14

Structures and Other Data Forms

You will learn about the following in this chapter:

- Keywords:

  `struct`, `union`, `typedef`

- Operators:

  `. ->`

- What C structures are and how to create structure templates and variables
- How to access the members of a structure and how to write functions to handle structures
- C's `typedef` facility
- Unions and pointers to functions

One of the most important steps in designing a program is choosing a good way to represent the data. In many cases, a simple variable or even an array is not enough. C takes your ability to represent data a step further with the C *structure variables*. The C structure is flexible enough in its basic form to represent a diversity of data, and it enables you to invent new forms. If you are familiar with Pascal records, you should be comfortable with structures. If not, this chapter will introduce you to C structures. Let's study a concrete example to see why a C structure might be needed and how to create and use one.

Sample Problem: Creating an Inventory of Books

Gwen Glenn wants to print an inventory of her books. She would like to print a variety of information for each book: title, author, publisher, copyright date, the number of pages, the number of copies, and the dollar value. Some of these items, such as the titles, can be stored in an array of strings. Other items require an array of `int`s or an array of `float`s. With seven

different arrays, keeping track of everything can get complicated, especially if Gwen wants to generate several complete lists—one sorted by title, one sorted by author, one sorted by value, and so on. A better solution is to use one array, in which each member contains all the information about one book.

Gwen needs a data form, then, that can contain both strings and numbers and somehow keep the information separate. The C structure meets this need. To see how a structure is set up and how it works, we'll start with a limited example. To simplify the problem, we will impose two restrictions. First, we'll include only title, author, and current market value. Second, we'll limit the inventory to one book. Don't worry about this limitation, however, because we'll extend the program soon.

Look at the program in Listing 14.1 and its output. Then read the explanation of the main points.

Listing 14.1 **The `book.c` Program**

```
//* book.c -- one-book inventory */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL  41      /* maximum length of title + 1         */
#define MAXAUTL  31      /* maximum length of author's name + 1 */

struct book {            /* structure template: tag is book     */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};                       /* end of structure template           */

int main(void)
{
    struct book library; /* declare library as a book variable  */

    printf("Please enter the book title.\n");
    s_gets(library.title, MAXTITL); /* access to the title portion        */
    printf("Now enter the author.\n");
    s_gets(library.author, MAXAUTL);
    printf("Now enter the value.\n");
    scanf("%f", &library.value);
    printf("%s by %s: $%.2f\n",library.title,
           library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
           library.title, library.value);
    printf("Done.\n");
```

```
    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

As in earlier chapters, we use `s_gets()` to strip the newline character that `fgets()` usually stores in a string. Here is a sample run:

```
Please enter the book title.
Chicken of the Andes
Now enter the author.
Disma Lapoult
Now enter the value.
29.99
Chicken of the Andes by Disma Lapoult: $29.99
Disma Lapoult: "Chicken of the Andes" ($29.99)
Done.
```

The structure created in Listing 14.1 has three parts (called members or fields)—one to store the title, one to store the author, and one to store the value. These are the three main skills you must acquire:

- Setting up a format or layout for a structure
- Declaring a variable to fit that layout
- Gaining access to the individual components of a structure variable

Setting Up the Structure Declaration

A *structure declaration* is the master plan that describes how a structure is put together. The declaration looks like this:

```
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

This declaration describes a structure made up of two character arrays and one `float` variable. It does not create an actual data object, but it describes what constitutes such an object. (Occasionally, we'll refer to a structure declaration as a *template* because it outlines how data will be stored. If you've heard of templates in C++, that's a different, more ambitious use of the word.) Let's look at the details. First comes the keyword `struct`. It identifies what comes next as a structure. Next comes an optional *tag*—the word `book`—that is a shorthand label you can use to refer to this structure. Therefore, later we have this declaration:

```
struct book library;
```

It declares `library` to be a structure variable using the `book` structure design.

Next in the structure declaration, the list of structure members are enclosed in a pair of braces. Each member is described by its own declaration, complete with a terminating semicolon. For example, the `title` portion is a `char` array with `MAXTITL` elements. A member can be any C data type—and that includes other structures!

A semicolon after the closing brace ends the definition of the structure design. You can place this declaration outside any function (externally), as we have done, or inside a function definition. If the declaration is placed inside a function, its tag can be used only inside that function. If the declaration is external, it is available to all the functions following the declaration in the file. For example, in a second function, you could define

```
struct book dickens;
```

and that function would have a variable, `dickens`, that follows the form of the `book` design.

The tag name is optional, but you must use one when you set up structures as we did, with the structure design defined one place and the actual variables defined elsewhere. We will return to this point soon, after we look at defining structure variables.

Defining a Structure Variable

The word *structure* is used in two senses. One is the sense "structure plan," which is what we just discussed. The structure plan tells the compiler *how* to represent the data, but it doesn't make the computer *allocate* space for the data. The next step is to create a *structure variable*, the

second sense of the word. The line in the program that causes a structure variable to be created is this:

```
struct book library;
```

Seeing this instruction, the compiler creates the variable `library`. Using the `book` template, the compiler allots space for a `char` array of `MAXTITL` elements, for a `char` array of `MAXAUTL` elements, and for a `float` variable. This storage is lumped together under the single name `library` (see Figure 14.1). (The next section explains how to unlump it as needed.)

In declaring a structure variable, `struct book` plays the same role that `int` or `float` does in simpler declarations. For example, you could declare two variables of the `struct book` type or even a pointer to that kind of structure:

```
struct book doyle, panshin, * ptbook;
```

The structure variables `doyle` and `panshin` would each have the parts `title`, `author`, and `value`. The pointer `ptbook` could point to `doyle`, `panshin`, or any other `book` structure. In essence, the `book` structure declaration creates a new type called `struct book`.

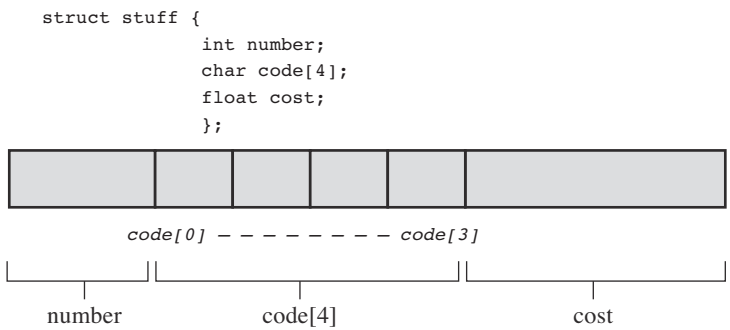

Figure 14.1 Memory allocation for a structure.

As far as the computer is concerned, the declaration

```
struct book library;
```

is short for

```
struct book {
   char title[MAXTITL];
   char author[AXAUTL];
   float value;
}  library;    /* follow declaration with variable name */
```

In other words, the process of declaring a structure and the process of defining a structure variable can be combined into one step. Combining the declaration and the variable definitions, as shown here, is the one circumstance in which a tag need not be used:

```
struct {         /* no tag */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
} library;
```

Use the tag form, however, if you plan to use a structure template more than once, or you can use the `typedef` alternative coming up later in this chapter.

There is one aspect of defining a structure variable that did not come up in this example: initialization. We'll look at that now.

Initializing a Structure

You've seen how to initialize variables and arrays:

```
int count = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

Can a structure variable be initialized, too? Yes, it can. To initialize a structure (any storage class for ANSI C and later, but excluding automatic variables for pre-ANSI C), you use a syntax similar to that used for arrays:

```
struct book library = {
    "The Pious Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
};
```

In short, you use a comma-separated list of initializers enclosed in braces. Each initializer should match the type of the structure member being initialized. Therefore, you can initialize the `title` member to a string and the `value` member to a number. To make the associations more obvious, we gave each member its own line of initialization, but all the compiler needs are commas to separate one member's initialization from the next.

> **Note Structure Initialization and Storage Class Duration**
>
> Chapter 12, "Storage Classes, Linkage, and Memory Management," mentioned that if you initialize a variable with static storage duration (such as static external linkage, static internal linkage, or static with no linkage), you have to use constant values. This applies to structures, too. If you are initializing a structure with static storage duration, the values in the initializer list must be constant expressions. If the storage duration is automatic, the values in the list need not be constants.

Gaining Access to Structure Members

A structure is like a "superarray," in which one element can be `char`, the next element `float`, and the next an `int` array. You can access the individual elements of an array by using a subscript. How do you access individual members of a structure? Use a dot (`.`), the structure member operator. For example, `library.value` is the `value` portion of `library`. You can use `library.value` exactly as you would use any other `float` variable. Similarly, you can use `library.title` exactly as you would use a `char` array. Therefore, the program uses expressions such as

```
s_gets(library.title, MAXTITL);
```

and

```
scanf("%f", &library.value);
```

In essence, `.title`, `.author`, and `.value` play the role of subscripts for a `book` structure.

Note that although `library` is a structure, `library.value` is a `float` type and is used like any other `float` type. For example, `scanf("%f",...)` requires the address of a `float` location, and that is what `&library.float` is. The dot has higher precedence than the `&` here, so the expression is the same as `&(library.float)`.

If you had a second structure variable of the same type, you would use the same method:

```
struct book bill, newt;

s_gets(bill.title, MAXTITL);
s_gets(newt.title, MAXTITL);
```

The `.title` refers to the first member of the `book` structure. Notice how the initial program prints the contents of the structure `library` in two different formats. This illustrates the freedom you have in using the members of a structure.

Initializers for Structures

C99 and C11 provide designated initializers for structures. The syntax is similar to that for designated initializers for arrays. However, designated initializers for structures use the dot operator and member names instead of brackets and indices to identify particular elements. For example, to initialize just the `value` member of a book structure, you would do this:

```
struct book surprise = { .value = 10.99};
```

You can use designated initializers in any order:

```
struct book gift = { .value = 25.99,
                     .author = "James Broadfool",
                     .title = "Rue for the Toad"};
```

Just as with arrays, a regular initializer following a designated initializer provides a value for the member following the designated member. Also, the last value supplied for a particular member is the value it gets. For example, consider this declaration:

```
struct book gift= { .value = 18.90,
                    .author = "Philionna Pestle",
                     0.25};
```

The value `0.25` is assigned to the `value` member because it is the one immediately listed after the `author` member in the structure declaration. The new value of `0.25` supersedes the value of `18.90` provided earlier. Now that you have these basics in hand, you're ready to expand your horizons and look at several ramifications of structures. You'll see arrays of structures, structures of structures, pointers to structures, and functions that process structures.

Arrays of Structures

Let's extend our book program to handle more books. Clearly, each book can be described by one structure variable of the `book` type. To describe two books, you need to use two such variables, and so on. To handle several books, you can use an array of such structures, and that is what we have created in the next program, shown in Listing 14.2. (If you're using Borland C/C++, see section "Borland C and Floating Point" later in the chapter.)

Structures and Memory

The `manybook.c` program uses an array of 100 structures. Because the array is an automatic storage class object, the information is typically placed on the stack. Such a large array requires a good-sized chunk of memory, which can cause problems. If you get a runtime error, perhaps complaining about the stack size or stack overflow, your compiler probably uses a default size for the stack that is too small for this example. To fix things, you can use the compiler options to set the stack size to 10,000 to accommodate the array of structures, or you can make the array static or external (so that it isn't placed in the stack), or you can reduce the array size to 16. Why didn't we just make the stack small to begin with? Because you should know about the potential stack size problem so that you can cope with it if you run into it on your own.

Listing 14.2 **The `manybook.c` Program**

```
/* manybook.c -- multiple book inventory */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL   40
#define MAXAUTL   40
#define MAXBKS   100              /* maximum number of books  */
```

```
struct book {                       /* set up book template     */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* array of book structures */
    int count = 0;
    int index;

    printf("Please enter the book title.\n");
    printf("Press [enter] at the start of a line to stop.\n");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
           && library[count].title[0] != '\0')
    {
        printf("Now enter the author.\n");
        s_gets(library[count].author, MAXAUTL);
        printf("Now enter the value.\n");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;          /* clear input line         */
        if (count < MAXBKS)
            printf("Enter the next title.\n");
    }

    if (count > 0)
    {
        printf("Here is the list of your books:\n");
        for (index = 0; index < count; index++)
            printf("%s by %s: $%.2f\n", library[index].title,
                   library[index].author, library[index].value);
    }
    else
        printf("No books? Too bad.\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
```

```
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

Borland C and Floating Point

Older Borland C compilers attempt to make programs more compact by using a small version of `scanf()` if the program doesn't use floating-point values. However, the compilers (through Borland C/C++ 3.1 for DOS, but not Borland C/C++ 4.0) are fooled if the only floating-point values are in an array of structures, as in the case for Listing 14.2. As a result, you get a message like this:

```
scanf : floating point formats not linked
Abnormal program termination
```

One workaround is adding this code to your program:

```
#include <math.h>
double dummy = sin(0.0);
```

This code forces the compiler to load the floating-point version of `scanf()`.

Here is a sample program run:

```
Please enter the book title.
Press [enter] at the start of a line to stop.
My Life as a Budgie
Now enter the author.
Mack Zackles
Now enter the value.
12.95
Enter the next title.
    ...more entries...
Here is the list of your books:
My Life as a Budgie by Mack Zackles: $12.95
Thought and Unthought Rethought by Kindra Schlagmeyer: $43.50
Concerto for Financial Instruments by Filmore Walletz: $49.99
The CEO Power Diet by Buster Downsize: $19.25
C++ Primer Plus by Stephen Prata: $59.99
Fact Avoidance: Perception as Reality by Polly Bull: $19.97
Coping with Coping by Dr. Rubin Thonkwacker: $0.02
```

```
Diaphanous Frivolity by Neda McFey: $29.99
Murder Wore a Bikini by Mickey Splats: $18.95
A History of Buvania, Volume 8, by Prince Nikoli Buvan: $50.04
Mastering Your Digital Watch, 5nd Edition, by Miklos Mysz: $28.95
A Foregone Confusion by Phalty Reasoner: $5.99
Outsourcing Government: Selection vs. Election by Ima Pundit: $33.33
```

First, we'll describe how to declare arrays of structures and how to access individual members. Then we will highlight two aspects of the program.

Declaring an Array of Structures

Declaring an array of structures is like declaring any other kind of array. Here's an example:

```
struct book library[MAXBKS];
```

This declares `library` to be an array with `MAXBKS` elements. Each element of this array is a structure of `book` type. Thus, `library[0]` is one `book` structure, `library[1]` is a second `book` structure, and so on. Figure 14.2 may help you visualize this. The name `library` itself is not a structure name; it is the name of the array whose elements are type `struct book` structures.

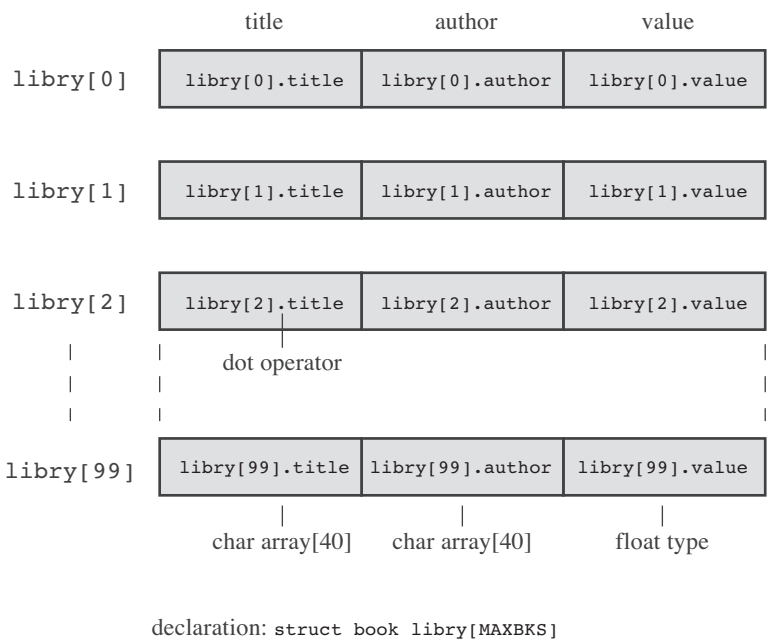

Figure 14.2 An array of structures.

Identifying Members of an Array of Structures

To identify members of an array of structures, you apply the same rule used for individual structures: Follow the structure name with the dot operator and then with the member name. Here's an example:

```
library[0].value   /* the value associated with the first array element */
library[4].title   /* the title associated with the fifth array element */
```

Note that the array subscript is attached to `library`, not to the end of the name:

```
library.value[2]    // WRONG
library[2].value    // RIGHT
```

The reason `library[2].value` is used is that `library[2]` is the structure variable name, just as `library[1]` is another structure variable name.

By the way, what do you suppose the following represents?

```
library[2].title[4]
```

It's the fifth character in the title (the `title[4]` part) of the book described by the third structure (the `library[2]` part). In the example, it would be the character *B*. This example points out that subscripts found to the right of the dot operator apply to individual members, but subscripts to the left of the dot operator apply to arrays of structures.

In summary, we have this sequence:

```
library             // an array of book structures
library[2]          // an array element, hence a book structure
library[2].title    // a char array (the title member of library[2])
library[2].title[4] // a char in the title member array
```

Let's finish the program now.

Program Discussion

The main change from the first program is that we inserted a loop to read multiple entries. The loop begins with this `while` condition:

```
while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
                      && library[count].title[0] != '\0')
```

The expression `s_gets(library[count].title, MAXTITL)` reads a string for the title of a book; the expression evaluates to `NULL` if `s_gets()` attempts to read past the end-of-file. The expression `library[count].title[0] != '\0'` tests whether the first character in the string is the null character (that is, if the string is empty). If the user presses the Enter key at the beginning of a line, the empty string is transmitted, and the loop ends. We also have a check to keep the number of books entered from exceeding the array's size limit.

Then the program has these lines:

```
while (getchar() != '\n')
    continue;              /* clear input line */
```

As you might recall from earlier chapters, this code compensates for the `scanf()` function ignoring spaces and newlines. When you respond to the request for the book's value, you type something like this:

```
12.50[enter]
```

This statement transmits the following sequence of characters:

```
12.50\n
```

The `scanf()` function collects the `1`, the `2`, the `.`, the `5`, and the `0`, but it leaves the `\n` sitting there, awaiting whatever read statement comes next. If the precautionary code were missing, the next read statement, `s_gets(library[count].title, MAXTITL)`, would read the left-over newline character as an empty line, and the program would think you had sent a stop signal. The code we inserted will eat up characters until it finds and disposes of the newline. It doesn't do anything with the characters except remove them from the input queue. This gives `s_gets()` a fresh start for the next input.

Now let's return to exploring structures.

Nested Structures

Sometimes it is convenient for one structure to contain, or *nest*, another. For example, Shalala Pirosky is building a structure of information about her friends. One member of the structure, naturally enough, is the friend's name. The name, however, can be represented by a structure itself, with separate entries for first and last name members. Listing 14.3 is a condensed example of Shalala's work.

Listing 14.3 **The `friend.c` Program**

```
// friend.c -- example of a nested structure
#include <stdio.h>
#define LEN 20
const char * msgs[5] =
{
    "    Thank you for the wonderful evening, ",
    "You certainly prove that a ",
    "is a special kind of guy. We must get together",
    "over a delicious ",
    " and have a few laughs"
};

struct names {                    // first structure
```

```
    char first[LEN];
    char last[LEN];
};

struct guy {                        // second structure
    struct names handle;            // nested structure
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow = {   // initialize a variable
        { "Ewen", "Villard" },
        "grilled salmon",
        "personality coach",
         68112.00
    };

    printf("Dear %s, \n\n", fellow.handle.first);
    printf("%s%s.\n", msgs[0], fellow.handle.first);
    printf("%s%s\n", msgs[1], fellow.job);
    printf("%s\n", msgs[2]);
    printf("%s%s%s", msgs[3], fellow.favfood, msgs[4]);
    if (fellow.income > 150000.0)
        puts("!!");
    else if (fellow.income > 75000.0)
        puts("!");
    else
        puts(".");
    printf("\n%40s%s\n", " ", "See you soon,");
    printf("%40s%s\n", " ", "Shalala");

    return 0;
}
```

Here is the output:

```
Dear Ewen,

    Thank you for the wonderful evening, Ewen.
You certainly prove that a personality coach
is a special kind of guy. We must get together
over a delicious grilled salmon and have a few laughs.

                                        See you soon,
                                        Shalala
```

First, note how the nested structure is set up in the structure declaration. It is simply declared, just as an `int` variable would be:

```
struct names handle;
```

This declaration says that `handle` is a variable of the `struct names` type. Of course, the file should also include the declaration for the `names` structure.

Second, note how you gain access to a member of a nested structure; you merely use the dot operator twice:

```
printf("Hello, %s!\n", fellow.handle.first);
```

The construction is interpreted this way, going from left to right:

```
(fellow.handle).first
```

That is, find `fellow`, then find the `handle` member of `fellow`, and then find the `first` member of that.

Pointers to Structures

Pointer lovers will be glad to know that you can have pointers to structures. There are at least four reasons why having pointers to structures is a good idea. First, just as pointers to arrays are easier to manipulate (in a sorting problem, say) than the arrays themselves, pointers to structures are often easier to manipulate than structures themselves. Second, in some older implementations, a structure can't be passed as an argument to a function, but a pointer to a structure can. Third, even if you can pass a structure as an argument, passing a pointer often is more efficient. Fourth, many wondrous data representations use structures containing pointers to other structures.

The next short example (see Listing 14.4) shows how to define a pointer to a structure and how to use it to access the members of a structure.

Listing 14.4 **The `friends.c` Program**

```
/* friends.c -- uses pointer to a structure */
#include <stdio.h>
#define LEN 20

struct names {
    char first[LEN];
    char last[LEN];
};
```

```
struct guy {
    struct names handle;
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow[2] = {
        {{ "Ewen", "Villard"},
         "grilled salmon",
         "personality coach",
         68112.00
        },
        {{"Rodney", "Swillbelly"},
         "tripe",
         "tabloid editor",
         232400.00
        }
    };
    struct guy * him;    /* here is a pointer to a structure */

    printf("address #1: %p #2: %p\n", &fellow[0], &fellow[1]);
    him = &fellow[0];    /* tell the pointer where to point  */
    printf("pointer #1: %p #2: %p\n", him, him + 1);
    printf("him->income is $%.2f: (*him).income is $%.2f\n",
         him->income, (*him).income);
    him++;               /* point to the next structure      */
    printf("him->favfood is %s:  him->handle.last is %s\n",
         him->favfood, him->handle.last);

    return 0;
}
```

The output, please:

```
address #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
pointer #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
him->income is $68112.00: (*him).income is $68112.00
him->favfood is tripe:  him->handle.last is Swillbelly
```

Let's look first at how we created a pointer to a `guy` structure. Then we'll explain how to specify individual structure members by using the pointer.

Declaring and Initializing a Structure Pointer

Declaration is as easy as can be:

```
struct guy * him;
```

First is the keyword `struct`, then the structure tag `guy`, and then an asterisk (`*`) followed by the pointer name. The syntax is the same as for the other pointer declarations you have seen.

This declaration does not create a new structure, but the pointer `him` can now be made to point to any existing structure of the `guy` type. For instance, if `barney` is a structure of the `guy` type, you could do this:

```
him = &barney;
```

Unlike the case for arrays, the name of a structure is not the address of the structure; you need to use the `&` operator.

In the example, `fellow` is an array of structures, which means that `fellow[0]` is a structure, so the code initializes `him` by making it point to `fellow[0]`:

```
him = &fellow[0];
```

The first two output lines show the success of this assignment. Comparing the two lines, you see that `him` points to `fellow[0]`, and `him + 1` points to `fellow[1]`. Note that adding `1` to `him` adds 84 to the address. In hexadecimal, 874 – 820 = 54 (hex) = 84 (base 10) because each `guy` structure occupies 84 bytes of memory: `names.first` is 20, `names.last` is 20, `favfood` is 20, `job` is 20, and `income` is 4, the size of `float` on our system. Incidentally, on some systems, the size of a structure may be greater than the sum of its parts. That's because a system's alignment requirements for data may cause gaps. For example, a system may have to place each member at an even address or at an address that is a multiple of four. Such structures might end up with unused "holes" in them.

Member Access by Pointer

The pointer `him` is pointing to the structure `fellow[0]`. How can you use `him` to get a value of a member of `fellow[0]`? The third output line shows two methods.

The first method, and the most common, uses a new operator, `->`. This operator is formed by typing a hyphen (`-`) followed by the greater-than symbol (`>`). We have these relationships:

```
him->income is barney.income if him == &barney
him->income is fellow[0].income if him == &fellow[0]
```

In other words, a structure pointer followed by the `->` operator works the same way as a structure name followed by the `.` (dot) operator. (You can't properly say `him.income` because `him` is not a structure name.)

It is important to note that `him` is a pointer, but `him->income` is a member of the pointed-to structure. So in this case, `him->income` is a `float` variable.

The second method for specifying the value of a structure member follows from this sequence: If `him == &fellow[0]`, then `*him == fellow[0]` because `&` and `*` are reciprocal operators. Hence, by substitution, you have the following:

```
fellow[0].income == (*him).income
```

The parentheses are required because the `.` operator has higher precedence than `*`.

In summary, if `him` is a pointer to a type `guy` structure named `barney`, the following are all equivalent:

```
barney.income == (*him).income == him->income   // assuming him == &barney
```

Now let's look at the interaction between structures and functions.

Telling Functions About Structures

Recall that function arguments pass values to the function. Each value is a number—perhaps `int`, perhaps `float`, perhaps ASCII character code, or perhaps an address. A structure is a bit more complicated than a single value, so it is not surprising that ancient C implementations do not allow a structure to be used as an argument for a function. This limitation was removed in newer implementations, and ANSI C allows structures to be used as arguments. Therefore, modern implementations give you a choice between passing structures as arguments and passing pointers to structures as arguments—or if you are concerned with just part of a structure, you can pass structure members as arguments. We'll examine all three methods, beginning with passing structure members as arguments.

Passing Structure Members

As long as a structure member is a data type with a single value (that is, an `int` or one of its relatives, a `char`, a `float`, a `double`, or a pointer), it can be passed as a function argument to a function that accepts that particular type. The fledgling financial analysis program in Listing 14.5, which adds the client's bank account to his or her savings and loan account, illustrates this point.

Listing 14.5 **The `funds1.c` Program**

```
/* funds1.c -- passing structure members as arguments */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
    double savefund;
};
```

```
double sum(double, double);

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n",
           sum(stan.bankfund, stan.savefund) );
    return 0;
}

/* adds two double numbers */
double sum(double x, double y)
{
    return(x + y);
}
```

Here is the result of running this program:

```
Stan has a total of $12576.21.
```

Ah, it works. Notice that the function `sum()` neither knows nor cares whether the actual arguments are members of a structure; it requires only that they be type `double`.

Of course, if you want a called function to affect the value of a member in the calling function, you can transmit the address of the member:

```
modify(&stan.bankfund);
```

This would be a function that alters Stan's bank account.

The next approach to telling a function about a structure involves letting the called function know that it is dealing with a structure.

Using the Structure Address

We will solve the same problem as before, but this time we will use the address of the structure as an argument. Because the function has to work with the `funds` structure, it, too, has to make use of the `funds` declaration. See Listing 14.6 for the program.

Listing 14.6 **The `funds2.c` Program**

```
/* funds2.c -- passing a pointer to a structure */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
    double savefund;
};

double sum(const struct funds *);  /* argument is a pointer */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(&stan));

    return 0;
}

double sum(const struct funds * money)
{
    return(money->bankfund + money->savefund);
}
```

This, too, produces the following output:

```
Stan has a total of $12576.21.
```

The `sum()` function uses a pointer (`money`) to a `funds` structure for its single argument. Passing the address `&stan` to the function causes the pointer `money` to point to the structure `stan`. Then the `->` operator is used to gain the values of `stan.bankfund` and `stan.savefund`. Because the function does not alter the contents of the pointed-to value, it declares `money` as a pointer-to-`const`.

This function also has access to the institution names, although it doesn't use them. Note that you must use the `&` operator to get the structure's address. Unlike the array name, the structure name alone is not a synonym for its address.

Passing a Structure as an Argument

For compilers that permit passing structures as arguments, the last example can be rewritten as shown in Listing 14.7.

Listing 14.7 **The `funds3.c` Program**

```
/* funds3.c -- passing a structure */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
    double savefund;
};

double sum(struct funds moolah);  /* argument is a structure */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(stan));

    return 0;
}

double sum(struct funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}
```

Again, the output is this:

```
Stan has a total of $12576.21.
```

We replaced `money`, which was a pointer to `struct funds`, with `moolah`, which is a `struct funds` variable. When `sum()` is called, an automatic variable called `moolah` is created according to the `funds` template. The members of this structure are then initialized to be copies of the values held in the corresponding members of the structure `stan`. Therefore, the computations

are done by using a copy of the original structure; whereas, the preceding program (the one using a pointer) used the original structure. Because `moolah` is a structure, the program uses `moolah.bankfund`, not `moolah->bankfund`. On the other hand, Listing 14.6 used `money->bankfund` because `money` is a pointer, not a structure.

More on Structure Features

Modern C allows you to assign one structure to another, something you can't do with arrays. That is, if `n_data` and `o_data` are both structures of the same type, you can do the following:

```
o_data = n_data;    // assigning one structure to another
```

This causes each member of `n_data` to be assigned the value of the corresponding member of `o_data`. This works even if a member happens to be an array. Also, you can initialize one structure to another of the same type:

```
struct names right_field = {"Ruthie", "George"};
struct names captain = right_field;  // initialize a structure to another
```

Under modern C, including ANSI C, not only can structures be passed as function arguments, they can be returned as function return values. Using structures as function arguments enables you to convey structure information to a function; using functions to return structures enables you to convey structure information from a called function to the calling function. Structure pointers also allow two-way communication, so you can often use either approach to solve programming problems. Let's look at another set of examples illustrating these two approaches.

To contrast the two approaches, we'll write a simple program that handles structures by using pointers; then we'll rewrite it by using structure passing and structure returns. The program itself asks for your first and last names and reports the total number of letters in them. This project hardly requires structures, but it offers a simple framework for seeing how they work. Listing 14.8 presents the pointer form.

Listing 14.8 **The `names1.c` Program**

```
/* names1.c -- uses pointers to a structure */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

void getinfo(struct namect *);
void makeinfo(struct namect *);
```

```
void showinfo(const struct namect *);
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    return 0;
}

void getinfo (struct namect * pst)
{
    printf("Please enter your first name.\n");
    s_gets(pst->fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(pst->lname, NLEN);
}

void makeinfo (struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
    strlen(pst->lname);
}

void showinfo (const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
           pst->fname, pst->lname, pst->letters);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
```

```
    }
    return ret_val;
}
```

Compiling and running the program produces results like the following:

```
Please enter your first name.
Viola
Please enter your last name.
Plunderfest
Viola Plunderfest, your name contains 16 letters.
```

The work of the program is allocated to three functions called from `main()`. In each case, the address of the `person` structure is passed to the function.

The `getinfo()` function transfers information from itself to `main()`. In particular, it gets names from the user and places them in the `person` structure, using the `pst` pointer to locate it. Recall that `pst->lname` means the `lname` member of the structure pointed to by `pst`. This makes `pst->lname` equivalent to the name of a `char` array, hence a suitable argument for `gets()`. Note that although `getinfo()` feeds information to the main program, it does not use the return mechanism, so it is type `void`.

The `makeinfo()` function performs a two-way transfer of information. By using a pointer to `person`, it locates the two names stored in the structure. It uses the C library function `strlen()` to calculate the total number of letters in each name and then uses the address of `person` to stow away the sum. Again, the type is `void`. Finally, the `showinfo()` function uses a pointer to locate the information to be printed. Because this function does not alter the contents of an array, it declares the pointer as `const`.

In all these operations, there has been but one structure variable, `person`, and each of the functions have used the structure address to access it. One function transferred information from itself to the calling program, one transferred information from the calling program to itself, and one did both.

Now let's see how you can program the same task using structure arguments and return values. First, to pass the structure itself, use the argument `person` rather than `&person`. The corresponding formal argument, then, is declared type `struct namect` instead of being a pointer to that type. Second, to provide structure values to `main()`, you can return a structure. Listing 14.9 presents the nonpointer version.

Listing 14.9 **The `names2.c` Program**

```
/* names2.c -- passes and returns structures */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
```

```
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

struct namect getinfo(void);
struct namect makeinfo(struct namect);
void showinfo(struct namect);
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    person = getinfo();
    person = makeinfo(person);
    showinfo(person);

    return 0;
}

struct namect getinfo(void)
{
    struct namect temp;
    printf("Please enter your first name.\n");
    s_gets(temp.fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(temp.lname, NLEN);

    return temp;
}

struct namect makeinfo(struct namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);

    return info;
}

void showinfo(struct namect info)
{
    printf("%s %s, your name contains %d letters.\n",
           info.fname, info.lname, info.letters);
}

char * s_gets(char * st, int n)
{
```

```
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

This version produces the same final result as the preceding one, but it proceeds in a different manner. Each of the three functions creates its own copy of `person`, so this program uses four distinct structures instead of just one.

Consider the `makeinfo()` function, for example. In the first program, the address of `person` was passed, and the function fiddled with the actual `person` values. In this second version, a new structure called `info` is created. The values stored in `person` are copied to `info`, and the function works with the copy. Therefore, when the number of letters is calculated, it is stored in `info`, but not in `person`. The return mechanism, however, fixes that. The `makeinfo()` line

```
return info;
```

combines with the `main()` line

```
person = makeinfo(person);
```

to copy the values stored in `info` into `person`. Note that the `makeinfo()` function had to be declared type `struct namect` because it returns a structure.

Structures or Pointer to Structures?

Suppose you have to write a structure-related function. Should you use structure pointers as arguments, or should you use structure arguments and return values? Each approach has its strengths and weaknesses.

The two advantages of the pointer argument method are that it works on older as well as newer C implementations and that it is quick; you just pass a single address. The disadvantage is that you have less protection for your data. Some operations in the called function could inadvertently affect data in the original structure. However, the ANSI C addition of the `const` qualifier solves that problem. For example, if you put code into the `showinfo()` function of Listing 11.8 that changes any member of the structure, the compiler will catch it as an error.

One advantage of passing structures as arguments is that the function works with copies of the original data, which is safer than working with the original data. Also, the programming style tends to be clearer. Suppose you define the following structure type:

```
struct vector {double x; double y;};
```

You want to set the vector `ans` to the sum of the vectors `a` and `b`. You could write a structure-passing and returning function that would make the program look like this:

```
struct vector ans, a, b;
struct vector sum_vect(struct vector, struct vector);
...
ans = sum_vect(a,b);
```

The preceding version is more natural looking to an engineer than a pointer version, which might look like this:

```
struct vector ans, a, b;
void sum_vect(const struct vector *, const struct vector *, struct vector *);
...
sum_vect(&a, &b, &ans);
```

Also, in the pointer version, the user has to remember whether the address for the sum should be the first or the last argument.

The two main disadvantages to passing structures are that older implementations might not handle the code and that it wastes time and space. It's especially wasteful to pass large structures to a function that uses only one or two members of the structure. In that case, passing a pointer or passing just the required members as individual arguments makes more sense.

Typically, programmers use structure pointers as function arguments for reasons of efficiency, using `const` when needed to protect data from unintended changes. Passing structures by value is most often done for structures that are small to begin with.

Character Arrays or Character Pointers in a Structure

The examples so far have used character arrays to store strings in a structure. You might have wondered if you can use pointers-to-`char` instead. For example, Listing 14.3 had this declaration:

```
#define LEN 20
struct names {
    char first[LEN];
    char last[LEN];
};
```

Can you do this instead?

```
struct pnames {
    char * first;
```

```
    char * last;
};
```

The answer is that you can, but you might get into trouble unless you understand the implications. Consider the following code:

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s\n", veep.first, treas.first);
```

This is valid code, and it works, but consider where the strings are stored. For the `struct names` variable `veep`, the strings are stored inside the structure; the structure has allocated a total of 40 bytes to hold the two names. For the `struct pnames` variable `treas`, however, the strings are stored wherever the compiler stores string constants. All the structure holds are the two addresses, which takes a total of 16 bytes on our system. In particular, the `struct pnames` structure allocates no space to store strings. It can be used only with strings that have had space allocated for them elsewhere, such as string constants or strings in arrays. In short, the pointers in a `pnames` structure should be used only to manage strings that were created and allocated elsewhere in the program.

Let's see where this restriction is a problem. Consider the following code:

```
struct names accountant;
struct pnames attorney;
puts("Enter the last name of your accountant:");
scanf("%s", accountant.last);
puts("Enter the last name of your attorney:");
scanf("%s", attorney.last);   /* here lies the danger */
```

As far as syntax goes, this code is fine. But where does the input get stored? For the accountant, the name is stored in the last member of the `accountant` variable; this structure has an array to hold the string. For the attorney, `scanf()` is told to place the string at the address given by `attorney.last`. Because this is an uninitialized variable, the address could have any value, and the program could try to put the name anywhere. If you are lucky, the program might work, at least some of the time—or an attempt could bring your program to a crashing halt. Actually, if the program works, you're unlucky, because the program will have a dangerous programming error of which you are unaware.

So if you want a structure to store the strings, it's simpler to use character array members. Storing pointers-to-`char` has its uses, but it also has the potential for serious misuse.

Structure, Pointers, and `malloc()`

One instance in which it does make sense to use a pointer in a structure to handle a string is if you use `malloc()` to allocate memory and use a pointer to store the address. This approach has the advantage that you can ask `malloc()` to allocate just the amount of space that's needed for a string. You can ask for 4 bytes to store `"Joe"` and 18 bytes for the Madagascan name

"Rasolofomasoandro". It doesn't take much to adapt Listing 14.9 to this approach. The two main changes are changing the structure definition to use pointers instead of arrays and then providing a new version of the getinfo() function.

The new structure definition will look like this:

```
struct namect {
    char * fname;  // using pointers instead of arrays
    char * lname;
    int letters;
};
```

The new version of getinfo() will read the input into a temporary array, use malloc() to allocate storage space, and copy the string to the newly allocated space. It will do so for each name:

```
void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // allocate memory to hold name
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // copy name to allocated memory
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

Make sure you understand that the two strings are not stored in the structure. They are stored in the chunk of memory managed by malloc(). However, the addresses of the two strings are stored in the structure, and the addresses are what string-handling functions typically work with. Therefore, the remaining functions in the program need not be changed at all.

However, as Chapter 12 suggests, you should balance calls to malloc() with calls to free(), so the program adds a new function called cleanup() to free the memory once the program is done using it. You'll find this new function and the rest of the program in Listing 14.10.

Listing 14.10 **The names3.c Program**

```
// names3.c -- use pointers and malloc()
#include <stdio.h>
#include <string.h>   // for strcpy(), strlen()
#include <stdlib.h>   // for malloc(), free()
#define SLEN 81
struct namect {
```

```
    char * fname;  // using pointers
    char * lname;
    int letters;
};

void getinfo(struct namect *);        // allocates memory
void makeinfo(struct namect *);
void showinfo(const struct namect *);
void cleanup(struct namect *);        // free memory when done
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    cleanup(&person);

    return 0;
}

void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // allocate memory to hold name
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // copy name to allocated memory
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}

void makeinfo (struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
    strlen(pst->lname);
}

void showinfo (const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
```

```
           pst->fname, pst->lname, pst->letters);
}

void cleanup(struct namect * pst)
{
    free(pst->fname);
    free(pst->lname);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

Here is some sample output:

```
Please enter your first name.
Floresiensis
Please enter your last name.
Mann
Floresiensis Mann, your name contains 16 letters.
```

Compound Literals and Structures (C99)

C99's compound literal feature is available for structures as well as for arrays. It's handy if you just need a temporary structure value. For instance, you can use compound literals to create a structure to be used as a function argument or to be assigned to another structure. The syntax is to preface a brace-enclosed initializer list with the type name in parentheses. For example, the following is a compound literal of the `struct book` type:

```
(struct book) {"The Idiot", "Fyodor Dostoyevsky", 6.99}
```

Listing 14.11 shows an example using compound literals to provide two alternative values for a structure variable. (At the time of writing, several, but not all, compilers support this feature, but time should remedy this problem.)

Listing 14.11 **The `complit.c` Program**

```
/* complit.c -- compound literals */
#include <stdio.h>
#define MAXTITL  41
#define MAXAUTL  31

struct book {          // structure template: tag is book
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book readfirst;
    int score;

    printf("Enter test score: ");
    scanf("%d",&score);

    if(score >= 84)
        readfirst = (struct book) {"Crime and Punishment",
                                   "Fyodor Dostoyevsky",
                                   11.25};
    else
          readfirst = (struct book) {"Mr. Bouncy's Nice Hat",
                                   "Fred Winsome",
                                    5.99};
    printf("Your assigned reading:\n");
    printf("%s by %s: $%.2f\n",readfirst.title,
          readfirst.author, readfirst.value);

    return 0;
}
```

You also can use compound literals as arguments to functions. If the function expects a structure, you can pass the compound literal as the actual argument:

```
struct rect {double x; double y;};
double rect_area(struct rect r){return r.x * r.y;}
...
```

```
double area;
area = rect_area( (struct rect) {10.5, 20.0});
```

This causes `area` to be assigned the value `210.0`.

If a function expects an address, you can pass the address of a compound literal:

```
struct rect {double x; double y;};
double rect_areap(struct rect * rp){return rp->x * rp->y;}
...
double area;
area = rect_areap( &(struct rect) {10.5, 20.0});
```

This causes `area` to be assigned the value `210.0`.

Compound literals occurring outside of any function have static storage duration, and those occurring inside a block have automatic storage duration. The same syntax rules hold for compound literals as hold for regular initializer lists. This means, for example, that you can use designated initializers in a compound literal.

Flexible Array Members (C99)

C99 has a feature called the *flexible array member*. It lets you declare a structure for which the last member is an array with special properties. One special property is that the array doesn't exist—at least, not immediately. The second special property is that, with the right code, you can use the flexible array member as if it did exist and has whatever number of elements you need. This probably sounds a little peculiar, so let's go through the steps of creating and using a structure with a flexible array member.

First, here are the rules for declaring a flexible array member:

- The flexible array member must be the last member of the structure.
- There must be at least one other member.
- The flexible array is declared like an ordinary array, except that the brackets are empty.

Here's an example illustrating these rules:

```
struct flex
{
    int count;
    double average;
    double scores[];   // flexible array member
};
```

If you declare a variable of type `struct flex`, you can't use `scores` for anything, because no memory space is set-aside for it. In fact, it's not intended that you ever declare variables of the `struct flex` type. Instead, you are supposed to declare a *pointer* to the `struct flex` type and

then use `malloc()` to allocate enough space for the ordinary contents of `struct flex` *plus* any extra space you want for the flexible array member. For example, suppose you want `scores` to represent an array of five `double` values. Then you would do this:

```
struct flex * pf;  // declare a pointer
// ask for space for a structure and an array
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

Now you have a chunk of memory large enough to store `count`, `average`, and an array of five `double` values. You can use the pointer `pf` to access these members:

```
pf->count = 5;          // set count member
pf->scores[2] = 18.5;   // access an element of the array member
```

Listing 14.12 carries this example a little further, letting the flexible array member represent five values in one case and nine values in a second case. It also illustrates writing a function for processing a structure with a flexible array element.

Listing 14.12 **The `flexmemb.c` Program**

```
// flexmemb.c -- flexible array member (C99 feature)
#include <stdio.h>
#include <stdlib.h>

struct flex
{
    size_t count;
    double average;
    double scores[];   // flexible array member
};

void showFlex(const struct flex * p);

int main(void)
{
    struct flex * pf1, *pf2;
    int n = 5;
    int i;
    int tot = 0;

    // allocate space for structure plus array
    pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf1->count = n;
    for (i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
```

```
    pf1->average = tot / n;
    showFlex(pf1);

    n = 9;
    tot = 0;
    pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf2->count = n;
    for (i = 0; i < n; i++)
    {
        pf2->scores[i] = 20.0 - i/2.0;
        tot += pf2->scores[i];
    }
    pf2->average = tot / n;
    showFlex(pf2);
    free(pf1);
    free(pf2);

    return 0;
}

void showFlex(const struct flex * p)
{
    int i;
    printf("Scores : ");
    for (i = 0; i < p->count; i++)
        printf("%g ", p->scores[i]);
    printf("\nAverage: %g\n", p->average);
}
```

Here is the output:

```
Scores : 20 19 18 17 16
Average: 18
Scores : 20 19.5 19 18.5 18 17.5 17 16.5 16
Average: 17
```

Structures with flexible array members do have some special handling requirements. First, don't use structure assignment for copying:

```
    struct flex * pf1, *pf2;  // *pf1 and *pf2 are structures
...
    *pf2 = *pf1;   // don't do this
```

This would just copy the nonflexible members of the structure. Instead, use the memcpy() function described in Chapter 16, "The C Preprocessor and the C Library."

Second, don't use this sort of structure with functions that pass structures by value. The reason is the same; passing an argument by value is like assignment. Instead, use functions that pass the address of the structure.

Third, don't use a structure with a flexible array member as an element of an array or a member of another structure.

Some of you may have heard of something similar to the flexible array member called the *struct hack*. Instead of using empty brackets to declare the flexible member, the struct hack specifies a 0 array size. However, the struct hack is something that worked for a particular compiler (GCC); it wasn't standard C. The flexible member approach provides a standard-sanctioned version of the technique.

Anonymous Structures (C11)

An anonymous structure is a structure member that is an unnamed structure. To see how this works, first consider the following setup for a nested structure:

```
struct names
{
    char first[20];
    char last[20];
};
struct person
{
    int id;
    struct names name;  // nested structure member
};
struct person ted = {8483, {"Ted", "Grass"}};
```

Here the `name` member is a nested structure, and you could use an expression like `ted.name.first` to access `"Ted"`:

```
puts(ted.name.first);
```

With C11, you can define `person` using a nested unnamed member structure:

```
struct person
{
    int id;
    struct {char first[20]; char last[20];};  // anonymous structure
};
```

You could initialize this structure in the same fashion:

```
struct person ted = {8483, {"Ted", "Grass"}};
```

But access is simplified as you use member names such as `first` as if they were `person` members:

```
puts(ted.first);
```

Of course, you could simply have made `first` and `last` direct members of `person` and eliminated nested structures. The anonymous feature becomes more useful with nested unions, which we will discuss later in this chapter.

Functions Using an Array of Structures

Suppose you have an array of structures that you want to process with a function. The name of an array is a synonym for its address, so it can be passed to a function. Again, the function needs access to the structure template. To show how this works, Listing 14.13 expands our monetary program to two people so that it has an array of two `funds` structures.

Listing 14.13 **The `funds4.c` Program**

```
/* funds4.c -- passing an array of structures to a function */
#include <stdio.h>
#define FUNDLEN 50
#define N 2

struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
    double savefund;
};

double sum(const struct funds money[], int n);

int main(void)
{
    struct funds jones[N] = {
        {
            "Garlic-Melon Bank",
            4032.27,
            "Lucky's Savings and Loan",
            8543.94

        },
        {
            "Honest Jack's Bank",
            3620.88,
            "Party Time Savings",
            3802.91
        }
    };

    printf("The Joneses have a total of $%.2f.\n",
```

```
            sum(jones,N));

    return 0;
}

double sum(const struct funds money[], int n)
{
    double total;
    int i;

    for (i = 0, total = 0; i < n; i++)
        total += money[i].bankfund + money[i].savefund;

    return(total);
}
```

The output is this:

```
The Joneses have a total of $20000.00.
```

(What an even sum! One would almost think the figures were contrived.)

The array name `jones` is the address of the array. In particular, it is the address of the first element of the array, which is the structure `jones[0]`. Therefore, initially the pointer `money` is given by this expression:

```
money = &jones[0];
```

Because money points to the first element of the `jones` array, `money[0]` is another name for the first element of that array. Similarly, `money[1]` is the second element. Each element is a funds structure, so each can use the dot (`.`) operator to access the structure members.

These are the main points:

- You can use the array name to pass the address of the first structure in the array to a function.
- You can then use array bracket notation to access the successive structures in the array. Note that the function call

  ```
  sum(&jones[0], N)
  ```

  would have the same effect as using the array name because both `jones` and `&jones[0]` are the same address. Using the array name is just an indirect way of passing the structure address.
- Because the `sum()` function ought not alter the original data, the function uses the ANSI C `const` qualifier.

Saving the Structure Contents in a File

Because structures can hold a wide variety of information, they are important tools for constructing databases. For example, you could use a structure to hold all the pertinent information about an employee or an auto part. Ultimately, you would want to be able to save this information in, and retrieve it from, a file. A database file could contain an arbitrary number of such data objects. The entire set of information held in a structure is termed a *record*, and the individual items are *fields*. Let's investigate these topics.

What is perhaps the most obvious way to save a record is the least efficient way, and that is to use `fprintf()`. For example, recall the `book` structure introduced in Listing 14.1:

```
#define MAXTITL   40
#define MAXAUTL   40
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

If `pbooks` identified a file stream, you could save the information in a `struct book` variable called `primer` with the following statement:

```
fprintf(pbooks, "%s %s %.2f\n", primer.title,
        primer.author, primer.value);
```

This setup becomes unwieldy for structures with, say, 30 members. Also, it poses a retrieval problem because the program would need some way of telling where one field ends and another begins. This problem can be fixed by using a format with fixed-size fields (for example, `"%39s%39s%8.2f"`), but the awkwardness remains.

A better solution is to use `fread()` and `fwrite()` to read and write structure-sized units. Recall that these functions read and write using the same binary representation that the program uses. For example,

```
fwrite(&primer, sizeof (struct book), 1, pbooks);
```

goes to the beginning address of the `primer` structure and copies all the bytes of the structure to the file associated with `pbooks`. The `sizeof (struct book)` term tells the function how large a block to copy, and the `1` indicates that it should copy just one block. The `fread()` function with the same arguments copies a structure-sized chunk of data from the file to the location pointed to by `&primer`. In short, these functions read and write one whole record at a time instead of a field at a time.

One drawback to saving data in binary representation is that different systems might use different binary representations, so the data file might not be portable. Even on the same system, different compiler settings could result in different binary layouts.

A Structure-Saving Example

To show how these functions can be used in a program, we've modified the program in Listing 14.2 so that the book titles are saved in a file called `book.dat`. If the file already exists, the program shows you its current contents and then enables you to add to the file. Listing 14.14 presents the new version. (If you're using an older Borland compiler, review the "Borland C and Floating Point" discussion in the sidebar near Listing 14.2.)

Listing 14.14 **The `booksave.c` Program**

```
/* booksave.c -- saves structure contents in a file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXTITL  40
#define MAXAUTL  40
#define MAXBKS   10             /* maximum number of books */
char * s_gets(char * st, int n);
struct book {                   /* set up book template    */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* array of structures     */
    int count = 0;
    int index, filecount;
    FILE * pbooks;
    int size = sizeof (struct book);

    if ((pbooks = fopen("book.dat", "a+b")) == NULL)
    {
        fputs("Can't open book.dat file\n",stderr);
        exit(1);
    }

    rewind(pbooks);            /* go to start of file     */
    while (count < MAXBKS &&  fread(&library[count], size,
                                    1, pbooks) == 1)
    {
        if (count == 0)
            puts("Current contents of book.dat:");
        printf("%s by %s: $%.2f\n",library[count].title,
               library[count].author, library[count].value);
        count++;
```

```
    }
    filecount = count;
    if (count == MAXBKS)
    {
        fputs("The book.dat file is full.", stderr);
        exit(2);
    }

    puts("Please add new book titles.");
    puts("Press [enter] at the start of a line to stop.");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
           && library[count].title[0] != '\0')
    {
        puts("Now enter the author.");
        s_gets(library[count].author, MAXAUTL);
        puts("Now enter the value.");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;               /* clear input line  */
        if (count < MAXBKS)
            puts("Enter the next title.");
    }

    if (count > 0)
    {
        puts("Here is the list of your books:");
        for (index = 0; index < count; index++)
            printf("%s by %s: $%.2f\n",library[index].title,
                   library[index].author, library[index].value);
        fwrite(&library[filecount], size, count - filecount,
               pbooks);
    }
    else
        puts("No books? Too bad.\n");

    puts("Bye.\n");
    fclose(pbooks);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
```

```
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

We'll look at a couple of sample runs and then discuss the main programming points.

```
$ booksave
Please add new book titles.
Press [enter] at the start of a line to stop.
Metric Merriment
Now enter the author.
Polly Poetica
Now enter the value.
18.99
Enter the next title.
Deadly Farce
Now enter the author.
Dudley Forse
Now enter the value.
15.99
Enter the next title.
[enter]
Here is the list of your books:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Bye.
$ booksave
Current contents of book.dat:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Please add new book titles.
The Third Jar
Now enter the author.
Nellie Nostrum
Now enter the value.
22.99
Enter the next title.
[enter]
Here is the list of your books:
```

```
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
The Third Jar by Nellie Nostrum: $22.99
Bye.
$
```

Running the `booksave.c` program again would show all three books as current file records.

Program Points

First, the `"a+b"` mode is used for opening the file. The `a+` part lets the program read the whole file and append data to the end of the file. The `b` is the ANSI way of signifying that the program will use the binary file format. For Unix systems that don't accept the `b`, you can omit it because Unix has only one file form anyway. For other pre-ANSI implementations, you might need to find the local equivalent to using `b`.

We chose the binary mode because `fread()` and `fwrite()` are intended for binary files. True, some of the structure contents are text, but the `value` member is not. If you use a text editor to look at `book.dat`, the text part will show up okay, but the numeric part will be unreadable and could even cause your text editor to barf.

The `rewind()` command ensures that the file position pointer is situated at the start of the file, ready for the first read.

The initial `while` loop reads one structure at a time into the array of structures, stopping when the array is full or when the file is exhausted. The variable `filecount` keeps track of how many structures were read.

The next `while` loop prompts for, and takes, user input. As in Listing 14.2, this loop quits when the array is full or when the user presses the Enter key at the beginning of a line. Notice that the `count` variable starts with the value it had after the preceding loop. This causes the new entries to be added to the end of the array.

The `for` loop then prints the data both from the file and from the user. Because the file was opened in the append mode, new writes to the file are appended to the existing contents.

We could have used a loop to add one structure at a time to the end of the file. However, we decided to use the ability of `fwrite()` to write more than one block at a time. The expression `count - filecount` yields the number of new book titles to be added, and the call to `fwrite()` writes that number of structure-sized blocks to the file. The expression `&library[filecount]` is the address of the first new structure in the array, so copying begins from that point.

This example is, perhaps, the simplest way to write structures to a file and to retrieve them, but it can waste space because the unused parts of a structure are saved, too. The size of this structure is `2 x 40 x sizeof (char) + sizeof (float)`, which totals 84 bytes on our system. None of the entries actually need all that space. However, each data chunk being the same size makes retrieving the data easy.

Another approach is to use variably sized records. To facilitate reading such records from a file, each record can begin with a numerical field specifying the record size. This is a bit more complex than what we have done. Normally, this method involves "linked structures," which we describe next, and dynamic memory allocation, which we discuss in Chapter 16.

Structures: What Next?

Before ending our exploration of structures, we would like to mention one of the more important uses of structures: creating new data forms. Computer users have developed data forms much more efficiently for certain problems than the arrays and simple structures we have presented. These forms have names such as queues, binary trees, heaps, hash tables, and graphs. Many such forms are built from linked structures. Typically, each structure contains one or two items of data plus one or two pointers to other structures of the same type. Those pointers link one structure to another and furnish a path to enable you to search through the overall assemblage of structures. For example, Figure 14.3 shows a binary tree structure, with each individual structure (or node) connected to the two below it.

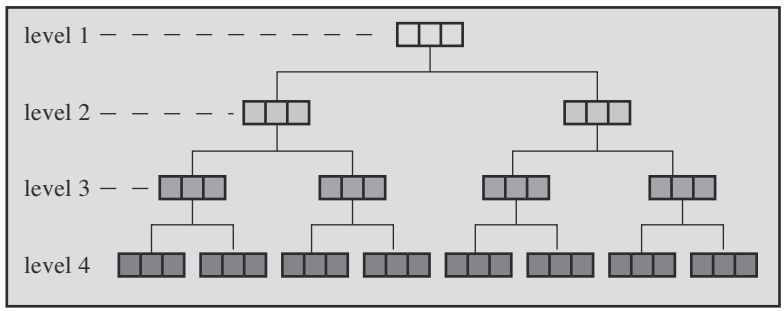

Figure 14.3 A binary tree structure.

Is the hierarchical, or *tree*, structure shown in Figure 14.3 more efficient than an array? Consider the case of a tree with 10 levels of nodes. It has 2^{10}–1, or 1,023, nodes in which you could store up to 1,023 words. If the words were arranged according to some sensible plan, you could start at the top level and find any word in at most nine moves as your search moves down one level to the next. If you have the words in an array, you might have to search all 1,023 elements before finding the word you seek.

If you are interested in more advanced concepts such as this, you can consult any number of computer science texts on data structures. With the C structures, you can create and use virtually every form presented in these texts. Also, Chapter 17, "Advanced Data Representation," investigates some of these advanced forms.

That's our final word on structures for this chapter, but we will present examples of linked structures in Chapter 17. Next, we'll look at three other C features for dealing with data: unions, enumerations, and `typedef`.

Unions: A Quick Look

A *union* is a type that enables you to store different data types in the same memory space (but not simultaneously). A typical use is a table designed to hold a mixture of types in some order that is neither regular nor known in advance. By using an array of unions, you can create an array of equal-sized units, each of which can hold a variety of data types.

Unions are set up in much the same way as structures. There is a union template and a union variable. They can be defined in one step or, by using a union tag, in two. Here is an example of a union template with a tag:

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
```

A structure with a similar declaration would be able to hold an `int` value *and* a `double` value *and* a `char` value. This union, however, can hold an `int` value *or* a `double` value *or* a `char` value.

Here is an example of defining three union variables of the `hold` type:

```
union hold fit;      // union variable of hold type
union hold save[10]; // array of 10 union variables
union hold * pu;     // pointer to a variable of hold type
```

The first declaration creates a single variable, `fit`. The compiler allots enough space so that it can hold the largest of the described possibilities. In this case, the biggest possibility listed is `double`, which requires 64 bits, or 8 bytes, on our system. The second declaration creates an array called `save` with 10 elements, each 8 bytes in size. The third declaration creates a pointer that can hold the address of a `hold` union.

You can initialize a union. Because the union holds only one value, the rules are different from those in a structure. In particular, you have three choices: You can initialize a union to another union of the same type, you can initialize the first element of a union, or, with C99, you can use a designated initializer:

```
union hold valA;
valA.letter = 'R';
union hold valB = valA;  // initialize one union to another
union hold valC = {88};  // initialize digit member of union
union hold valD = {.bigfl = 118.2};  // designated initializer
```

Using Unions

Here is how you can use a union:

```
fit.digit = 23;   // 23 is stored in fit; 2 bytes used
fit.bigfl = 2.0;  // 23 cleared, 2.0 stored; 8 bytes used
fit.letter = 'h'; // 2.0 cleared, h stored; 1 byte used
```

The dot operator shows which data type is being used. Only one value is stored at a time. You can't store a `char` and an `int` at the same time, even though there is enough space to do so. It is your responsibility to write the program so that it keeps track of the data type currently being stored in a union.

You can use the `->` operator with pointers to unions in the same fashion that you use the operator with pointers to structures:

```
pu = &fit;
x = pu->digit;  // same as x = fit.digit
```

The next sequence shows what *not* to do:

```
fit.letter = 'A';
flnum = 3.02*fit.bigfl;   // ERROR ERROR ERROR
```

This sequence is wrong because a `char` type is stored, but the next line assumes that the content of `fit` is a `double` type.

However, sometimes it can be useful to use one member to place values into a union and to then use a different member for viewing the contents. Listing 15.4 in the next chapter shows an example.

Another place you might use a union is in a structure for which the stored information depends on one of the members. For example, suppose you have a structure representing an automobile. If the automobile is owned by the user, you want a structure member describing the owner. If the automobile is leased, you want the member to describe the leasing company. Then you can do something along the following lines:

```
struct owner {
     char socsecurity[12];
     ...
};

struct leasecompany  {
     char name[40];
     char headquarters[40];
     ...
};

union data {
     struct owner owncar;
```

```
     struct leasecompany leasecar;
};

struct car_data {
     char make[15];
     int status; /* 0 = owned, 1 = leased */
     union data ownerinfo;
     ...
};
```

Suppose `flits` is a `car_data` structure. Then if `flits.status` were 0, the program could use `flits.ownerinfo.owncar.socsecurity`, and if `flits.status` were 1, the program could use `flits.ownerinfo.leasecar.name`.

Anonymous Unions (C11)

Anonymous unions work much the same as anonymous structures. That is, an anonymous union is an unnamed member union of a structure or union. For instance, we can redefine the `car_data` structure as follows:

```
struct owner {
     char socsecurity[12];
     ...
};

struct leasecompany  {
     char name[40];
     char headquarters[40];
     ...
};

struct car_data {
     char make[15];
     int status; /* 0 = owned, 1 = leased */
     union {
         struct owner owncar;
         struct leasecompany leasecar;
     };
...
};
```

Now, if `flits` is a `car_data` structure, we can use `flits.owncar.socsecurity` instead of `flits.ownerinfo.owncar.socsecurity`.

Summary: Structure and Union Operators

The Membership Operator: .

General Comments:

This operator is used with a structure or union name to specify a member of that structure or union. If `name` is the name of a structure and `member` is a member specified by the structure template, the following identifies that member of the structure:

```
name.member
```

The type of `name.member` is the type specified for `member`. The membership operator can also be used in the same fashion with unions.

Example:

```
struct {
       int code;
       float cost;
} item;

item.code = 1265;
```

The last statement assigns a value to the `code` member of the structure `item`.

The Indirect Membership Operator: ->

General Comments:

This operator is used with a pointer to a structure or union to identify a member of that structure or union. Suppose that `ptrstr` is a pointer to a structure and that `member` is a member specified by the structure template. Then the statement

```
ptrstr->member
```

identifies that member of the pointed-to structure. The indirect membership operator can be used in the same fashion with unions.

Example:

```
struct {
       int code;
       float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

The last statement assigns an `int` value to the `code` member of `item`. The following three expressions are equivalent:

```
ptrst->code    item.code    (*ptrst).code
```

Enumerated Types

You can use the *enumerated type* to declare symbolic names to represent integer constants. By using the `enum` keyword, you can create a new "type" and specify the values it may have. (Actually, `enum` constants are type `int`; therefore, they can be used wherever you would use an `int`.) The purpose of enumerated types is to enhance the readability of a program. The syntax is similar to that used for structures. For example, you can make these declarations:

```
enum spectrum {red, orange, yellow, green, blue, violet};
enum spectrum color;
```

The first declaration establishes `spectrum` as a tag name, which allows you to use `enum spectrum` as a type name. The second declaration makes `color` a variable of that type. The identifiers within the braces enumerate the possible values that a `spectrum` variable can have. Therefore, the possible values for `color` are `red`, `orange`, `yellow`, and so on. These symbolic constants are termed *enumerators*. Then, you can use statements such as the following:

```
int c;
color = blue;
if (color == yellow)
   ...;
for (color = red; color <= violet; color++)
   ...;
```

Although enumerators such as `red` and `blue` are type `int`, enumerated variables are more loosely constrained to be an integral type as long as the type can hold the enumerated constants. For example, the enumerated constants for `spectrum` have the range 0–5, so a compiler could choose to use `unsigned char` to represent the `color` variable.

Incidentally, some C enumeration properties don't carry over to C++. For example, C allows you to apply the ++ operator to an enumeration variable, and the C++ standard doesn't. So if you think your code might be incorporated into a C++ program some day, you should declare `color` as type `int` in the previous example. Then the code will work with either C or C++.

`enum` Constants

Just what are `blue` and `red`? Technically, they are type `int` constants. For example, given the preceding enumeration declaration, you can try this:

```
printf("red = %d, orange = %d\n", red, orange);
```

Here is the output:

```
red = 0, orange = 1
```

What has happened is that `red` has become a named constant representing the integer 0. Similarly, the other identifiers are named constants representing the integers 1 through 5. You can use an enumerated constant anywhere you can use an integer constant. For example,

you can use them as sizes in array declarations, and you can use them as labels in a switch statement.

Default Values

By default, the constants in the enumeration list are assigned the integer values 0, 1, 2, and so on. Therefore, the declaration

```
enum kids {nippy, slats, skippy, nina, liz};
```

results in `nina` having the value `3`.

Assigned Values

You can choose the integer values that you want the constants to have. Just include the desired values in the declaration:

```
enum levels {low = 100, medium = 500, high = 2000};
```

If you assign a value to one constant but not to the following constants, the following constants will be numbered sequentially. For example, suppose you have this declaration:

```
enum feline {cat, lynx = 10, puma, tiger};
```

Then `cat` is `0`, by default, and `lynx`, `puma`, and `tiger` are `10`, `11`, and `12`, respectively.

`enum` Usage

Recall that the purpose of enumerated types is to enhance a program's readability and make it easier to maintain. If you are dealing with colors, using `red` and `blue` is much more obvious than using `0` and `1`. Note that the enumerated types are for internal use. If you want to enter a value of `orange` for `color`, you have to enter a `1`, not the word `orange`, or you can read in the string `"orange"` and have the program convert it to the value `orange`.

Because the enumerated type is an integer type, `enum` variables can be used in expressions in the same manner as integer variables. They make convenient labels for a `case` statement.

Listing 14.15 shows a short example using `enum`. The example relies on the default value-assignment scheme. This gives `red` the value `0`, which makes it the index for the pointer to the string `"red"`.

Listing 14.15 **The `enum.c` Program**

```
/* enum.c -- uses enumerated values */
#include <stdio.h>
#include <string.h>    // for strcmp(), strchr()
#include <stdbool.h>   // C99 feature
char * s_gets(char * st, int n);
```

```
enum spectrum {red, orange, yellow, green, blue, violet};
const char * colors[] = {"red", "orange", "yellow",
    "green", "blue", "violet"};
#define LEN 30

int main(void)
{
    char choice[LEN];
    enum spectrum color;
    bool color_is_found = false;

    puts("Enter a color (empty line to quit):");
    while (s_gets(choice, LEN) != NULL && choice[0] != '\0')
    {
        for (color = red; color <= violet; color++)
        {
            if (strcmp(choice, colors[color]) == 0)
            {
                color_is_found = true;
                break;
            }
        }
        if (color_is_found)
            switch(color)
        {
            case red    : puts("Roses are red.");
                break;
            case orange : puts("Poppies are orange.");
                break;
            case yellow : puts("Sunflowers are yellow.");
                break;
            case green  : puts("Grass is green.");
                break;
            case blue   : puts("Bluebells are blue.");
                break;
            case violet : puts("Violets are violet.");
                break;
        }
        else
            printf("I don't know about the color %s.\n", choice);
        color_is_found = false;
        puts("Next color, please (empty line to quit):");
    }
    puts("Goodbye!");

    return 0;
```

```
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

The code breaks out of the `for` loop if the input string matches one of the strings pointed to by the members of the `colors` array. If the loop finds a matching color, the program then uses the value of the enumeration variable to match an enumeration constant used as a case label. Here is a sample run:

```
Enter a color (empty line to quit):
blue
Bluebells are blue.
Next color, please (empty line to quit):
orange
Poppies are orange.
Next color, please (empty line to quit):
purple
I don't know about the color purple.
Next color, please (empty line to quit):

Goodbye!
```

Shared Namespaces

C uses the term *namespace* to identify parts of a program in which a name is recognized. Scope is part of the concept: Two variables having the same name but in different scopes don't conflict; two variables having the same name in the same scope do conflict. There also is a category aspect to namespaces. Structure tags, union tags, and enumeration tags in a particular scope all share the same namespace, and that namespace is different from the one used by ordinary variables. What this means is that you can use the same name for one variable and

one tag in the same scope without causing an error, but you can't declare two tags of the same name or two variables of the same name in the same scope. For example, the following doesn't cause a conflict in C:

```
struct rect { double x; double y; };
int rect;   // not a conflict in C
```

However, it can be confusing to use the same identifier in two different ways; also, C++ doesn't allow this because it puts tags and variable names into the same namespace.

`typedef`: A Quick Look

The `typedef` facility is an advanced data feature that enables you to create your own name for a type. It is similar to `#define` in that respect, but with three differences:

- Unlike `#define`, `typedef` is limited to giving symbolic names to types only and not to values.
- The `typedef` interpretation is performed by the compiler, not the preprocessor.
- Within its limits, `typedef` is more flexible than `#define`.

Let's see how `typedef` works. Suppose you want to use the term `BYTE` for one-byte numbers. You simply define `BYTE` as if it were a `char` variable and precede the definition by the keyword `typedef`, like so:

```
typedef unsigned char BYTE;
```

From then on, you can use `BYTE` to define variables:

```
BYTE x, y[10], * z;
```

The scope of this definition depends on the location of the `typedef` statement. If the definition is inside a function, the scope is local, confined to that function. If the definition is outside a function, the scope is global.

Often, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, too:

```
typedef unsigned char byte;
```

The same rules that govern the valid names of variables govern the name used for a `typedef`.

Creating a name for an existing type might seem a bit frivolous, but it can be useful. With the preceding example, using `BYTE` instead of `unsigned char` helps document that you plan to use `BYTE` variables to represent numbers rather than character codes. Using `typedef` also helps increase portability. For example, we've mentioned the `size_t` type, which represents the type returned by the `sizeof` operator, and the `time_t` type, which represents the type of value returned by the `time()` function. The C standard says `sizeof` and `time()` return integer types but leaves it up to the implementation to determine which type. The reason for this lack

of specificity is that the C standards committee feels that no one choice is likely to be the best choice for every computer platform. So they make up a new type name, such as `time_t`, and let the implementation use a `typedef` to set that name to some specific type. That way, they can provide a general prototype such as the following:

```
time_t time(time_t *);
```

On one system, `time_t` can be `unsigned long`; on another, it can be `unsigned long long`. As long as you include the `time.h` header file, your program can access the appropriate definition, and you can declare `time_t` variables in your code.

Some features of `typedef` can be duplicated with a `#define`. For example,

```
#define BYTE unsigned char
```

causes the preprocessor to replace `BYTE` with `unsigned char`. Here is one that can't be duplicated with a `#define`:

```
typedef char * STRING;
```

Without the keyword `typedef`, this example would identify `STRING` itself as a pointer-to-`char`. With the keyword, it makes `STRING` an identifier for pointers-to-`char`. Therefore,

```
STRING name, sign;
```

means

```
char * name, * sign;
```

Suppose, instead, you did this:

```
#define STRING char *
```

Then

```
STRING name, sign;
```

would translate to the following:

```
char * name, sign;
```

In this case, only `name` would be a pointer.

You can use `typedef` with structures, too:

```
typedef struct complex {
        float real;
        float imag;
} COMPLEX;
```

You can then use the type `COMPLEX` instead of the `struct` called `complex` to represent complex numbers. One reason to use `typedef` is to create convenient, recognizable names for types that turn up often. For instance, many people prefer to use `STRING` or its equivalent, as in the earlier example.

You can omit a tag when using `typedef` to name a structure type:

```
typedef struct {double x; double y;} rect;
```

Suppose you use the `typedef` like this:

```
rect r1 = {3.0, 6.0};
rect r2;
```

This is translated to

```
struct {double x; double y;} r1= {3.0, 6.0};
struct {double x; double y;} r2;
r2 = r1;
```

If two structures are declared without a tag but with identical members (with both member names and types matching), C considers the two structures to be of the same type, so assigning `r1` to `r2` is a valid operation.

A second reason for using `typedef` is that `typedef` names are often used for complicated types. For example, the declaration

```
typedef char (* FRPTC ()) [5];
```

makes `FRPTC` announce a type that is a function that returns a pointer to a five-element array of `char`. (See the upcoming discussion on fancy declarations in the next section.)

When using `typedef`, bear in mind that it does not create new types; instead, it just creates convenient labels. This means, for example, that variables using the `STRING` type we created can be used as arguments for functions expecting type pointer-to-`char`.

With structures, unions, and `typedef`, C gives you the tools for efficient and portable data handling.

Fancy Declarations

C enables you to create elaborate data forms. Although we are sticking to simpler forms, we feel it is our duty to point out some of the potentialities. When you make a declaration, the name (or identifier) can be modified by tacking on a modifier.

Modifier	Significance
`*`	Indicates a pointer
`()`	Indicates a function
`[]`	Indicates an array

C enables you to use more than one modifier at a time, and that enables you to create a variety of types, as shown in the following examples:

```
int board[8][8];   // an array of arrays of int
int ** ptr;        // a pointer to a pointer to int
int * risks[10];   // a 10-element array of pointers to int
int (* rusks)[10]; // a pointer to an array of 10 ints
int * oof[3][4];   // a 3 x 4 array of pointers to int
int (* uuf)[3][4]; // a pointer to a 3 x 4 array of ints
int (* uof[3])[4]; // a 3-element array of pointers to
                        4-element arrays of int
```

The trick to unraveling these declarations is figuring out the order in which to apply the modifiers. These rules should get you through:

1. The `[]`, which indicates an array, and the `()`, which indicates a function, have the same precedence. This precedence is higher than that of the `*` indirection operator, which means that the following declaration makes `risks` an array of pointers rather than a pointer to an array:

   ```
   int * risks[10];
   ```

2. The `[]` and `()` associate from left to right. Thus, the next declaration makes `goods` an array of 12 arrays of 50 `int`s, not an array of 50 arrays of 12 `int`s:

   ```
   int goods[12][50];
   ```

3. Both `[]` and `()` have the same precedence, but because they associate from left to right, the following declaration groups `*` and `rusks` together before applying the brackets. This means that the following declaration makes `rusks` a pointer to an array of 10 `int`s:

   ```
   int (* rusks)[10];
   ```

Let's apply these rules to this declaration:

```
int * oof[3][4];
```

The `[3]` has higher precedence than the `*`, and, because of the left-to-right rule, it is applied before the `[4]`. Hence, `oof` is an array with three elements. Next in order is `[4]`, so the elements of `oof` are arrays of four elements. The `*` tells us that these elements are pointers. The `int` completes the picture: `oof` is a three-element array of four-element arrays of pointers to `int`, or, for short, a 3×4 array of pointers to `int`. Storage is set aside for 12 pointers.

Now look at this declaration:

```
int (* uuf)[3][4];
```

The parentheses cause the `*` modifier to have first priority, making `uuf` a pointer to a 3×4 array of `int`s. Storage is set aside for a single pointer.

These rules also yield the following types:

```
char * fump(int);      // function returning pointer to char
char (* frump)(int);   // pointer to a function that returns type char
char (* flump[3])(int);// array of 3 pointers to functions that
                       //  return type char
```

All three functions take an `int` argument.

You can use `typedef` to build a sequence of related types:

```
typdef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs; // togs an array of 5 int
p_arr5 p2; // p2 a pointer to an array of 5 int
arrp10 ap; // ap an array of 10 pointers to array-of-5-int
```

When you bring structures into the picture, the possibilities for declarations truly grow baroque. And the applications... well, we'll leave that for more advanced texts.

Functions and Pointers

As the discussion on declarations illustrated, it's possible to declare pointers to functions. You might wonder whether such a beast has any usefulness. Typically, a function pointer is used as an argument to another function, telling the second function which function to use. For instance, sorting an array involves comparing two elements to see which comes first. If the elements are numbers, you can use the > operator. More generally, the elements may be a string or a structure, requiring a function call to do the comparison. The `qsort()` function from the C library is designed to work with arrays of any kind as long as you tell it what function to use to compare elements. For that purpose, it takes a pointer to a function as one of its arguments. The `qsort()` function then uses that function to sort the type—whether it be integer, string, or structure.

Let's take a closer look at function pointers. First, what does it mean? A pointer to, say, an `int` holds the address of a location in memory at which an `int` can be stored. Functions, too, have addresses, because the machine-language implementation of a function consists of code loaded into memory. A pointer to a function can hold the address marking the start of the function code.

Next, when you declare a data pointer, you have to declare the type of data to which it points. When declaring a function pointer, you have to declare the type of function pointed to. To specify the function type, you specify the function signature, that is, the return type for the function and the parameter types for a function. For example, consider this prototype:

```
void ToUpper(char *);   // convert string to uppercase
```

The type for the `ToUpper()` function is "function with `char *` parameter and return type `void`." To declare a pointer called `pf` to this function type, do this:

```
void (*pf)(char *);     // pf a pointer-to-function
```

Reading this declaration, you see the first parentheses pair associates the `*` operator with `pf`, meaning that `pf` is a pointer to a function. This makes `(*pf)` a function, which makes `(char *)` the parameter list for the function and `void` the return type. Probably the simplest way to create this declaration is to note that it replaces the function name `ToUpper` with the expression `(*pf)`. So if you want to declare a pointer to a specific type of function, you can declare a function of that type and then replace the function name with an expression of the form `(*pf)` to create a function pointer declaration. As mentioned earlier, the first parentheses are needed because of operator precedence rules. Omitting them leads to something quite different:

```
void *pf(char *);     // pf a function that returns a pointer
```

> **Tip**
>
> To declare a pointer to a particular type of function, first declare a function of the desired type and then replace the function name with an expression of the form `(*pf)`; `pf` then becomes a pointer to a function of that type.

After you have a function pointer, you can assign to it the addresses of functions of the proper type. In this context, the *name* of a function can be used to represent the address of the function:

```
void ToUpper(char *);
void ToLower(char *);
int round(double);
void (*pf)(char *);
pf = ToUpper;        // valid, ToUpper is address of the function
pf = ToLower;        // valid, ToLower is address of the function
pf = round;          // invalid, round is the wrong type of function
pf = ToLower();      // invalid, ToLower() is not an address
```

The last assignment is also invalid because you can't use a `void` function in an assignment statement. Note that the pointer `pf` can point to any function that takes a `char *` argument and has a return type of `void`, but not to functions with other characteristics.

Just as you can use a data pointer to access data, you can use a function pointer to access a function. Strangely, there are two logically inconsistent syntax rules for doing so, as the following illustrates:

```
void ToUpper(char *);
void ToLower(char *);
void (*pf)(char *);
char mis[] = "Nina Metier";
pf = ToUpper;
```

```
(*pf)(mis);     // apply ToUpper to mis (syntax 1)
pf = ToLower;
pf(mis);        // apply ToLower to mis (syntax 2)
```

Each approach sounds sensible. Here is the first approach: Because `pf` points to the `ToUpper` function, `*pf` is the `ToUpper` function, so the expression `(*pf)(mis)` is the same as `ToUpper(mis)`. Just look at the declarations of `ToUpper` and of `pf` to see that `ToUpper` and `(*pf)` are equivalent. Here is the second approach: Because the name of a function is a pointer, you can use a pointer and a function name interchangeably, hence `pf(mis)` is the same as `ToLower(mis)`. Just look at the assignment statement for `pf` to see that `pf` and `ToLower` are equivalent. Historically, the developers of C and Unix at Bell Labs took the first view and the extenders of Unix at Berkeley took the second view. K&R C did not allow the second form, but to maintain compatibility with existing code, ANSI C accepted both forms (`(*pf)(mis)` and `pf(mis)`) as equivalent. Subsequent standards have continued with this lofty ambivalence.

Just as one of the most common uses of a data pointer is an argument to a function, one of the most common uses of a function pointer is an argument to a function. For example, consider this function prototype:

```
void show(void (* fp)(char *), char * str);
```

It looks messy, but it declares two parameters, `fp` and `str`. The `fp` parameter is a function pointer, and the `str` is a data pointer. More specifically, `fp` points to a function that takes a `char *` parameter and has a `void` return type, and `str` points to a `char`. So, given the declarations we had earlier, you can make function calls such as the following:

```
show(ToLower, mis);  /* show() uses ToLower() function: fp = ToLower   */
show(pf, mis);       /* show() uses function pointed to by pf: fp = pf */
```

And how does `show()` use the function pointer passed to it? It uses either the `fp()` or the `(*fp)()` syntax to invoke the function:

```
void show(void (* fp)(char *), char * str)
{
    (*fp)(str); /* apply chosen function to str */
    puts(str);  /* display result               */
}
```

Here, for example, `show()` first transforms the string `str` by applying to it the function pointed to by `fp`, and then it displays the transformed string.

By the way, functions with return values can be used two different ways as arguments to other functions. For example, consider the following:

```
function1(sqrt);      /* passes address of sqrt function      */
function2(sqrt(4.0)); /* passes return value of sqrt function */
```

The first passes the address of the `sqrt()` function, and presumably `function1()` will use that function in its code. The second statement initially calls the `sqrt()` function, evaluates it, and then passes the return value (2.0, in this case) to `function2()`.

To show the essential ideas, the program in Listing 14.16 uses `show()` with a variety of transforming functions as arguments. The listing also shows some useful techniques for handling a menu.

Listing 14.16 **The `func_ptr.c` Program**

```
// func_ptr.c -- uses function pointers
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LEN 81
char * s_gets(char * st, int n);
char showmenu(void);
void eatline(void);     // read through end of line
void show(void (* fp)(char *), char * str);
void ToUpper(char *);   // convert string to uppercase
void ToLower(char *);   // convert string to uppercase
void Transpose(char *); // transpose cases
void Dummy(char *);     // leave string unaltered

int main(void)
{
    char line[LEN];
    char copy[LEN];
    char choice;
    void (*pfun)(char *); // points a function having a
                          // char * argument and no
                          // return value
    puts("Enter a string (empty line to quit):");
    while (s_gets(line, LEN) != NULL && line[0] != '\0')
    {
        while ((choice = showmenu()) != 'n')
        {
            switch (choice   )  // switch sets pointer
            {
                case 'u' : pfun = ToUpper;   break;
                case 'l' : pfun = ToLower;   break;
                case 't' : pfun = Transpose; break;
                case 'o' : pfun = Dummy;     break;
            }
            strcpy(copy, line);// make copy for show()
            show(pfun, copy);  // use selected function
        }
```

```
        puts("Enter a string (empty line to quit):");
    }
    puts("Bye!");

    return 0;
}

char showmenu(void)
{
    char ans;
    puts("Enter menu choice:");
    puts("u) uppercase       l) lowercase");
    puts("t) transposed case o) original case");
    puts("n) next string");
    ans = getchar();    // get response
    ans = tolower(ans); // convert to lowercase
    eatline();          // dispose of rest of line
    while (strchr("ulton", ans) == NULL)
    {
        puts("Please enter a u, l, t, o, or n:");
        ans = tolower(getchar());
        eatline();
    }

    return ans;
}

void eatline(void)
{
    while (getchar() != '\n')
        continue;
}

void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}

void ToLower(char * str)
{
    while (*str)
    {
        *str = tolower(*str);
```

```
        str++;
    }
}
void Transpose(char * str)
{
    while (*str)
    {
        if (islower(*str))
            *str = toupper(*str);
        else if (isupper(*str))
            *str = tolower(*str);
        str++;
    }
}

void Dummy(char * str)
{
    // leaves string unchanged
}

void show(void (* fp)(char *), char * str)
{
    (*fp)(str); // apply chosen function to str
    puts(str);  // display result
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

Here is a sample run:

```
Enter a string (empty line to quit):
Does C make you feel loopy?
```

```
Enter menu choice:
u) uppercase       l) lowercase
t) transposed case o) original case
n) next string
t
dOES c MAKE YOU FEEL LOOPY?
Enter menu choice:
u) uppercase       l) lowercase
t) transposed case o) original case
n) next string
l
does c make you feel loopy?
Enter menu choice:
u) uppercase       l) lowercase
t) transposed case o) original case
n) next string
n
Enter a string (empty line to quit):

Bye!
```

Note that the `ToUpper()`, `ToLower()`, `Transpose()`, and `Dummy()` functions all have the same type, so all four can be assigned to the `pfun` pointer. This program uses `pfun` as the argument to `show()`, but you can also use any of the four function names directly as arguments, as in `show(Transpose, copy)`.

You can use `typedef` in situations like these. For example, the program could have done this:

```
typedef void (*V_FP_CHARP)(char *);
void show (V_FP_CHARP fp, char *);
V_FP_CHARP pfun;
```

If you're feeling adventurous, you can declare and initialize an array of such pointers:

```
V_FP_CHARP arpf[4] = {ToUpper, ToLower, Transpose, Dummy};
```

If you then modify the `showmenu()` function so that it is type `int` and returns `0` if the user enters `u`, `1` if the user enters `l`, `2` if the user enters `t`, and so on, you could replace the loop holding the `switch` statement with the following:

```
index = showmenu();
while (index >= 0 && index <= 3)
{
    strcpy(copy, line);       /* make copy for show()  */
    show(arpf[index], copy);  /* use selected function */
    index = showmenu();
}
```

You can't have an array of functions, but you can have an array of function pointers.

You've now seen all four ways in which a function name can be used: in defining a function, in declaring a function, in calling a function, and as a pointer. Figure 14.4 sums up the uses.

function name used in a prototype declaration:	`int comp(int x, int y);`
function name used in a function call:	`status = comp(q,r);`
function name used in a function definition:	`int comp(intx, inty)` `{ ...`
function name used as a pointer in assignment:	`pfunct = comp;`
function name used as pointer argument:	`slowsort(arr,n,comp);`

Figure 14.4 Uses for a function name.

As far as menu handling goes, the `showmenu()` function shows several techniques. First, the code

```
ans = getchar();    // get response
ans = tolower(ans); // convert to lowercase
```

and

```
ans = tolower(getchar());
```

show two ways to convert user input to one case so that you don't have to test for both `'u'` and `'U'`, and so on.

The `eatline()` function disposes of the rest of the entry line. This is useful on two accounts. First, to enter a choice, the user types a letter and then presses the Enter key, which generates a newline character. That newline character will be read as the next response unless you get rid of it first. Second, suppose the user responds by typing the entire word *uppercase* instead of the letter *u*. Without the `eatline()` function, the program would treat each character in the word *uppercase* as a separate response. With `eatline()`, the program processes the *u* and discards the rest of the line.

Next, the `showmenu()` function is designed to return only valid choices to the program. To help with that task, the program uses the standard library function `strchr()` from the `string.h` header file:

```
while (strchr("ulton", ans) == NULL)
```

This function looks for the location of the first occurrence of the character `ans` in the string `"ulton"` and returns a pointer to it. If it doesn't find the character, it returns the null pointer. Therefore, this `while` loop test is a more convenient replacement for the following:

```
while (ans != 'u' && ans != 'l' && ans != 't' && ans != 'o' && ans != 'n')
```

The more choices you have to check, the more convenient using `strchr()` becomes.

Key Concepts

The information we need to represent a programming problem often is more involved than a single number or a list of numbers. A program may deal with an entity or collection of entities having several properties. For example, you might represent a client by his or her name, address, phone number, and other information. Or you might describe a movie DVD by its title, distributor, playing time, cost, and so on. A C structure lets you collect all this information in a single unit. This is very helpful in organizing a program. Rather than storing information in a scattered collection of variables, you can store all the related information in one place.

When you design a structure, it's often useful to develop a package of functions to go along with it. For example, rather than write a bunch of `printf()` statements every time you want to display the contents of a structure, you can write a display function that takes the structure (or its address) as an argument. Because all the information is in the structure, you need just one argument. If you had put the information into separate variables, you would have had to use a separate argument for each individual part. Also, if you, say, add a member to the structure, you have to rewrite the functions, but you don't have to change the function calls, which is a great convenience if you modify the design.

A union declaration looks much like a structure declaration. However, the union members share the same memory space and only one member can inhabit the union at a time. In essence, a union allows you to create a variable that can hold one value, but more than one type.

The `enum` facility offers a means of defining symbolic constants, and the `typedef` facility offers a means to create a new identifier for a basic or derived type.

Pointers to functions provide a means to tell one function which function it should use.

Summary

A C structure provides the means to store several data items, usually of different types, in the same data object. You can use a tag to identify a specific structure template and to declare variables of that type. The membership dot operator (`.`) enables you to access the individual members of a structure by using labels from the structure template.

If you have a pointer to a structure, you can use the pointer and the indirect membership operator (`->`) instead of a name and the dot operator to access individual members. To find the address of a structure, use the `&` operator. Unlike arrays, the name of a structure does not serve as the address of the structure.

Traditionally, structure-related functions have used pointers to structures as arguments. Modern C permits structures to be passed as arguments, used as return values, and assigned to structures of the same type. However, passing an address usually is more efficient.

Unions use the same syntax as structures. However, with unions, the members share a common storage space. Instead of storing several data types simultaneously in the manner of a structure,

the union stores a single data item type from a list of choices. That is, a structure can hold, say, an `int` and a `double` and a `char`, and the corresponding union can hold an `int` or a `double` or a `char`.

Enumerations allow you to create a group of symbolic integer constants (enumeration constants) and to define an associated enumeration type.

The `typedef` facility enables you to establish aliases or shorthand representations of standard C types.

The name of a function yields the address of that function. Such addresses can be passed as arguments to functions, which then use the pointed-to function. If `pf` is a function pointer that has been assigned the address of a particular function, you can invoke that function in two ways:

```
#include <math.h>    /* declares double sin(double) function */
...
double (*pdf)(double);
double x;
pdf = sin;
x = (*pdf)(1.2);  // invokes sin(1.2)
x = pdf(1.2);     // also invokes sin(1.2)
```

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What's wrong with this template?

```
structure {
       char itable;
       int  num[20];
       char * togs
}
```

2. Here is a portion of a program. What will it print?

```
#include <stdio.h>
struct house {
    float sqft;
    int rooms;
    int stories;
    char address[40];
};
int main(void)
{
  struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};
```

```
    struct house *sign;

    sign = &fruzt;
    printf("%d %d\n", fruzt.rooms, sign->stories);
    printf("%s \n", fruzt.address);
    printf("%c %c\n", sign->address[3], fruzt.address[4]);
    return 0;
}
```

3. Devise a structure template that will hold the name of a month, a three-letter abbreviation for the month, the number of days in the month, and the month number.

4. Define an array of 12 structures of the sort in question 3 and initialize it for a non-leap year.

5. Write a function that, when given the month number, returns the total days in the year up to and including that month. Assume that the structure template of question 3 and an appropriate array of such structures are declared externally.

6. a. Given the following `typedef`, declare a 10-element array of the indicated structure. Then, using individual member assignment (or the string equivalent), let the third element describe a Remarkatar lens with a focal length of 500 mm and an aperture of f/2.0.

```
typedef struct lens {    /* lens descriptor */
    float foclen;        /* focal length,mm */
    float fstop;         /* aperture        */
    char brand[30];      /* brand name      */
} LENS;
```

   b. Repeat part a., but use an initialization list with a designated initializer in the declaration rather than using separate assignment statements for each member.

7. Consider the following programming fragment:

```
struct name {
        char first[20];
        char last[20];
};
struct bem {
        int limbs;
        struct name title;
        char type[30];
};
struct bem * pb;
struct bem deb = {
```

```
        6,
        {"Berbnazel", "Gwolkapwolk"},
        "Arcturan"

};

pb = &deb;
```

a. What would each of the following statements print?

```
printf("%d\n", deb.limbs);
printf("%s\n", pb->type);
printf("%s\n", pb->type + 2);
```

b. How could you represent `"Gwolkapwolk"` in structure notation (two ways)?

c. Write a function that takes the address of a `bem` structure as its argument and prints the contents of that structure in the form shown here (assume that the structure template is in a file called `starfolk.h`):

`Berbnazel Gwolkapwolk is a 6-limbed Arcturan.`

8. Consider the following declarations:

```
struct fullname {
                char fname[20];
                char lname[20];
                };
struct bard     {
                struct fullname name;
                int born;
                int died;
                };
struct bard willie;
struct bard *pt = &willie;
```

a. Identify the `born` member of the `willie` structure using the `willie` identifier.

b. Identify the `born` member of the `willie` structure using the `pt` identifier.

c. Use a `scanf()` call to read in a value for the `born` member using the `willie` identifier.

d. Use a `scanf()` call to read in a value for the `born` member using the `pt` identifier.

e. Use a `scanf()` call to read in a value for the `lname` member of the `name` member using the `willie` identifier.

f. Use a `scanf()` call to read in a value for the `lname` member of the `name` member using the `pt` identifier.

g. Construct an identifier for the third letter of the first name of someone described by the `willie` variable.

h. Construct an expression representing the total number of letters in the first and last names of someone described by the `willie` variable.

9. Define a structure template suitable for holding the following items: the name of an automobile, its horsepower, its EPA city-driving MPG rating, its wheelbase, and its year. Use `car` as the template tag.

10. Suppose you have this structure:

```
struct gas {
        float distance;
        float gals;
        float mpg;
};
```

a. Devise a function that takes a `struct gas` argument. Assume that the passed structure contains the `distance` and `gals` information. Have the function calculate the correct value for the `mpg` member and return the now completed structure.

b. Devise a function that takes the address of a `struct gas` argument. Assume that the passed structure contains the `distance` and `gals` information. Have the function calculate the correct value for the `mpg` member and assign it to the appropriate member.

11. Declare an enumeration with the tag choices that sets the enumeration constants `no`, `yes`, and `maybe` to 0, 1, and 2, respectively.

12. Declare a pointer to a function that returns a pointer-to-`char` and that takes a pointer-to-`char` and a `char` as arguments.

13. Declare four functions and initialize an array of pointers to point to them. Each function should take two `double` arguments and return a `double`. Also, show two ways using the array to invoke the second function with arguments of 10.0 and 2.5.

Programming Exercises

1. Redo Review Question 5, but make the argument the spelled-out name of the month instead of the month number. (Don't forget about `strcmp()`.) Test the function in a simple program.

2. Write a program that prompts the user to enter the day, month, and year. The month can be a month number, a month name, or a month abbreviation. The program then should return the total number of days in the year up through the given day. (Do take leap years into account.)

3. Revise the book-listing program in Listing 14.2 so that it prints the book descriptions in the order entered, then alphabetized by title, and then in order of increased value.

4. Write a program that creates a structure template with two members according to the following criteria:

   a. The first member is a social security number. The second member is a structure with three members. Its first member contains a first name, its second member contains a middle name, and its final member contains a last name. Create and initialize an array of five such structures. Have the program print the data in this format:

      ```
      Dribble, Flossie M. —— 302039823
      ```

   Only the initial letter of the middle name is printed, and a period is added. Neither the initial (of course) nor the period should be printed if the middle name member is empty. Write a function to do the printing; pass the structure array to the function.

   b. Modify part a. by passing the structure value instead of the address.

5. Write a program that fits the following recipe:

   a. Externally define a `name` structure template with two members: a string to hold the first name and a string to hold the second name.

   b. Externally define a `student` structure template with three members: a `name` structure, a `grade` array to hold three floating-point scores, and a variable to hold the average of those three scores.

   c. Have the `main()` function declare an array of `CSIZE` (with `CSIZE = 4`) student structures and initialize the name portions to names of your choice. Use functions to perform the tasks described in parts d., e., f., and g.

   d. Interactively acquire scores for each student by prompting the user with a student name and a request for scores. Place the scores in the grade array portion of the appropriate structure. The required looping can be done in `main()` or in the function, as you prefer.

   e. Calculate the average score value for each structure and assign it to the proper member.

   f. Print the information in each structure.

   g. Print the class average for each of the numeric structure members.

6. A text file holds information about a softball team. Each line has data arranged as follows:

```
4 Jessie Joybat 5 2 1 1
```

The first item is the player's number, conveniently in the range 0–18. The second item is the player's first name, and the third is the player's last name. Each name is a single word. The next item is the player's official times at bat, followed by the number of hits, walks, and runs batted in (RBIs). The file may contain data for more than one game, so the same player may have more than one line of data, and there may be data for other players between those lines. Write a program that stores the data into an array of structures. The structure should have members to represent the first and last names, the at bats, hits, walks, and RBIs (runs batted in), and the batting average (to be calculated later). You can use the player number as an array index. The program should read to end-of-file, and it should keep cumulative totals for each player.

The world of baseball statistics is an involved one. For example, a walk or reaching base on an error doesn't count as an at-bat but could possibly produce an RBI. But all this program has to do is read and process the data file, as described next, without worrying about how realistic the data is.

The simplest way for the program to proceed is to initialize the structure contents to zeros, read the file data into temporary variables, and then add them to the contents of the corresponding structure. After the program has finished reading the file, it should then calculate the batting average for each player and store it in the corresponding structure member. The batting average is calculated by dividing the cumulative number of hits for a player by the cumulative number of at-bats; it should be a floating-point calculation. The program should then display the cumulative data for each player along with a line showing the combined statistics for the entire team.

7. Modify Listing 14.14 so that as each record is read from the file and shown to you, you are given the chance to delete the record or to modify its contents. If you delete the record, use the vacated array position for the next record to be read. To allow changing the existing contents, you'll need to use the `"r+b"` mode instead of the `"a+b"` mode, and you'll have to pay more attention to positioning the file pointer so that appended records don't overwrite existing records. It's simplest to make all changes in the data stored in program memory and then write the final set of information to the file. One approach to keeping track is to add a member to the book structure that indicates whether it is to be deleted.

8. The Colossus Airlines fleet consists of one plane with a seating capacity of 12. It makes one flight daily. Write a seating reservation program with the following features:

   a. The program uses an array of 12 structures. Each structure should hold a seat identification number, a marker that indicates whether the seat is assigned, the last name of the seat holder, and the first name of the seat holder.

b. The program displays the following menu:

```
To choose a function, enter its letter label:
a) Show number of empty seats
b) Show list of empty seats
c) Show alphabetical list of seats
d) Assign a customer to a seat assignment
e) Delete a seat assignment
f) Quit
```

c. The program successfully executes the promises of its menu. Choices `d)` and `e)` require additional input, and each should enable the user to abort an entry.

d. After executing a particular function, the program shows the menu again, except for choice `f)`.

e. Data is saved in a file between runs. When the program is restarted, it first loads in the data, if any, from the file.

9. Colossus Airlines (from exercise 8) acquires a second plane (same capacity) and expands its service to four flights daily (Flights 102, 311, 444, and 519). Expand the program to handle four flights. Have a top-level menu that offers a choice of flights and the option to quit. Selecting a particular flight should then bring up a menu similar to that of exercise 8. However, one new item should be added: confirming a seat assignment. Also, the quit choice should be replaced with the choice of exiting to the top-level menu. Each display should indicate which flight is currently being handled. Also, the seat assignment display should indicate the confirmation status.

10. Write a program that implements a menu by using an array of pointers to functions. For instance, choosing `a` from the menu would activate the function pointed to by the first element of the array.

11. Write a function called `transform()` that takes four arguments: the name of a source array containing type `double` data, the name of a target array of type `double`, an `int` representing the number of array elements, and the name of a function (or, equivalently, a pointer to a function). The `transform()` function should apply the indicated function to each element in the source array, placing the return value in the target array. For example, the call

```
transform(source, target, 100, sin);
```

would set `target[0]` to `sin(source[0])`, and so on, for 100 elements. Test the function in a program that calls `transform()` four times, using two functions from the `math.h` library and two suitable functions of your own devising as arguments to successive calls of the `transform()` function.