Documento de lecturas complementarias

La bibliografía relacionada a este material pueden encontrarse en
https://www.dropbox.com/s/nxp8bva28h9o0f8/prgrmmng_II.pdf?dl=0

12 de febrero de 2020

Parte I

Lecturas sobre Estructuras

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.

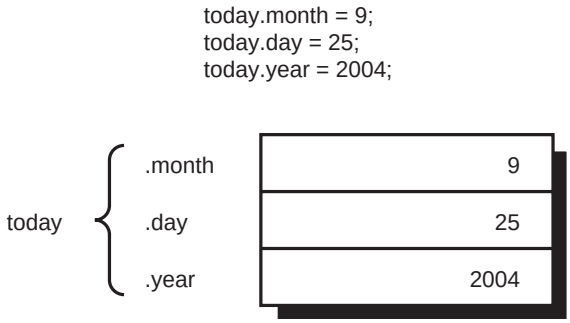

Figure 9.1. Assigning values to a structure variable.

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);
```

illustrates the ability to pass a structure to a function and to return one as well. The `dateUpdate` function has the appropriate declaration to indicate that the function returns a value of type `struct date`. Inside the function is the same code that was included in the `main` routine of Program 9.3. The functions `numberOfDays` and `isLeapYear` remain unchanged from that program.

Make certain that you understand the hierarchy of function calls in the preceding program: The `main` function calls `dateUpdate`, which in turn calls `numberOfDays`, which itself calls the function `isLeapYear`.

A Structure for Storing the Time

Suppose you have the need to store values inside a program that represents various times expressed as hours, minutes, and seconds. Because you have seen how useful our `date` structure has been in helping you to logically group the day, month, and year, it seems only natural to use a structure that you could call appropriately enough, `time`, to group the hours, minutes, and seconds. The structure definition is straightforward enough, as follows:

```
struct time
{
    int   hour;
    int   minutes;
    int   seconds;
};
```

Most computer installations choose to express the time in terms of a 24-hour clock, known as military time. This representation avoids the hassle of having to qualify a time with a.m. or p.m. The hour begins with 0 at 12 midnight and increases by 1 until it reaches 23, which represents 11:00 p.m. So, for example, 4:30 means 4:30 a.m., whereas 16:30 represents 4:30 p.m.; and 12:00 represents noon, whereas 00:01 represents 1 minute after midnight.

Virtually all computers have a clock inside in the system that is always running. This clock is used for such diversified purposes as informing the user of the current time, causing certain events to occur or programs to be executed at specific times, or recording the time that a particular event occurs. One or more computer programs are usually associated with the clock. One of these programs might be executed every second, for example, to update the current time that is stored somewhere in the computer's memory.

Suppose you want to mimic the function of the program described previously—namely, to develop a program that updates the time by one second. If you think about this for a second (pun intentional), you realize that this problem is quite analagous to the problem of updating the date by one day.

Just as finding the next day had some special requirements, so does the process of updating the time. In particular, these special cases must be handled:

1. If the number of seconds reaches 60, the seconds must be reset to 0 and the minutes increased by 1.
2. If the number of minutes reaches 60, the minutes must be reset to 0 and the hour increased by 1.
3. If the number of hours reaches 24, the hours, minutes, and seconds must be reset to 0.

Program 9.5 uses a function called `timeUpdate`, which takes as its argument the current time and returns a time that is one second later.

Program 9.5 **Updating the Time by One Second**

```
//  Program to update the time by one second

#include <stdio.h>

struct  time
{
    int  hour;
    int  minutes;
    int  seconds;
};

int main (void)
{
    struct time  timeUpdate (struct time  now);
    struct time  currentTime, nextTime;

    printf ("Enter the time (hh:mm:ss): ");
    scanf ("%i:%i:%i", &currentTime.hour,
             &currentTime.minutes, &currentTime.seconds);

    nextTime = timeUpdate (currentTime);

    printf ("Updated time is %.2i:%.2i:%.2i\n", nextTime.hour,
                nextTime.minutes, nextTime.seconds );

    return 0;
}

// Function to update the time by one second

struct time  timeUpdate (struct time  now)
{
    ++now.seconds;
```

Program 9.5 **Continued**

```
    if ( now.seconds == 60 )  {     // next minute
        now.seconds = 0;
        ++now.minutes;

        if ( now.minutes == 60 ) {  // next hour
            now.minutes = 0;
            ++now.hour;

            if ( now.hour == 24 ) // midnight
               now.hour = 0;
        }
    }

    return now;
}
```

Program 9.5 **Output**

```
Enter the time (hh:mm:ss): 12:23:55
Updated time is 12:23:56
```

Program 9.5 **Output (Rerun)**

```
Enter the time (hh:mm:ss): 16:12:59
Updated time is 16:13:00
```

Program 9.5 **Output (Second Rerun)**

```
Enter the time (hh:mm:ss): 23:59:59
Updated time is 00:00:00
```

The `main` routine asks the user to enter in the time. The `scanf` call uses the format string

```
"%i:%i:%i"
```

to read the data. Specifying a nonformat character, such as `':'`, in a format string signals to the `scanf` function that the particular character is expected as input. Therefore, the format string listed in Program 9.5 specifies that three integer values are to be input—the first separated from the second by a colon, and the second separated from the third by a colon. In Chapter 16, "Input and Output Operations in C," you learn how the `scanf` function returns a value that can be tested to determine if the values were entered in the correct format.

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.

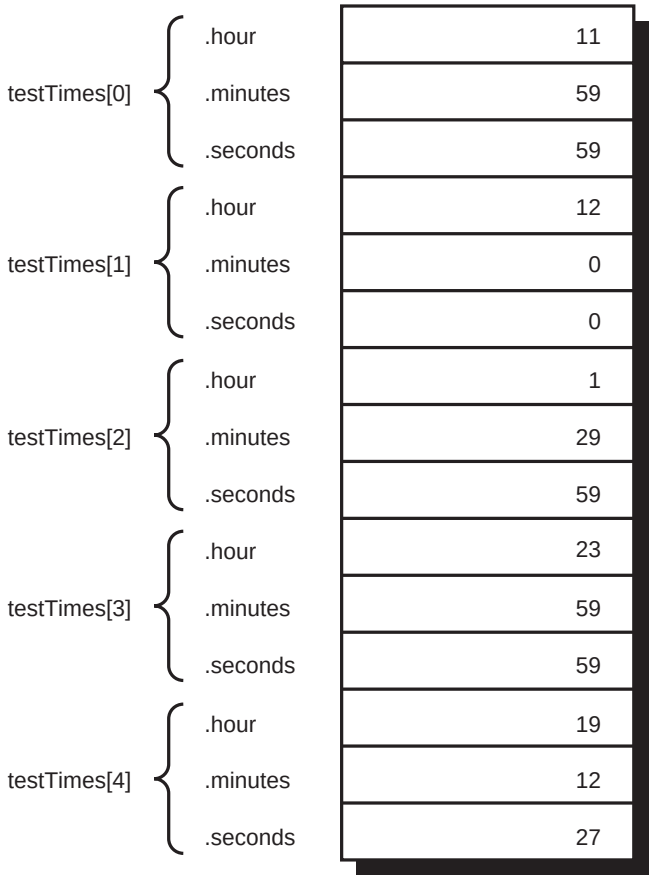

Figure 9.2 The array `testTimes` in memory.

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'`).

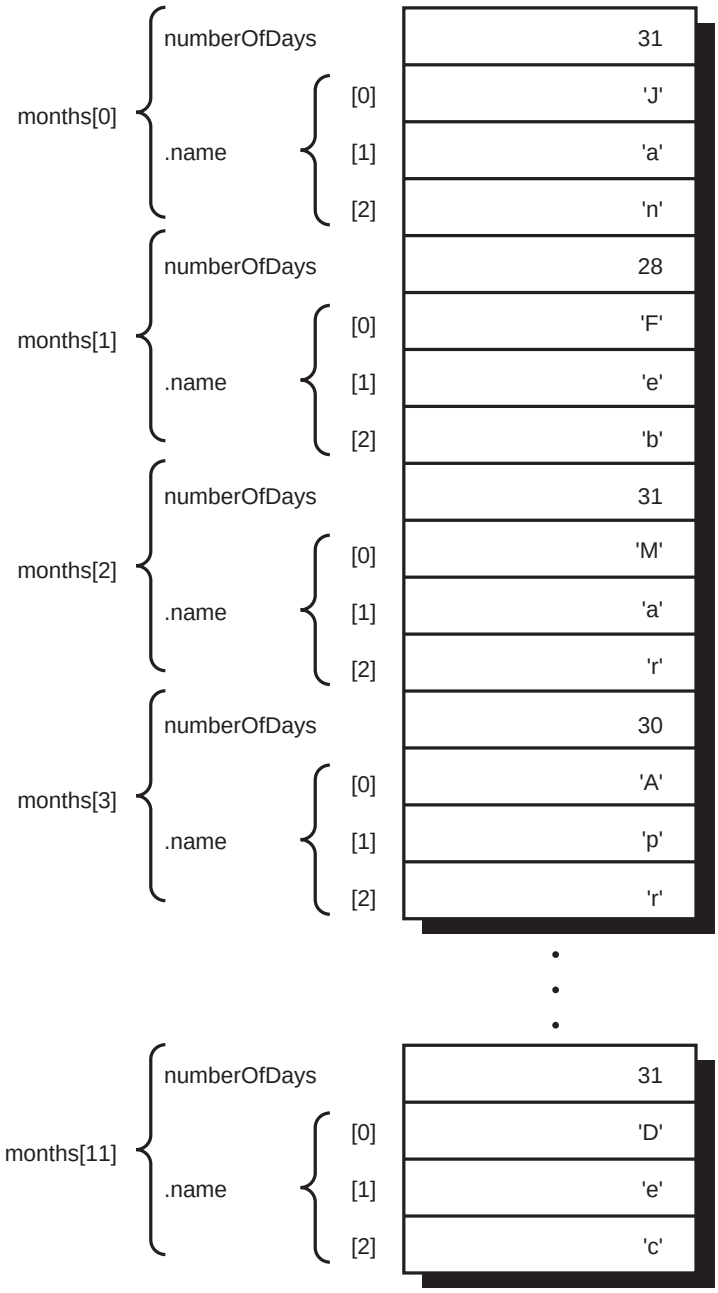

Figure 9.3 The array `months`.

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

```
struct  date
{
    int  month;
    int  day;
    int  year;
} todaysDate, purchaseDate;
```

defines the structure `date` *and* also declares the variables `todaysDate` and `purchaseDate` to be of this type. You can also assign initial values to the variables in the normal fashion. Thus,

```
struct  date
{
    int  month;
    int  day;
    int  year;
} todaysDate = { 1, 11, 2005 };
```

defines the structure `date` and the variable `todaysDate` with initial values as indicated.

If all of the variables of a particular structure type are defined when the structure is defined, the structure name can be omitted. So the statement

```
struct
{
    int  month;
    int  day;
    int  year;
}  dates[100];
```

defines an array called `dates` to consist of 100 elements. Each element is a structure containing three integer members: `month`, `day`, and `year`. Because you did not supply a name to the structure, the only way to subsequently declare variables of the same type is by explicitly defining the structure again.

You have seen how structures can be used to conveniently reference groups of data under a single label. You've also seen in this chapter how easily you can define arrays of structures and work with them with functions. In the next chapter, you learn how to work with arrays of characters, also known as character strings. Before going on, try the following exercises.

Exercises

1. Type in and run the seven programs presented in this chapter. Compare the output produced by each program with the output presented after each program in the text.
2. In certain applications, particularly in the financial area, it is often necessary to calculate the number of elapsed days between two dates. For example, the number of

days between July 2, 2005, and July 16, 2005, is obviously 14. But how many days are there between August 8, 2004, and February 22, 2005? This calculation requires a bit more thought.

Luckily, a formula can be used to calculate the number of days between two dates. This is affected by computing the value of *N* for each of the two dates and then taking the difference, where *N* is calculated as follows:

```
N = 1461 x f(year, month) / 4 + 153 x g(month) / 5  + day
```

where

```
f(year, month)    =    year - 1          if  month <= 2
                       year              otherwise


g(month)  =       month + 13        if month <= 2
                  month + 1         otherwise
```

As an example of applying the formula, to calculate the number of days between August 8, 2004, and February 22, 2005, you can calculate the values of N_1 and N_2 by substituting the appropriate values into the preceding formula as shown:

```
N1    = 1461 x f(2004, 8) / 4  +  153 x g(8) / 5  +  3
      = (1461 x 2004) / 4  +  (153 x 9) / 5  +  3
      = 2,927,844 / 4  +  1,377 / 5  +  3
      = 731,961 + 275 + 3
      = 732,239

N2    = 1461 x f(2005, 2) / 4  +  153 x g(2) / 5  +  21
      = (1461 x 2004) / 4  +  (153 x 15) / 5  +  21
      = 2,927,844 / 4  +  2295 / 5  +  21
      = 731,961 + 459 + 21
      = 732,441
```

Number of elapsed days = N_2 - N_1

```
                    = 732,441  – 732,239
                    = 202
```

So the number of days between the two dates is shown to be 202. The preceding formula is applicable for any dates after March 1, 1900 (1 must be added to *N* for dates from March 1, 1800, to February 28, 1900, and 2 must be added for dates between March 1, 1700, and February 28, 1800).

Write a program that permits the user to type in two dates and then calculates the number of elapsed days between the two dates. Try to structure the program logically into separate functions. For example, you should have a function that accepts as an argument a `date` structure and returns the value of *N* computed as shown

previously. This function can then be called twice, once for each date, and the difference taken to determine the number of elapsed days.

3. Write a function `elapsed_time` that takes as its arguments two `time` structures and returns a `time` structure that represents the elapsed time (in hours, minutes, and seconds) between the two times. So the call

   ```
   elapsed_time (time1, time2)
   ```

   where `time1` represents 3:45:15 and `time2` represents 9:44:03, should return a `time` structure that represents 5 hours, 58 minutes, and 48 seconds. Be careful with times that cross midnight.

4. If you take the value of *N* as computed in exercise 2, subtract 621,049 from it, and then take that result modulo 7, you get a number from 0 to 6 that represents the day of the week (Sunday through Saturday, respectively) on which the particular day falls. For example, the value of *N* computed for August 8, 2004, is 732,239 as derived previously. 732,239 – 621,049 gives 111,190, and 111,190 % 7 gives 2, indicating that this date falls on a Tuesday.

   Use the functions developed in the previous exercise to develop a program that displays the day of the week on which a particular date falls. Make certain that the program displays the day of the week in English (such as "Monday").

5. Write a function called `clockKeeper` that takes as its argument a `dateAndTime` structure as defined in this chapter. The function should call the `timeUpdate` function, and if the time reaches midnight, the function should call the `dateUpdate` function to switch over to the next day. Have the function return the updated `dateAndTime` structure.

6. Replace the `dateUpdate` function from Program 9.4 with the modified one that uses compound literals as presented in the text. Run the program to verify its proper operation.