xcode, then working with multiple source files is easy. You simply have to identify the particular files that belong to the project on which you are working, and the software handles the rest for you. The next section describes how to work with multiple files if you're not using such a tool, also known as an Integrated Development Environment (IDE). That is, the next section assumes you are compiling programs from the command line by directly issuing `gcc` or `cc` commands, for example.

Compiling Multiple Source Files from the Command Line

Suppose you have conceptually divided your program into three modules and have entered the statements for the first module into a file called `mod1.c`, the statements for the second module into a file called `mod2.c`, and the statements for your `main` routine into the file `main.c`. To tell the system that these three modules actually belong to the same program, you simply include the names of all three files when you enter the command to compile the program. For example, using `gcc`, the command

```
$ gcc mod1.c mod2.c main.c –o dbtest
```

has the effect of separately compiling the code contained in `mod1.c`, `mod2.c`, and `main.c`. Errors discovered in `mod1.c`, `mod2.c`, and `main.c` are separately identified by the compiler. For example, if the `gcc` compiler gives output that looks like this:

```
mod2.c:10: mod2.c: In function 'foo':
mod2.c:10: error: 'i' undeclared (first use in this function)
mod2.c:10: error: (Each undeclared identifier is reported only once
mod2.c:10: error: for each function it appears in.)
```

then the compiler indicates that `mod2.c` has an error at line 10, which is in the function `foo`. Because no messages are displayed for `mod1.c` and `main.c`, no errors are found compiling those modules.

Typically, if there are errors discovered in a module, you have to edit the module to correct the mistakes.[1] In this case, because an error was discovered only inside `mod2.c`, you have to edit only this file to fix the mistake. You can then tell the C compiler to recompile your modules after the correction has been made:

```
$ gcc mod1.c mod2.c main.c –o dbtest
$
```

Because no error message was reported, the executable was placed in the file `dbtest`.

Normally, the compiler generates intermediate object files for each source file that it compiles. The compiler places the resulting object code from compiling *`mod.c`* into the file *`mod`*`.o` by default. (Most Windows compilers work similarly, only they might place

1. The error might be due to a problem with a header file included by that module, for example, which means the header file and not the module would have to be edited.

the resulting object code into `.obj` files instead of `.o` files.) Typically, these intermediate object files are automatically deleted after the compilation process ends. Some C compilers (and, historically, the standard Unix C compiler) keep these object files around and do not delete them when you compile more than one file at a time. This fact can be used to your advantage for recompiling a program after making a change to only one or several of your modules. So in the previous example, because `mod1.c` and `main.c` had no compiler errors, the corresponding `.o` files—`mod1.o` and `main.o`—would still be around after the `gcc` command completed. Replacing the `c` from the filename *`mod`*`.c` with an `o` tells the C compiler to use the object file that was produced the last time *`mod`*`.c` was compiled. So, the following command line could be used with a compiler (in this case, `cc`) that does not delete the object code files:

```
$ cc mod1.o mod2.c main.o –o dbtest
```

So, not only do you not have to reedit `mod1.c` and `main.c` if no errors are discovered by the compiler, but you also don't have to recompile them.

If your compiler automatically deletes the intermediate `.o` files, you can still take advantage of performing incremental compilations by compiling each module separately and using the `–c` command-line option. This option tells the compiler not to link your file (that is, not to try to produce an executable) and to retain the intermediate object file that it creates. So, typing

```
$ gcc –c mod2.c
```

compiles the file `mod2.c`, placing the resulting executable in the file `mod2.o`.

So, in general, you can use the following sequence to compile your three-module program `dbtest` using the incremental compilation technique:

```
$ gcc –c mod1.c                       Compile mod1.c => mod1.o
$ gcc –c mod2.c                       Compile mod2.c => mod2.o
$ gcc –c main.c                       Compile main.c => main.o
$ gcc mod1.o mod2.o mod3.o –o dbtest  Create executable
```

The three modules are compiled separately. The previous output shows no errors were detected by the compiler. If any were, the file could be edited and incrementally recompiled. The last line that reads

```
$ gcc mod1.o mod2.o mod3.o
```

lists only object files and no source files. In this case, the object files are just linked together to produce the executable output file `dbtest`.

If you extend the preceding examples to programs that consist of many modules, you can see how this mechanism of separate compilations can enable you to develop large programs more efficiently. For example, the commands

```
$ gcc –c legal.c                     Compile legal.c, placing output in legal.o
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o –o superchess
```

could be used to compile a program consisting of six modules, in which only the module `legal.c` needs to be recompiled.

As you'll see in the last section of this chapter, the process of incremental compilation can be automated by using a tool called `make`. The IDE tools that were mentioned at the beginning of this chapter invariably have this knowledge of what needs recompilation, and they only recompile files as necessary.

Communication Between Modules

Several methods can be used so that the modules contained in separate files can effectively communicate. If a function from one file needs to call a function contained inside another file, the function call can be made in the normal fashion, and arguments can be passed and returned in the usual way. Of course, in the file that calls the function, you should *always make certain to include a prototype declaration so the compiler knows the function's argument types and the type of the return value.* As noted in Chapter 14, "More on Data Types," in the absence of any information about a function, the compiler assumes it returns an `int` and converts `short` or `char` arguments to `int`s and `float` arguments to `double`s when the function is called.

It's important to remember that even though more than one module might be specified to the compiler at the same time on the command line, *the compiler compiles each module independently*. That means that no knowledge about structure definitions, function return types, or function argument types is shared across module compilations by the compiler. It's totally up to you to ensure that the compiler has sufficient information about such things to correctly compile each module.

External Variables

Functions contained in separate files can communicate through *external variables*, which are effectively an extension to the concept of the global variable discussed in Chapter 8, "Working with Functions."

An external variable is one whose value can be accessed and changed by another module. Inside the module that wants to access the external variable, the variable is declared in the normal fashion and the keyword `extern` is placed before the declaration. This signals to the system that a globally defined variable from another file is to be accessed.

Suppose you want to define an `int` variable called `moveNumber`, whose value you want to access and possibly modify from within a function contained in another file. In Chapter 8, you learned that if you wrote the statement

```
int  moveNumber = 0;
```

at the beginning of your program, *outside* of any function, then its value could be referenced by any function within that program. In such a case, `moveNumber` was defined as a global variable.

Actually, this same definition of the variable `moveNumber` also makes its value accessible by functions contained in other files. Specifically, the preceding statement defines the

Using Header Files Effectively

In Chapter 13, "The Preprocessor," you were introduced to the concept of the include file. As stated there, you can group all your commonly used definitions inside such a file and then simply include the file in any program that needs to use those definitions. Nowhere is the usefulness of the `#include` facility greater than in developing programs that have been divided into separate program modules.

If more than one programmer is working on developing a particular program, include files provide a means of standardization: Each programmer is using the same definitions, which have the same values. Furthermore, each programmer is thus spared the time-consuming and error-prone task of typing these definitions into each file that must use them. These last two points are made even stronger when you start placing common structure definitions, external variable declarations, `typedef` definitions, and function prototype declarations into include files. Various modules of a large programming system invariably deal with common data structures. By centralizing the definition of these data structures into one or more include files, you eliminate the error that is caused by two modules that use different definitions for the same data structure. Furthermore, if a change has to be made to the definition of a particular data structure, it can be done in one place only—inside the include file.

Recall your `date` structure from Chapter 9, "Working with Structures"; following is an include file that might be similar to one you would set up if you have to work with a lot of dates within different modules. It is also a good example of how to tie together many of the concepts you've learned up to this point.

```
// Header file for working with dates


#include <stdbool.h>


// Enumerated types

enum kMonth { January=1, February, March, April, May, June,
        July, August, September, October, November, December };

enum kDay { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday };


struct  date
{
    enum  kMonth month;
    enum  kDay   day;
    int          year;
};
```

```
// Date type
typedef struct date Date;

// Functions that work with dates
Date  dateUpdate (Date today);
int   numberOfDays  (Date  d);
bool  isLeapYear (Date  d);

// Macro to set a date in a structure
#define setDate(s,mm,dd,yy)  s = (Date) {mm, dd, yy}

// External variable reference
extern Date todaysDate;
```

The header file defines two enumerated data types, `kMonth` and `kDay`, and the date structure (and note the use of the enumerated data types); uses `typedef` to create a type called `Date`; and declares functions that use this type, a macro to set a date to specific values (using compound literals), and an external variable called `todaysDate`, that will presumably be set to today's date (and is defined in one of the source files).

As an example using this header file, the following is a rewritten version of the `dateUpdate` function from Chapter 9.

```
#include "date.h"

// Function to calculate tomorrow's date

Date dateUpdate (Date today)
{
    Date  tomorrow;

    if ( today.day != numberOfDays (today) )
        setDate (tomorrow, today.month, today.day + 1, today.year);
    else if ( today.month == December )     // end of year
        setDate (tomorrow, January, 1, today.year + 1);
    else                              // end of month
        setDate (tomorrow, today.month + 1, 1, today.year);

    return tomorrow;
} .
```

Other Utilities for Working with Larger Programs

As briefly mentioned previously, the IDE can be a powerful tool for working with larger programs. If you still want to work from the command line, there are tools you might the output file by `argv[2]`. After opening the first file for reading and the second file for writing, and after checking both opens to make certain they succeeded, the program copies the file character by character as before.

Note that there are four different ways for the program to terminate: incorrect number of command-line arguments, can't open the file to be copied for reading, can't open the output file for writing, and successful termination. Remember, if you're going to use the exit status, you should *always* terminate the program with one. If your program terminates by falling through the bottom of `main`, it returns an *undefined* exit status.

If Program 17.1 were called `copyf` and the program was executed with the following command line:

```
copyf foo foo1
```

then the `argv` array would look like Figure 17.1 when `main` is entered.

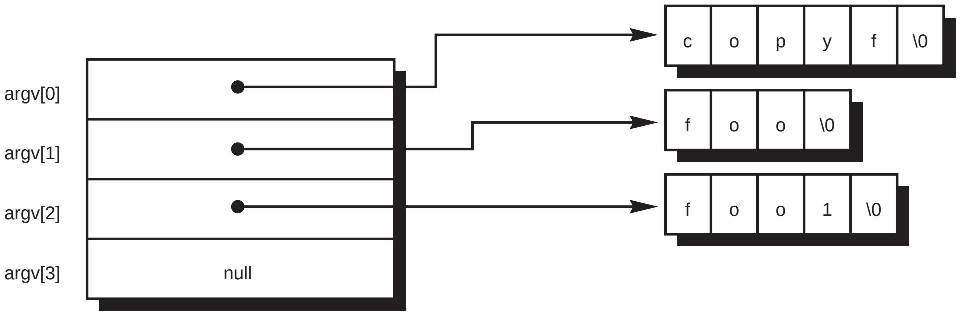

Figure 17.1 `argv` array on startup of `copyf`.

Remember that command-line arguments are *always* stored as character strings. Execution of the program `power` with the command-line arguments 2 and 16, as in

```
power 2 16
```

stores a pointer to the character string `"2"` inside `argv[1]`, and a pointer to the string `"16"` inside `argv[2]`. If the arguments are to be interpreted as numbers by the program (as you might suspect is the case in the `power` program), they must be converted by the program itself. Several routines are available in the program library for doing such conversions, such as `sscanf`, `atof`, `atoi`, `strtod`, and `strtol`. These are described in Appendix B, "The Standard C Library."

Dynamic Memory Allocation

Whenever you define a variable in C—whether it is a simple data type, an array, or a structure—you are effectively reserving one or more locations in the computer's memory to contain the values that will be stored in that variable. The C compiler automatically allocates the correct amount of storage for you.

It is frequently desirable, if not necessary, to be able to *dynamically* allocate storage while a program is running. Suppose you have a program that is designed to read in a set of data from a file into an array in memory. Suppose, however, that you don't know how much data is in the file until the program starts execution. You have three choices:

- Define the array to contain the maximum number of possible elements at compile time.
- Use a variable-length array to dimension the size of the array at runtime.
- Allocate the array dynamically using one of C's memory allocation routines.

Using the first approach, you have to define your array to contain the maximum number of elements that would be read into the array, as in the following:

```
#define  kMaxElements    1000

struct dataEntry  dataArray [kMaxElements];
```

Now, as long as the data file contains 1,000 elements or less, you're in business. But if the number of elements exceeds this amount, you must go back to the program, change the value of `kMaxElements`, and recompile it. Of course, no matter what value you select, you always have the chance of running into the same problem again in the future.

With the second approach, if you can determine the number of elements you need before you start reading in the data (perhaps from the size of the file, for example), you can then define a variable-length array as follows:

```
struct dateEntry dataArray [dataItems];
```

Here, it is assumed that the variable `dataItems` contains the aforementioned number of data items to read in.

Using the dynamic memory allocation functions, you can get storage as you need it. That is, this approach also enables you to allocate memory as the program is executing. To use dynamic memory allocation, you must first learn about three functions and one new operator.

The `calloc` and `malloc` Functions

In the standard C library, two functions, called `calloc` and `malloc`, can be used to allocate memory at runtime. The `calloc` function takes two arguments that specify the number of elements to be reserved and the size of each element in *bytes*. The function returns a pointer to the beginning of the allocated storage area in memory. The storage area is also automatically set to 0.

`calloc` returns a pointer to `void`, which is C's generic pointer type. Before storing this returned pointer inside a pointer variable in your program, it can be converted into a pointer of the appropriate type using the type cast operator.

The `malloc` function works similarly, except that it only takes a single argument—the total number of bytes of storage to allocate—and also doesn't automatically set the storage area to 0.

The dynamic memory allocation functions are declared in the standard header file `<stdlib.h>`, which should be included in your program whenever you want to use these routines.

The `sizeof` Operator

To determine the size of data elements to be reserved by `calloc` or `malloc` in a machine-independent way, the C `sizeof` operator should be used. The `sizeof` operator returns the size of the specified item in bytes. The argument to the `sizeof` operator can be a variable, an array name, the name of a basic data type, the name of a derived data type, or an expression. For example, writing

```
sizeof (int)
```

gives the number of bytes needed to store an integer. On a Pentium 4 machine, this has the value `4` because an integer occupies 32 bits on that machine. If `x` is defined to be an array of 100 integers, the expression

```
sizeof (x)
```

gives the amount of storage required for the 100 integers of `x` (or the value `400` on a Pentium 4). The expression

```
sizeof (struct dataEntry)
```

has as its value the amount of storage required to store one `dataEntry` structure. Finally, if `data` is defined as an array of `struct dataEntry` elements, the expression

```
sizeof (data) / sizeof (struct dataEntry)
```

gives the number of elements contained in `data` (`data` must be a previously defined array, and not a formal parameter or externally referenced array). The expression

```
sizeof (data) / sizeof (data[0])
```

also produces the same result. The macro

```
#define  ELEMENTS(x)   (sizeof(x) / sizeof(x[0]))
```

simply generalizes this technique. It enables you to write code like

```
if ( i >= ELEMENTS (data) )
   ...
```

and

```
for ( i = 0; i < ELEMENTS (data); ++i )
   ...
```

You should remember that `sizeof` is actually an operator, and not a function, even though it looks like a function. This operator is evaluated at compile time and not at runtime, unless a variable-length array is used in its argument. If such an array is not used, the compiler evaluates the value of the `sizeof` expression and replaces it with the result of the calculation, which is treated as a constant.

Use the `sizeof` operator wherever possible to avoid having to calculate and hard-code sizes into your program.

Getting back to dynamic memory allocation, if you want to allocate enough storage in your program to store 1,000 integers, you can call `calloc` as follows:

```
#include <stdlib.h>
  ...
int  *intPtr;
 ...
intPtr = (int *) calloc (sizeof (int), 1000);
```

Using `malloc`, the function call looks like this:

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

Remember that both `malloc` and `calloc` are defined to return a pointer to `void` and, as noted, this pointer should be type cast to the appropriate pointer type. In the preceding example, the pointer is type cast to an integer pointer and then assigned to `intPtr`.

If you ask for more memory than the system has available, `calloc` (or `malloc`) returns a null pointer. Whether you use `calloc` or `malloc`, be certain to test the pointer that is returned to ensure that the allocation succeeded.

The following code segment allocates space for 1,000 integer pointers and tests the pointer that is returned. If the allocation fails, the program writes an error message to standard error and then exits.

```
#include <stdlib.h>
#include <stdio.h>
   ...
int  *intPtr;
   ...
intptr = (int *) calloc (sizeof (int), 1000);

if ( intPtr == NULL )
{
    fprintf (stderr, "calloc failed\n");
    exit (EXIT_FAILURE);
}
```

If the allocation succeeds, the integer pointer variable `intPtr` can be used as if it were pointing to an array of 1,000 integers. So, to set all 1,000 elements to –1, you could write

```
for ( p = intPtr; p < intPtr + 1000; ++p )
    *p = -1;
```

assuming `p` is declared to be an integer pointer.

To reserve storage for `n` elements of type `struct dataEntry`, you first need to define a pointer of the appropriate type

```
struct dataEntry  *dataPtr;
```

and could then proceed to call the `calloc` function to reserve the appropriate number of elements

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

Execution of the preceding statement proceeds as follows:

1. The `calloc` function is called with two arguments, the first specifying that storage for `n` elements is to be dynamically allocated and the second specifying the size of each element.
2. The `calloc` function returns a pointer in memory to the allocated storage area. If the storage cannot be allocated, the null pointer is returned.
3. The pointer is type cast into a pointer of type "pointer to `struct dataEntry`" and is then assigned to the pointer variable `dataPtr`.

Once again, the value of `dataPtr` should be subsequently tested to ensure that the allocation succeeded. If it did, its value is nonnull. This pointer can then be used in the normal fashion, as if it were pointing to an array of `n dataEntry` elements. For example, if `dataEntry` contains an integer member called `index`, you can assign 100 to this member as pointed to by `dataPtr` with the following statement:

```
dataPtr->index = 100;
```

The `free` Function

When you have finished working with the memory that has been dynamically allocated by `calloc` or `malloc`, you should give it back to the system by calling the `free` function. The single argument to the function is a pointer to the beginning of the allocated memory, as returned by a `calloc` or `malloc` call. So, the call

```
free (dataPtr);
```

returns the memory allocated by the `calloc` call shown previously, provided that the value of `dataPtr` still points to the *beginning* of the allocated memory.

The `free` function does not return a value.

The memory that is released by `free` can be reused by a later call to `calloc` or `malloc`. For programs that need to allocate more storage space than would otherwise be available if it were all allocated at once, this is worth remembering. Make certain you give the `free` function a valid pointer to the beginning of some previously allocated space.

Dynamic memory allocation is invaluable when dealing with linked structures, such as linked lists. When you need to add a new entry to the list, you can dynamically allocate storage for one entry in the list and link it into the list with the pointer returned by `calloc` or `malloc`. For example, assume that `listEnd` points to the end of a singly linked list of type `struct entry`, defined as follows:

```
struct entry
{
    int           value;
    struct entry  *next;
};
```

Here is a function called `addEntry` that takes as its argument a pointer to the start of the linked list and that adds a new entry to the end of the list.

```
#include <stdlib.h>
#include <stddef.h>

// add new entry to end of linked list

struct entry *addEntry (struct entry *listPtr)
{
    // find the end of the list

    while ( listPtr->next != NULL )
        listPtr = listPtr->next;

    // get storage for new entry

    listPtr->next = (struct entry *) malloc (sizeof (struct entry));

    // add null to the new end of the list

    if ( listPtr->next != NULL )
         (listPtr->next)->next = (struct entry *) NULL;

    return listPtr->next;
}
```

If the allocation succeeds, a null pointer is placed in the `next` member of the newly allocated linked-list entry (pointed to by `listPtr->next`).

The function returns a pointer to the new list entry, or the null pointer if the allocation fails (verify that this is, in fact, what happens). If you draw a picture of a linked list and trace through the execution of `addEntry`, it will help you to understand how the function works.

Another function, called `realloc`, is associated with dynamic memory allocation. It can be used to shrink or expand the size of some previously allocated storage. For more details, consult Appendix B.

This chapter concludes coverage of the features of the C language. In Chapter 18, "Debugging Programs," you learn some techniques that will help you to debug your C programs. One involves using the preprocessor. The other involves the use of a special tool, called an interactive debugger.