Chapter 11 Structures

11-1. Simple Structures

A *structure* is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. The next section shows a simple example.

You should start with simple structures. Note that the C language makes no distinction between simple and complex structures, but it's easier to explain structures in this way.

11-1-1. Defining and Declaring Structures

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {
    int x;
    int y;
};
```

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or *tag* (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.

The preceding statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. In other words, they don't *declare* (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here:

```
struct coord {
    int x;
    int y;
} first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. first and second are each *instances* of type coord; first contains two integer members named x and y, and so does second.

This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord:

```
struct coord {
    int x;
    int y;
};
/* Additional code may go here */
struct coord first, second;
```

11-1-2. Accessing Structure Members

Individual structure members can be used like other variables of the same type. Structure members are accessed using the *structure member operator* (.), also called the *dot operator,* between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

The struct Keyword

```
struct tag {
    structure_member(s);
    /* additional statements may go here */
} instance;
```

The struct keyword is used to declare structures. A structure is a collection of one or more variables (*structure_members*) that have been grouped under a single name for easy man-ipulation. The variables don't have to be of the same variable type, nor do they have to be simple variables. Structures also can hold arrays, pointers, and other structures.

The keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure. Following the tag are the structure members, enclosed in braces. An *instance*, the actual declaration of a structure, can also be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. Here is a template's format:

```
struct tag {
    structure_member(s);
    /* additional statements may go here */
};
```

To use the template, you use the following format:

```
struct tag instance;
```

To use this format, you must have previously declared a structure with the given tag.

Example 1

```
/* Declare a structure template called SSN */
struct SSN {
    int first_three;
    char dash1;
    int second_two;
    char dash2;
    int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Example 2

```
/* Declare a structure and instance together */
struct date {
    char month[2];
    char day[2];
    char year[4];
} current_date;
```

Example 3

```
/* Declare and initialize a structure */
struct time {
    int hours;
    int minutes;
    int seconds;
} time_of_birth = { 8, 45, 0 };
```

11-2. More-Complex Structures

Now that you have been introduced to simple structures, you can get to the more interesting and complex types of structures. These are structures that contain other structures as members and structures that contain arrays as members.

11-2-1. Structures That Contain Structures

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this.

Assume that your graphics program needs to deal with rectangles. A rectangle can be defined by the coordinates of two diagonally opposite corners. You've already seen how to define a structure that can hold the two coordinates required for a single point. You need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle {
    struct coord topleft;
```

```
    struct coord bottomrt;
};
```

This statement defines a structure of type rectangle that contains two structures of type coord. These two type coord structures are named topleft and bottomrt.

The preceding statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as

```
struct rectangle mybox;
```

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
} mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus, the expression

```
mybox.topleft.x
```

refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Maybe this is getting a bit confusing. You might understand better if you look at Figure 11.1, which shows the relationship between the type rectangle structure, the two type coord structures it contains, and the two type int variables each type coord structure contains. These structures are named as in the preceding example.

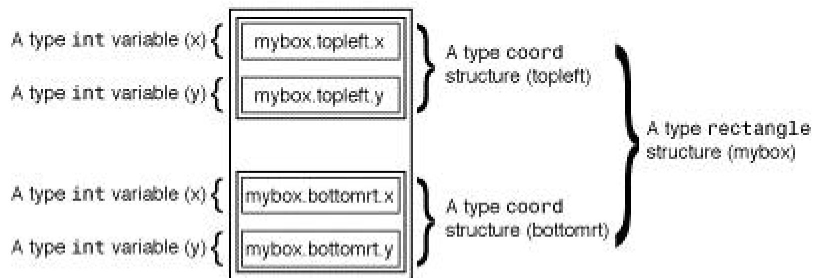

Figure 11-1. The relationship between a structure, structures within a structure, and the structure members.

Let's look at an example of using structures that contain other structures. Listing 11.1 takes input from the user for the coordinates of a rectangle and then calculates and displays the rectangle's area. Note the program's assumptions, given in comments near the start of the program (lines 3 through 8).

Listing 11.1. A demonstration of structures that contain other structures.

```
1:   /* Demonstrates structures that contain other structures. */
2:
3:   /* Receives input for corner coordinates of a rectangle and
4:      calculates the area. Assumes that the y coordinate of the
```

```
5:     upper-left corner is greater than the y coordinate of the
6:     lower-right corner, that the x coordinate of the lower-
7:     right corner is greater than the x coordinate of the upper-
8:     left corner, and that all coordinates are positive. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct rectangle{
21:     struct coord topleft;
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Input the coordinates */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
38:     printf("\nEnter the bottom right y coordinate: ");
39:     scanf("%d", &mybox.bottomrt.y);
40:
41:     /* Calculate the length and width */
42:
43:     width = mybox.bottomrt.x - mybox.topleft.x;
44:     length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:     /* Calculate and display the area */
47:
48:     area = width * length;
49:     printf("\nThe area is %ld units.\n", area);
50:
51:     return 0;
52: }
```

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
```

```
Enter the bottom right y coordinate: 10
The area is 81 units.
```

ANALYSIS: The coord structure is defined in lines 15 through 18 with its two members, x and y. Lines 20 through 23 declare and define an instance, called mybox, of the rectangle structure. The two members of the rectangle structure are topleft and bottomrt, both structures of type coord.

Lines 29 through 39 fill in the values in the mybox structure. At first it might seem that there are only two values to fill, because mybox has only two members. However, each of mybox's members has its own members. topleft and bottomrt have two members each, x and y from the coord structure. This gives a total of four members to be filled. After the members are filled with values, the area is calculated using the structure and member names. When using the x and y values, you must include the structure instance name. Because x and y are in a structure within a structure, you must use the instance names of both structures--mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x, and mybox.topleft.y--in the calculations.

C places no limits on the nesting of structures. While memory allows, you can define structures that contain structures that contain structures that contain structures--well, you get the idea! Of course, there's a limit beyond which nesting becomes unproductive. Rarely are more than three levels of nesting used in any C program.

11-2-2. Structures That Contain Arrays

You can define a structure that contains one or more arrays as members. The array can be of any C data type (int, char, and so on). For example, the statements

```
struct data{
    int  x[4];
    char y[10];
};
```

define a structure of type data that contains a four-element integer array member named x and a 10-element character array member named y. You can then declare a structure named record of type data as follows:

```
struct data record;
```

The organization of this structure is shown in Figure 11.2. Note that, in this figure, the elements of array x take up twice as much space as the elements of array y. This is because a type int typically requires two bytes of storage, whereas a type char usually requires only one byte (as you learned on Day 3, "Storing Data: Variables and Constants").

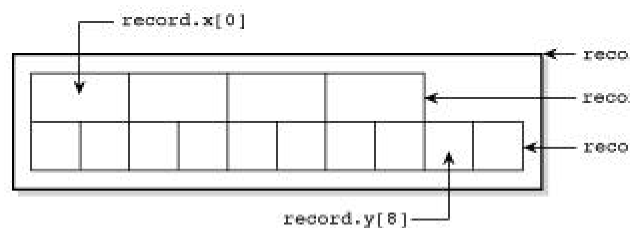

Figure 11-2. The organization of a structure that contains arrays as members.

You access individual elements of arrays that are structure members using a combination of the member operator and array subscripts:

```
record.x[2] = 100;
record.y[1] = `x';
```

You probably remember that character arrays are most frequently used to store strings. You should also remember (from Day 9, "Understanding Pointers") that the name of an array, without brackets, is a pointer to the array. Because this holds true for arrays that are structure members, the expression

```
record.y
```

is a pointer to the first element of array y[] in the structure record. Therefore, you could print the contents of y[] on-screen using the statement

```
puts(record.y);
```

Now look at another example. Listing 11.2 uses a structure that contains a type float variable and two type char arrays.

Listing 11.2. A structure that contains array members.

```
1:  /* Demonstrates a structure that has array members. */
2:
3:  #include <stdio.h>
4:
5:  /* Define and declare a structure to hold the data. */
6:  /* It contains one float variable and two char arrays. */
7:
8:  struct data{
9:      float amount;
10:     char fname[30];
11:     char lname[30];
12: } rec;
13:
14: main()
15: {
16:     /* Input the data from the keyboard. */
17:
18:     printf("Enter the donor's first and last names,\n");
19:     printf("separated by a space: ");
20:     scanf("%s %s", rec.fname, rec.lname);
21:
22:     printf("\nEnter the donation amount: ");
23:     scanf("%f", &rec.amount);
24:
25:     /* Display the information. */
26:     /* Note: %.2f specifies a floating-point value */
27:     /* to be displayed with two digits to the right */
28:     /* of the decimal point. */
29:
30:     /* Display the data on the screen. */
31:
32:     printf("\nDonor %s %s gave $%.2f.\n", rec.fname, rec.lname,
33:             rec.amount);
34:
35:     return 0;
36: }
```

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

ANALYSIS: This program includes a structure that contains array members named fname[30] and lname[30]. Both are arrays of characters that hold a person's first name and last name, respectively. The structure declared in lines 8 through 12 is called data. It contains the fname and lname character arrays with a type float variable called amount. This structure is ideal for holding a person's name (in two parts, first name and last name) and a value, such as the amount the person donated to a charitable organization.

An instance of the array, called rec, has also been declared in line 12. The rest of the program uses rec to get values from the user (lines 18 through 23) and then print them (lines 32 and 33).

11-3. Arrays of Structures

If you can have structures that contain arrays, can you also have arrays of structures? You bet you can! In fact, arrays of structures are very powerful programming tools. Here's how it's done.

You've seen how a structure definition can be tailored to fit the data your program needs to work with. Usually a program needs to work with more than one instance of the data. For example, in a program to maintain a list of phone numbers, you can define a structure to hold each person's name and number:

```
struct entry{
    char fname[10];
    char lname[12];
    char phone[8];
};
```

A phone list must hold many entries, however, so a single instance of the entry structure isn't of much use. What you need is an array of structures of type entry. After the structure has been defined, you can declare an array as follows:

```
struct entry list[1000];
```

This statement declares an array named list that contains 1,000 elements. Each element is a structure of type entry and is identified by subscript like other array element types. Each of these structures has three elements, each of which is an array of type char. This entire complex creation is diagrammed in Figure 11.3.

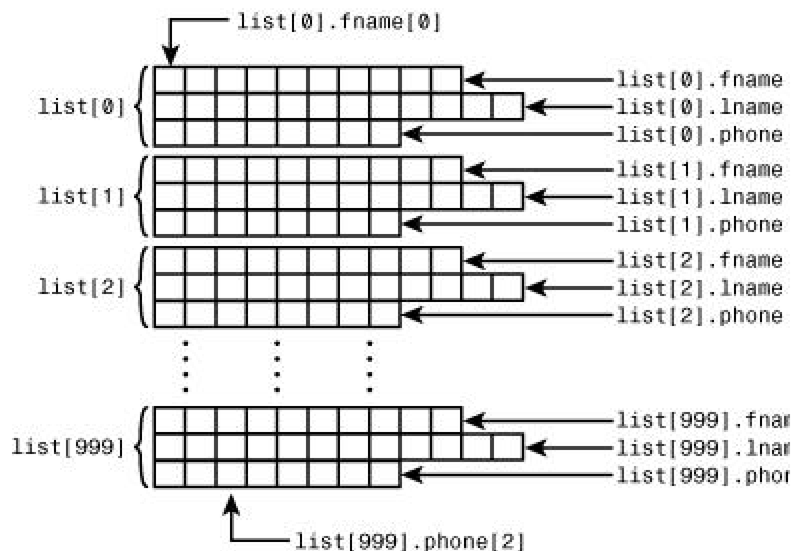

Figure 11-3. The organization of the array of structures defined in the text.

When you have declared the array of structures, you can manipulate the data in many ways. For example, to assign the data in one array element to another array element, you would write

```
list[1] = list[5];
```

This statement assigns to each member of the structure list[1] the values contained in the corresponding members of list[5]. You can also move data between individual structure members. The statement

```
strcpy(list[1].phone, list[5].phone);
```

copies the string in list[5].phone to list[1].phone. (The strcpy() library function copies one string to another string. You'll learn the details of this on Day 17, "Manipulating Strings.") If you want to, you can also move data between individual elements of the structure member arrays:

```
list[5].phone[1] = list[2].phone[3];
```

This statement moves the second character of list[5]'s phone number to the fourth position in list[2]'s phone number. (Don't forget that subscripts start at offset 0.)

Listing 11.3 demonstrates the use of arrays of structures. Moreover, it demonstrates arrays of structures that contain arrays as members.

Listing 11.3. Arrays of structures.

```
1:  /* Demonstrates using arrays of structures. */
2:
3:  #include <stdio.h>
4:
5:  /* Define a structure to hold entries. */
6:
7:  struct entry {
8:      char fname[20];
9:      char lname[20];
10:     char phone[10];
11: };
12:
13: /* Declare an array of structures. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22:     /* Loop to input data for four people. */
23:
24:     for (i = 0; i < 4; i++)
25:     {
26:         printf("\nEnter first name: ");
27:         scanf("%s", list[i].fname);
28:         printf("Enter last name: ");
29:         scanf("%s", list[i].lname);
30:         printf("Enter phone in 123-4567 format: ");
31:         scanf("%s", list[i].phone);
32:     }
33:
34:     /* Print two blank lines. */
35:
36:     printf("\n\n");
37:
38:     /* Loop to display data. */
```

```
39:
40:     for (i = 0; i < 4; i++)
41:     {
42:         printf("Name: %s %s", list[i].fname, list[i].lname);
43:         printf("\t\tPhone: %s\n", list[i].phone);
44:     }
45:
46:     return 0;
47: }
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434
Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Deanna
Enter last name: Townsend
Enter phone in 123-4567 format: 555-1234
Name: Bradley Jones             Phone: 555-1212
Name: Peter Aitken              Phone: 555-3434
Name: Melissa Jones             Phone: 555-1212
Name: Deanna Townsend           Phone: 555-1234
```

ANALYSIS: This listing follows the same general format as most of the other listings. It starts with the comment in line 1 and, for the input/output functions, the #include file STDIO.H in line 3. Lines 7 through 11 define a template structure called entry that contains three character arrays: fname, lname, and phone. Line 15 uses the template to define an array of four entry structure variables called list. Line 17 defines a variable of type int to be used as a counter throughout the program. main() starts in line 19. The first function of main() is to perform a loop four times with a for statement. This loop is used to get information for the array of structures. This can be seen in lines 24 through 32. Notice that list is being used with a subscript in the same way as the array variables on Day 8, "Using Numeric Arrays," were subscripted.

Line 36 provides a break from the input before starting with the output. It prints two new lines in a manner that shouldn't be new to you. Lines 40 through 44 display the data that the user entered in the preceding step. The values in the array of structures are printed with the subscripted array name followed by the member operator (.) and the structure member name.

Familiarize yourself with the techniques used in Listing 11.3. Many real-world programming tasks are best accomplished by using arrays of structures containing arrays as members.

DON'T forget the structure instance name and member operator (.) when using a structure's members.

DON'T confuse a structure's tag with its instances! The tag is used to declare the structure's template, or format. The instance is a variable declared using the tag.

DON'T forget the struct keyword when declaring an instance from a previously defined structure.

DO declare structure instances with the same scope rules as other variables. (Day 12, "Understanding Variable Scope," covers this topic fully.)

11-4. Initializing Structures

Like other C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values separated by commas and enclosed in braces. For example, look at the following statements:

```
1: struct sale {
2:     char customer[20];
3:     char item[20];
4:     float amount;
5: } mysale = { "Acme Industries",
6:              "Left-handed widget",
7:               1000.00
8:            };
```

When these statements are executed, they perform the following actions:

1. Define a structure type named sale (lines 1 through 5).
2. Declare an instance of structure type sale named mysale (line 5).
3. Initialize the structure member mysale.customer to the string "Acme Industries" (line 5).
4. Initialize the structure member mysale.item to the string "Left-handed widget" (line 6).
5. Initialize the structure member mysale.amount to the value 1000.00 (line 7).

For a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

```
1:  struct customer {
2:      char firm[20];
3:      char contact[25];
4:  }
5:
6:  struct sale {
7:     struct customer buyer;
8:     char item[20];
9:     float amount;
10: } mysale = { { "Acme Industries", "George Adams"},
11:                "Left-handed widget",
12:                 1000.00
13:            };
```

These statements perform the following initializations:

1. The structure member mysale.buyer.firm is initialized to the string "Acme Industries" (line 10).
2. The structure member mysale.buyer.contact is initialized to the string "George Adams" (line 10).
3. The structure member mysale.item is initialized to the string "Left-handed widget" (line 11).
4. The structure member mysale.amount is initialized to the amount 1000.00 (line 12).

You can also initialize arrays of structures. The initialization data that you supply is applied, in order, to the structures in the array. For example, to declare an array of structures of type sale and initialize the first two array elements (that is, the first two structures), you could write

```
1:  struct customer {
2:      char firm[20];
3:      char contact[25];
4:  };
5:
6:  struct sale {
7:      struct customer buyer;
8:      char item[20];
9:      float amount;
10: };
11:
12:
13: struct sale y1990[100] = {
14:      { { "Acme Industries", "George Adams"},
15:          "Left-handed widget",
16:           1000.00
17:      }
18:      { { "Wilson & Co.", "Ed Wilson"},
19:          "Type 12 gizmo",
20:           290.00
21:      }
22: };
```

This is what occurs in this code:

1. The structure member y1990[0].buyer.firm is initialized to the string "Acme Industries" (line 14).
2. The structure member y1990[0].buyer.contact is initialized to the string "George Adams" (line 14).
3. The structure member y1990[0].item is initialized to the string "Left-handed widget" (line 15).
4. The structure member y1990[0].amount is initialized to the amount 1000.00 (line 16).
5. The structure member y1990[1].buyer.firm is initialized to the string "Wilson & Co." (line 18).
6. The structure member y1990[1].buyer.contact is initialized to the string "Ed Wilson" (line 18).
7. The structure member y1990[1].item is initialized to the string "Type 12 gizmo" (line 19).
8. The structure member y1990[1].amount is initialized to the amount 290.00 (line 20).

11-5. Structures and Pointers

Given that pointers are such an important part of C, you shouldn't be surprised to find that they can be used with structures. You can use pointers as structure members, and you can also declare pointers to structures. These topics are covered in the following sections.

11-5-1. Pointers as Structure Members

You have complete flexibility in using pointers as structure members. Pointer members are declared in the same manner as pointers that aren't members of structures--that is, by using the indirection operator (*). Here's an example:

```
struct data {
    int *value;
    int *rate;
} first;
```

These statements define and declare a structure whose two members are both pointers to type int. As with all pointers, declaring them is not enough; you must, by assigning them the address of a variable, initialize them to point to something. If cost and interest have been declared to be type int variables, you could write

```
first.value = &cost;
first.rate = &interest;
```

Now that the pointers have been initialized, you can use the indirection operator (*), as explained on Day 9. The expression *first.value evaluates to the value of cost, and the expression *first.rate evaluates to the value of interest.

Perhaps the type of pointer most frequently used as a structure member is a pointer to type char. Recall from Day 10, "Characters and Strings," that a string is a sequence of characters delineated by a pointer that points to the string's first character and a null character that indicates the end of the string. To refresh your memory, you can declare a pointer to type char and initialize it to point at a string as follows:

```
char *p_message;
p_message = "Teach Yourself C In 21 Days";
```

You can do the same thing with pointers to type char that are structure members:

```
struct msg {
    char *p1;
    char *p2;
} myptrs;
myptrs.p1 = "Teach Yourself C In 21 Days";
myptrs.p2 = "By SAMS Publishing";
```

Figure 11.4 illustrates the result of executing these statements. Each pointer member of the structure points to the first byte of a string, stored elsewhere in memory. Contrast this with Figure 11.3, which shows how data is stored in a structure that contains arrays of type char.

You can use pointer structure members anywhere a pointer can be used. For example, to print the pointed-to strings, you would write

```
printf("%s %s", myptrs.p1, myptrs.p2);
```

What's the difference between using an array of type char as a structure member and using a pointer to type char? These are both methods for "storing" a string in a structure, as shown here in the structure msg, which uses both methods:

```
struct msg {
    char p1[30];
    char *p2;
} myptrs;
```

Recall that an array name without brackets is a pointer to the first array element. Therefore, you can use these two structure members in similar fashion:

```
strcpy(myptrs.p1, "Teach Yourself C In 21 Days");
strcpy(myptrs.p2, "By SAMS Publishing");
```

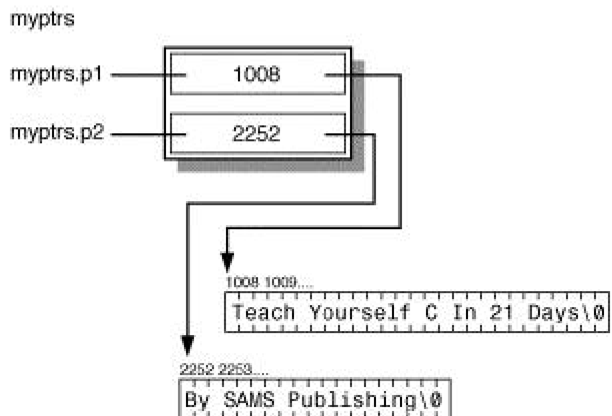

Figure 11-4. A structure that contains pointers to type char.

```
/* additional code goes here */
puts(myptrs.p1);
puts(myptrs.p2);
```

What's the difference between these methods? It is this: If you define a structure that contains an array of type char, every instance of that structure type contains storage space for an array of the specified size. Furthermore, you're limited to the specified size; you can't store a larger string in the structure. Here's an example:

```
struct msg {
    char p1[10];
    char p2[10];
} myptrs;
...
strcpy(p1, "Minneapolis"); /* Wrong! String longer than array.*/
strcpy(p2, "MN");          /* OK, but wastes space because    */
                           /* string shorter than array.      */
```

If, on the other hand, you define a structure that contains pointers to type char, these restrictions don't apply. Each instance of the structure contains storage space for only the pointer. The actual strings are stored elsewhere in memory (but you don't need to worry about *where* in memory). There's no length restriction or wasted space. The actual strings aren't stored as part of the structure. Each pointer in the structure can point to a string of any length. That string becomes part of the structure, even though it isn't stored in the structure.

11-5-2. Pointers to Structures

A C program can declare and use pointers to structures, just as it can declare pointers to any other data storage type. As you'll see later in this chapter, pointers to structures are often used when passing a structure as an argument to a function. Pointers to structures are also used in a very powerful data storage method known as *linked lists*. Linked lists are explored on Day 15, "Pointers: Beyond the Basics."

For now, take a look at how your program can create and use pointers to structures. First, define a structure:

```
struct part {
    int number;
    char name[10];
};
```

Now declare a pointer to type part:

```
struct part *p_part;
```

Remember, the indirection operator (*) in the declaration says that p_part is a pointer to type part, not an instance of type part.

Can the pointer be initialized now? No, because even though the structure part has been defined, no instances of it have been declared. Remember that it's a declaration, not a definition, that sets aside storage space in memory for a data object. Because a pointer needs a memory address to point to, you must declare an instance of type part before anything can point to it. Here's the declaration:

```
struct part gizmo;
```

Now you can perform the pointer initialization:

```
p_part = &gizmo;
```

This statement assigns the address of gizmo to p_part. (Recall the address-of operator, &, from Day 9.) Figure 11.5 shows the relationship between a structure and a pointer to the structure.

Now that you have a pointer to the structure gizmo, how do you make use of it? One method uses the indirection operator (*). Recall from Day 9 that if ptr is a pointer to a data object, the expression *ptr refers to the object pointed to.

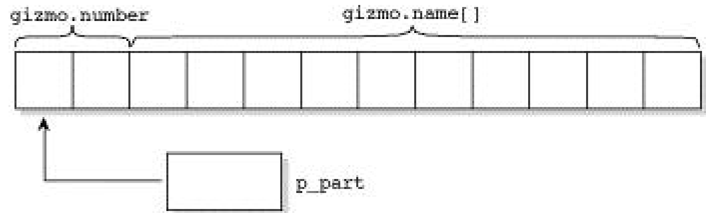

Figure 11-5. A pointer to a structure points to the structure's first byte.

Applying this to the current example, you know that p_part is a pointer to the structure gizmo, so *p_part refers to gizmo. You then apply the structure member operator (.) to access individual members of gizmo. To assign the value 100 to gizmo.number, you could write

```
(*p_part).number = 100;
```

*p_part must be enclosed in parentheses because the (.) operator has a higher precedence than the (*) operator.

A second way to access structure members using a pointer to the structure is to use the *indirect membership operator,* which consists of the characters -> (a hyphen followed by the greater-than symbol). (Note that when they are used together in this way, C treats them as a single operator, not two.) This symbol is placed between the pointer name and the member name. For example, to access the number member of gizmo with the p_part pointer, you would write

```
p_part->number
```

Looking at another example, if str is a structure, p_str is a pointer to str, and memb is a member of str, you can access str.memb by writing

```
p_str->memb
```

Therefore, there are three ways to access a structure member:

- Using the structure name
- Using a pointer to the structure with the indirection operator (*)
- Using a pointer to the structure with the indirect membership operator (->)

If p_str is a pointer to the structure str, the following expressions are all equivalent:

```
str.memb
(*p_str).memb
p_str->memb
```

11-5-3. Pointers and Arrays of Structures

You've seen that arrays of structures can be a very powerful programming tool, as can pointers to structures. You can combine the two, using pointers to access structures that are array elements.

To illustrate, here is a structure definition from an earlier example:

```
struct part {
    int number;
```

```
    char name[10];
};
```

After the part structure is defined, you can declare an array of type part:

```
struct part data[100];
```

Next you can declare a pointer to type part and initialize it to point to the first structure in the array data:

```
struct part *p_part;
p_part = &data[0];
```

Recall that the name of an array without brackets is a pointer to the first array element, so the second line could also have been written as

```
p_part = data;
```

You now have an array of structures of type part and a pointer to the first array element (that is, the first structure in the array). For example, you could print the contents of the first element using the statement

```
printf("%d %s", p_part->number, p_part->name);
```

What if you wanted to print all the array elements? You would probably use a for loop, printing one array element with each iteration of the loop. To access the members using pointer notation, you must change the pointer p_part so that with each iteration of the loop it points to the next array element (that is, the next structure in the array). How do you do this?

C's pointer arithmetic comes to your aid. The unary increment operator (++) has a special meaning when applied to a pointer: It means "increment the pointer by the size of the object it points to." Put another way, if you have a pointer ptr that points to a data object of type obj, the statement

```
ptr++;
```

has the same effect as

```
ptr += sizeof(obj);
```

This aspect of pointer arithmetic is particularly relevant to arrays because array elements are stored sequentially in memory. If a pointer points to array element *n*, incrementing the pointer with the (++) operator causes it to point to element *n* + 1. This is illustrated in Figure 11.6, which shows an array named x[] that consists of four-byte elements (for example, a structure containing two type int members, each two bytes long). The pointer ptr was initialized to point to x[0]; each time ptr is incremented, it points to the next array element.

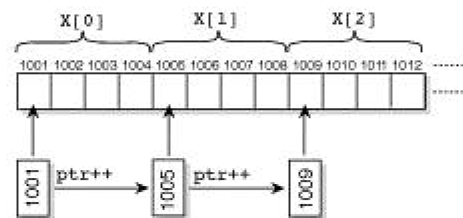

Figure 11-6. With each increment, a pointer steps to the next array element.

What this means is that your program can step through an array of structures (or an array of any other data type, for that matter) by incrementing a pointer. This sort of notation is usually easier to use and more concise than using array subscripts to perform the same task. Listing 11.4 shows how you do this.

Listing 11.4. Accessing successive array elements by incrementing a pointer.

```
1:   /* Demonstrates stepping through an array of structures */
```

```
2:  /* using pointer notation. */
3:
4:  #include <stdio.h>
5:
6:  #define MAX 4
7:
8:  /* Define a structure, then declare and initialize */
9:  /* an array of four structures. */
10:
11: struct part {
12:     int number;
13:     char name[10];
14: } data[MAX] = {1, "Smith",
15:                2, "Jones",
16:                3, "Adams",
17:                4, "Wilson"
18:                };
19:
20: /* Declare a pointer to type part, and a counter variable. */
21:
22: struct part *p_part;
23: int count;
24:
25: main()
26: {
27:     /* Initialize the pointer to the first array element. */
28:
29:     p_part = data;
30:
31:     /* Loop through the array, incrementing the pointer */
32:     /* with each iteration. */
33:
34:     for (count = 0; count < MAX; count++)
35:     {
36:         printf("At address %d: %d %s\n", p_part, p_part->number,
37:                 p_part->name);
38:         p_part++;
39:     }
40:
41:     return 0;
42: }
At address 96: 1 Smith
At address 108: 2 Jones
At address 120: 3 Adams
```

At address 132: 4 Wilson**ANALYSIS:** First, in lines 11 through 18, this program declares and initializes an array of structures called data. A pointer called p_part is then defined in line 22 to be used to point to the data structure. The main() function's first task in line 29 is to set the pointer, p_part, to point to the part structure that was declared. All the elements are then printed using a for loop in lines 34 through 39 that increments the pointer to the array with each iteration. The program also displays the address of each element.

Look closely at the addresses displayed. The precise values might differ on your system, but they are in equal-sized increments--just the size of the structure part (most systems will have an increment of 12). This clearly illustrates that incrementing a pointer increases it by an amount equal to the size of the data object it points to.

11-5-4. Passing Structures as Arguments to Functions

Like other data types, a structure can be passed as an argument to a function. Listing 11.5 shows how to do this. This program is a modification of the program shown in Listing 11.2. It uses a function to display data on the screen, whereas Listing 11.2 uses statements that are part of main().

Listing 11.5. Passing a structure as a function argument.

```
1:  /* Demonstrates passing a structure to a function. */
2:
3:  #include <stdio.h>
4:
5:  /* Declare and define a structure to hold the data. */
6:
7:  struct data{
8:      float amount;
9:      char fname[30];
10:     char lname[30];
11: } rec;
12:
13: /* The function prototype. The function has no return value, */
14: /* and it takes a structure of type data as its one argument. */
15:
16: void print_rec(struct data x);
17:
18: main()
19: {
20:     /* Input the data from the keyboard. */
21:
22:     printf("Enter the donor's first and last names,\n");
23:     printf("separated by a space: ");
24:     scanf("%s %s", rec.fname, rec.lname);
25:
26:     printf("\nEnter the donation amount: ");
27:     scanf("%f", &rec.amount);
28:
29:     /* Call the display function. */
30:     print_rec( rec );
31:
32:     return 0;
33: }
34: void print_rec(struct data x)
35: {
36:     printf("\nDonor %s %s gave $%.2f.\n", x.fname, x.lname,
37:             x.amount);
38: }
```

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
```

```
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

ANALYSIS: Looking at line 16, you see the function prototype for the function that is to receive the structure. As you would with any other data type that was going to be passed, you need to include the proper arguments. In this case, it is a structure of type data. This is repeated in the header for the function in line 34. When calling the function, you only need to pass the structure instance name--in this case, rec (line 30). That's all there is to it. Passing a structure to a function isn't very different from passing a simple variable.

You can also pass a structure to a function by passing the structure's address (that is, a pointer to the structure). In fact, in older versions of C, this was the only way to pass a structure as an argument. It's not necessary now, but you might see older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (->) to access structure members in the function.

DON'T confuse arrays with structures!

DO take advantage of declaring a pointer to a structure--especially when using arrays of structures.

DON'T forget that when you increment a pointer, it moves a distance equivalent to the size of the data to which it points. In the case of a pointer to a structure, this is the size of the structure.

DO use the indirect membership operator (->) when working with a pointer to a structure.

11-6. Unions

Unions are similar to structures. A union is declared and used in the same ways that a structure is. A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the same area of memory. They are laid on top of each other.

11-6-1. Defining, Declaring, and Initializing Unions

Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct. To define a simple union of a char variable and an integer variable, you would write the following:

```
union shared {
    char c;
    int i;
};
```

This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. Unlike a structure that would hold both values, the union can hold only one value at a time. Figure 11.7 illustrates how the shared union would appear in memory.

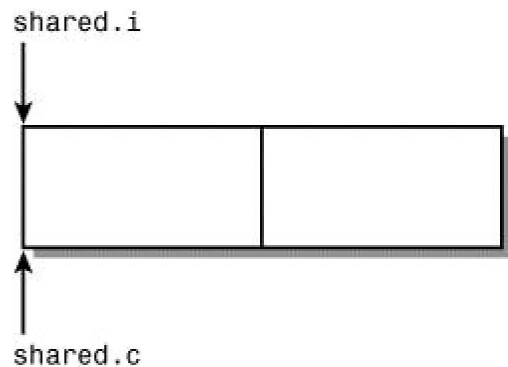

Figure 11-7. The union can hold only one value at a time.

A union can be initialized on its declaration. Because only one member can be used at a time, only one can be initialized. To avoid confusion, only the first member of the union can be initialized. The following code shows an instance of the shared union being declared and initialized:

```
union shared generic_variable = {`@'};
```

Notice that the generic_variable union was initialized just as the first member of a structure would be initialized.

Accessing Union Members

Individual union members can be used in the same way that structure members can be used--by using the member operator (.). However, there is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Listing 11.6 presents an example.

Listing 11.6. An example of the wrong use of unions.

```
1:    /* Example of using more than one union member at a time */
2:    #include <stdio.h>
3:
4:    main()
5:    {
6:        union shared_tag {
7:            char   c;
8:            int    i;
9:            long   l;
10:           float  f;
11:           double d;
12:       } shared;
13:
14:       shared.c = `$';
15:
16:       printf("\nchar c   = %c",  shared.c);
17:       printf("\nint i    = %d",  shared.i);
18:       printf("\nlong l   = %ld", shared.l);
19:       printf("\nfloat f  = %f",  shared.f);
20:       printf("\ndouble d = %f",  shared.d);
21:
22:       shared.d = 123456789.8765;
23:
24:       printf("\n\nchar c   = %c",  shared.c);
25:       printf("\nint i    = %d",  shared.i);
26:       printf("\nlong l   = %ld", shared.l);
27:       printf("\nfloat f  = %f",  shared.f);
28:       printf("\ndouble d = %f\n",  shared.d);
29:
30:       return 0;
31:   }
char c   = $
int i    = 4900
long l   = 437785380
float f  = 0.000000
double d = 0.000000
char c   = 7
int i    = -30409
long l   = 1468107063
```

```
float f   = 284852666499072.000000
double d = 123456789.876500
```

ANALYSIS: In this listing, you can see that a union named shared is defined and declared in lines 6 through 12. shared contains five members, each of a different type. Lines 14 and 22 initialize individual members of shared. Lines 16 through 20 and 24 through 28 then present the values of each member using printf() statements.

Note that, with the exceptions of char c = $ and double d = 123456789.876500, the output might not be the same on your computer. Because the character variable, c, was initialized in line 14, it is the only value that should be used until a different member is initialized. The results of printing the other union member variables (i, l, f, and d) can be unpredictable (lines 16 through 20). Line 22 puts a value into the double variable, d. Notice that the printing of the variables again is unpredictable for all but d. The value entered into c in line 14 has been lost because it was overwritten when the value of d in line 22 was entered. This is evidence that the members all occupy the same space.

The union Keyword

```
union tag {
    union_member(s);
    /* additional statements may go here */
}instance;
```

The union keyword is used for declaring unions. A union is a collection of one or more variables (*union_members*) that have been grouped under a single name. In addition, each of these union members occupies the same area of memory.

The keyword union identifies the beginning of a union definition. It's followed by a tag that is the name given to the union. Following the tag are the union members enclosed in braces. An *instance*, the actual declaration of a union, also can be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. The following is a template's format:

```
union tag {
    union_member(s);
    /* additional statements may go here */
};
```

To use the template, you would use the following format:

```
union tag instance;
```

To use this format, you must have previously declared a union with the given tag.

Example 1

```
/* Declare a union template called tag */
union tag {
    int nbr;
    char character;
}
/* Use the union template */
union tag mixed_variable;
```

Example 2

```
/* Declare a union and instance together */
union generic_type_tag {
    char c;
    int i;
    float f;
    double d;
} generic;
```

Example 3

```
/* Initialize a union. */
union date_tag {
    char full_date[9];
    struct part_date_tag {
        char month[2];
        char break_value1;
        char day[2];
        char break_value2;
        char year[2];
    } part_date;
}date = {"01/01/97"};
```

Listing 11.7 shows a more practical use of a union. Although this use is simplistic, it's one of the more common uses of a union.

Listing 11.7. A practical use of a union.

```
1:   /* Example of a typical use of a union */
2:
3:   #include <stdio.h>
4:
5:   #define CHARACTER   `C'
6:   #define INTEGER     `I'
7:   #define FLOAT       `F'
8:
9:   struct generic_tag{
10:      char type;
11:      union shared_tag {
12:          char   c;
13:          int    i;
14:          float  f;
15:      } shared;
16:  };
17:
18:  void print_function( struct generic_tag generic );
19:
20:  main()
21:  {
22:      struct generic_tag var;
23:
24:      var.type = CHARACTER;
25:      var.shared.c = `$';
```

```
26:       print_function( var );
27:
28:       var.type = FLOAT;
29:       var.shared.f = (float) 12345.67890;
30:       print_function( var );
31:
32:       var.type = `x';
33:       var.shared.i = 111;
34:       print_function( var );
35:       return 0;
36:   }
37:   void print_function( struct generic_tag generic )
38:   {
39:       printf("\n\nThe generic value is...");
40:       switch( generic.type )
41:       {
42:           case CHARACTER: printf("%c",  generic.shared.c);
43:                           break;
44:           case INTEGER:   printf("%d",  generic.shared.i);
45:                           break;
46:           case FLOAT:     printf("%f",  generic.shared.f);
47:                           break;
48:           default:        printf("an unknown type: %c\n",
49:                                   generic.type);
50:                           break;
51:       }
52:   }
The generic value is...$
The generic value is...12345.678711
The generic value is...an unknown type: x
```

ANALYSIS: This program is a very simplistic version of what could be done with a union. This program provides a way of storing multiple data types in a single storage space. The generic_tag structure lets you store either a character, an integer, or a floating-point number within the same area. This area is a union called shared that operates just like the examples in Listing 11.6. Notice that the generic_tag structure also adds an additional field called type. This field is used to store information on the type of variable contained in shared. type helps prevent shared from being used in the wrong way, thus helping to avoid erroneous data such as that presented in Listing 11.6.

A formal look at the program shows that lines 5, 6, and 7 define constants CHARACTER, INTEGER, and FLOAT. These are used later in the program to make the listing more readable. Lines 9 through 16 define a generic_tag structure that will be used later. Line 18 presents a prototype for the print_function(). The structure var is declared in line 22 and is first initialized to hold a character value in lines 24 and 25. A call to print_function() in line 26 lets the value be printed. Lines 28 through 30 and 32 through 34 repeat this process with other values.

The print_function() is the heart of this listing. Although this function is used to print the value from a generic_tag variable, a similar function could have been used to initialize it. print_function() will evaluate the type variable in order to print a statement with the appropriate variable type. This prevents getting erroneous data such as that in Listing 11.6.

DON'T try to initialize more than the first union member.

DO remember which union member is being used. If you fill in a member of one type and then try to use a different type, you can get unpredictable results.

DON'T forget that the size of a union is equal to its largest member.

DO note that unions are an advanced C topic.

11-7. typedef and Structures

You can use the typedef keyword to create a synonym for a structure or union type. For example, the following statements define coord as a synonym for the indicated structure:

```
typedef struct {
    int x;
    int y;
} coord;
```

You can then declare instances of this structure using the coord identifier:

```
coord topleft, bottomright;
```

Note that a typedef is different from a structure tag, as described earlier in this chapter. If you write

```
struct coord {
    int x;
    int y;
};
```

the identifier coord is a tag for the structure. You can use the tag to declare instances of the structure, but unlike with a typedef, you must include the struct keyword:

```
struct coord topleft, bottomright;
```

Whether you use typedef or a structure tag to declare structures makes little difference. Using typedef results in slightly more concise code, because the struct keyword doesn't need to be used. On the other hand, using a tag and having the struct keyword explicit makes it clear that it is a structure being declared.

11-8. Summary

This chapter showed you how to use structures, a data type that you design to meet the needs of your program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a *member,* is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and they can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

Q&A

Q Is there any reason to declare a structure without an instance?
A I showed you two ways of declaring a structure. The first was to declare a structure body, tag, and instance all at once. The second was to declare a structure body and tag without an instance. An instance can then be declared later by using the struct keyword, the tag, and a name for the instance. It's common programming practice to use the second method. Many programmers declare the structure body and tag without any instances. The instances are then declared later in the program. The next chapter describes variable scope. Scope will apply to the instance, but not to the tag or structure body.
Q Is it more common to use a typedef or a structure tag?
A Many programmers use typedefs to make their code easier to read, but it makes little practical difference. Many add-in libraries that contain functions are available for purchase. These add-ins usually have a lot of typedefs to make the product unique. This is especially true of database add-in products.
Q Can I simply assign one structure to another with the assignment operator?
A Yes and no. Newer versions of C compilers let you assign one structure to another, but older versions might not. In older versions of C, you might need to assign each member of the structures individually! This is true of unions also.
Q How big is a union?
A Because each of the members in a union is stored in the same memory location, the amount of room required to store the union is equal to that of its largest member.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. How is a structure different from an array?
2. What is the structure member operator, and what purpose does it serve?
3. What keyword is used in C to create a structure?
4. What is the difference between a structure tag and a structure instance?
5. What does the following code fragment do?

```
struct address {
    char name[31];
    char add1[31];
    char add2[31];
    char city[11];
    char state[3];
    char zip[11];
} myaddress = { "Bradley Jones",
                "RTSoftware",
                "P.O. Box 1213",
                "Carmel", "IN", "46032-1213"};
```

6. Assume you have declared an array of structures and that ptr is a pointer to the first array element (that is, the first structure in the array). How would you change ptr to point to the second array element?

Exercises

1. Write code that defines a structure named time, which contains three int members.
2. Write code that performs two tasks: defines a structure named data that contains one type int member and two type float members, and declares an instance of type data named info.
3. Continuing with exercise 2, how would you assign the value 100 to the integer member of the structure info?
4. Write code that declares and initializes a pointer to info.
5. Continuing with exercise 4, show two ways of using pointer notation to assign the value 5.5 to the first float member of info.
6. Write the definition for a structure type named data that can hold a single string of up to 20 characters.

7. Create a structure containing five strings: address1, address2, city, state, and zip. Create a typedef called RECORD that can be used to create instances of this structure.

8. Using the typedef from exercise 7, allocate and initialize an element called myaddress.

9. BUG BUSTER: What is wrong with the following code?

```
struct {
    char zodiac_sign[21];
    int month;
} sign = "Leo", 8;
```

10. BUG BUSTER: What is wrong with the following code?

```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW", 1000 };
```