

Reading: Structures in C

Reading Objective:

In this reading, you will learn about user-defined data types in C. You will also learn how to create your own data and how to use them in programming.

The Need for Structures in C

Until now, we have seen a lot of built-in data types in C, such as `int`, `float`, and `char`, etc. These data types can represent integers, real numbers, characters, and strings. We can also create arrays of these data types.

But now, let us consider a diverse group of data items that are somehow related to each other. For example, let us take a student database, where each student has an associated ID, name, and CGPA. How would we implement this database in C?

Based on what we have learned so far, we could create three different arrays as follows:

```
int ID[n];  
  
char* name[n];  
  
float CGPA[n];
```

Now, for each student, we have stored their data in three separate arrays. If we have to add or delete a student's records, it has to be done separately in all three arrays. Moreover, it would require three parameters to pass a student's data into a function. Also, there is no real linkage between the ID and the name of the student in our code.

Consider another example—say we want to represent a point in a three-dimensional space. This can be done by specifying the x, y, and z coordinates. If we want to store a bunch of such points, we would have to create three different arrays again:

```
int x[n];  
  
int y[n];  
  
int z[n];
```

Would not it be much better if we were able to represent a Point or a Student as one single entity, just as it is in real life? Fortunately, C offers this functionality! We can design our own custom data types using the **struct** keyword in C.

Structures in C

A **structure** is a data type that is designed by the user. It is a combination of primitive data types, such as **int**, **float**, **char**, or **bool**.

We can define a structure as follows:

```
struct struct_name {  
    data_type member_1;  
    data_type member_2;  
    ...  
  
};
```

Note the semicolon at the end of structure definition! Not adding a semicolon would give you an error. Structures are usually defined **globally**, that is, outside of `int main()` or any other function.

Let us now try to define the structures we discussed in the previous sections:

```
struct Student {  
    int ID;  
    char name[20];  
    float CGPA;  
  
};
```

```
struct Point {  
    int x, y, z;  
  
};
```

Now that we have defined the structure using the **struct** keyword, we can declare structure variables just like we declared variables of primitive data types. The only difference is that a **struct** keyword is required whenever declaring a variable, as follows:

```
struct Student s1;  
  
struct Point p1, p2;
```

Now, **s1** is a Student variable and has associated attributes: **ID**, **name**, and **CGPA**. Similarly, **p1** and **p2** are point variables.

Accessing Members of a Structure

We can access members of a struct using the **dot** operator. For example, it is possible to access the ID of student **s1** as **s1.ID**.

In general, to access any member, just call `structure_variable.member_name`

As another example, we can also directly transfer values into the structure by taking input. Consider the following code snippet:

```

struct Student {
    int ID;
    char name[20];
    float CGPA;
};

int main() {
    struct Student s;
    scanf("%d", &s.ID);
    scanf("%s", s.name);
    scanf("%f", &s.CGPA);
}

```

This takes the student's ID, name, and CGPA as input and stores them in their respective attributes inside the student structure.

Initializing a Structure Variable

We can specify all attributes of a structure during initialization itself as comma-separated entries inside curly braces:

```

struct Student s1 = {5134, "Alice", 9.10};

struct Student s2 = {84, "Bob", 8.25};

```

Operations on Structures

Not many operations can be performed on structures as a whole in C. The only possible operators that can be used with structures are =, dot, ->, and &. Equals ('=') is the assignment operator. Assignment works just as in the case of primitive data types.

For example, **s1 = s2** will give all attributes of s1 the same value that they had in s2. Similarly, & retains its usual significance with structures. However, dot ('.') and arrow ('->') are operators exclusively meant for accessing members of a structure.

Note that we cannot use any other operations with structures. We cannot add two structures together, or find their bitwise XOR, or check equality using '=='.

Question: If the '==' operator is not allowed to be used with structures, how can we check if two structures are equal?

Answer: Compare all attributes individually.

```

if(s1.ID == s2.ID && s1.CGPA == s2.CGPA && strcmp(s1.name, s2.name)
== 0) {

    printf("The two structs are equal!");

} else {

    printf("Not equal!");

}

```

Note: The strcmp() function compares two-character arrays for equality. It returns 0 if both strings are equal.

Extra Note: typedef

It seems unnecessarily lengthy, typing “struct” before every variable declaration, does not it? Fortunately, C has a feature that can bypass this! The keyword **typedef** can create a simple synonym for the words **struct** <struct_name>, as follows:

```

typedef struct Student stu;

stu s2, s3;

```

The **typedef** keyword here basically does this: whenever “stu” is encountered in the code, the compiler will internally convert it to “struct Student.” This saves time for us as we have to type less code!

We can even combine **typedef** with the structure definition as follows:

```

typedef struct {

    float real, imag;

} Complex;

Complex c1, c2;

```

Reading Summary:

In this reading, you have learned the following:

- The need for user-defined data types in C
- How to define structures and create structure variables (usage of the **struct** keyword)
- How to access members of a structure and other possible operations with structures
- How to use **typedef** to improve readability and reduce typing effort in coding

Practice Lab 1: Structures in C

Assignment Brief

Practice Lab 1_Question_1

Create a Point structure as taught in the lectures. Then, create two-point objects representing the points (3, 2) and (-1, -5). Print the x-coordinate and y-coordinate of the points to standard output.

Practice Lab 1_Question_2

Create a structure called Employee. It should contain the ID, name, and job of the employees. Read the ID, first name, and job of three employees as input, and print the details of the employee whose ID is 2.

| Testcase1 |
|--|
| Input: Enter the ID of employee: 1 Enter the name of employee: Aditya Enter the job of employee: Intern Enter the ID of employee: 3 Enter the name of employee: Rahul Enter the job of employee: Manager Enter the ID of employee: 2 Enter the name of employee: Tanveer Enter the job of employee: Accountant |
| Output: The details are: ID = 2, Name = Tanveer, Job = Accountant |

Practice Lab 1_Question_3

Write a program that takes two distances as input. The distances are in the form of X kilometers and Y meters.

For example, 23 68 means 23 kilometers and 68 meters. Store the distances in a structure. After that, find the difference between the two distances. Store this difference in another structure(1,000 meters = 1 kilometer). It is guaranteed that the second distance is greater than the first distance.

A structure definition has been provided, complete the missing parts of the code!

| Testcase1 | Testcase2 |
|---|---|
| Input: Enter first distance = 5 40 Enter second distance = 10 60 Output: The difference is = 5 kilometers and 20 meters | Input: Enter first distance = 1 500 Enter second distance = 2 0 Output: The difference is = 0 kilometers and 500 meters |
| | |

Reading: More Structures

Reading Objective:

In this reading, you will learn about nested structures and how to create arrays of structures.

Nested Structures

A structure can even contain other structures as attributes. A structure can have another structure, and the inner structure can contain more structures as its members, and so on. This can help in modeling more complex data.

Example:

```
struct Address {  
    int pincode;  
    char city[20];  
    char state[20];  
};  
  
typedef struct {  
    char first_name[20], middle_name[20], last_name[20];  
} Name;  
  
struct Student {  
    int ID;  
    float CGPA;  
    struct Address ad;  
    Name name;  
};
```

In this example, we created a struct Student that contains, as its members, two more structs, Address, and name.

Question: How can we access the last name of a student in the above definition?

Answer:

```
struct Student s;  
s.name.last_name
```

So, the dot operator can be used multiple times to access inner struct data.

It is also possible to initialize nested structures, for example:

```
struct Student s1 = {84, 9.64, {333031, "Pilani", "Rajasthan"},  
{"Tanveer", "", "Singh"}};
```

Array of Structures

It is possible to create arrays of structures, just as we could create arrays of int, char, bool, etc. The declaration is pretty straightforward. For example:

```
struct Student stu_records[500];
```

This creates an array of Student, of size 500. Now, let us say we want to access the CGPA of the 26th student in this array. We can do that using stu_records[25]. CGPA (remember that arrays are 0-indexed, so the 26th student would be at location 25!).

Structures and Functions

The functionality of structures can be enhanced using functions. Structures can be passed as parameters to functions (both pass-by-value and pass-by-reference are supported) and can be returned from functions too. For example, we can create a function to add two complex numbers together or to print the real and imaginary parts of a complex number.

```
struct Complex {  
    float real, imag;  
};  
  
// Pass-by-value  
  
struct Complex add(struct Complex a, struct Complex b) {  
    struct Complex c = {a.real + b.real, a.imag + b.imag};  
    return c;  
}  
  
void display(struct Complex c) {  
    printf("(%f,%f)", c.real, c.imag);  
}
```

Reading Summary:

In this reading, you have learned the following:

- Nested structures in C
- How to declare arrays of structures and access member variables while dealing with arrays of structures
- How to pass structure variables as arguments to functions, and how to return structure variables from functions

Practice Lab 2: More Structures

Assignment Brief

PracticeLab2_Question1

Write a program that takes N points in the x-y coordinate plane as input, and then print the point with the maximum distance from the origin. Try to use structures in your program!

Hint: The distance of a point from the origin is equal to $\sqrt{x^2 + y^2}$.

The input should be taken in the following format—the first line will contain only a single integer N as input. After that, the next N lines will contain two real values each, first x coordinate then y coordinate.

| Testcase1 |
|-----------------------------|
| Input: |
| 3 |
| 4 5 |
| 1.5-1.5 |
| 0 5 |
| Output: |
| Maximum distance = 6.403124 |

PracticeLab2_Question2

The definition of the Complex structure is given in the file Question2.c, which is used to represent complex numbers in C. Also, the function signature of the addition function is provided, to add two complex numbers.

1. Complete the addition function, which should return the sum of the two complex numbers.
2. Create two new functions, subtract and multiply, which return the difference and product of the two complex numbers respectively.

Note: You should not make any changes inside int main().

Hint: Given two complex numbers $(a + ib)$ and $(c + id)$, their sum is given by $(a + c) + i(b + d)$, their difference is given by $(a - c) + i(b - d)$, and their product is given by $(ac - bd) + i(ad + bc)$.

For example: If the complex numbers are $(3 + 4i)$ and $(7 + 12i)$, their sum is $(10 + 16i)$, their difference is $(-4 - 8i)$, their product is $(-27 + 64i)$.

PracticeLab2_Question3

Write a function that takes two dates as parameters, compares them, and returns which date is older.

1. You should make a structure called Date, and store both the dates in the structure.
2. You should make a function that takes two Dates as input and returns the larger Date.
3. Now, write a program so that it takes N = 10 dates as input, compares all of them, and prints the oldest date.

Input will be given in the following format—each line will contain a new date in the form “D M Y.”

| Testcase1 |
|--|
| Input: 12 9 2002 27 1 2003 Output: The oldest date is 12-9-2002 |

Explanation: The dates correspond to 12th September, 2002 and 27th January, 2003. The older date is printed.

Reading: Example Cases

Reading Objective:

In this reading, you will learn about two comprehensive examples. The first one is on the point structure, which is essential for coordinate geometry, and the second one is on a student record/database.

Main Reading Section:

Case Study 1: Structures for Co-Ordinate Geometry

Question: Define a structure for representing two-dimensional (2D) grid points on a plane. Using the structure, define functions to answer the following questions:

- In which quadrant does a given point lie?
- Does a given point lie on the coordinate axes?
- Do the two given points lie on the same quadrant?

Answer: Let us start by defining the point structure and the function prototypes.

```
#include <stdio.h>

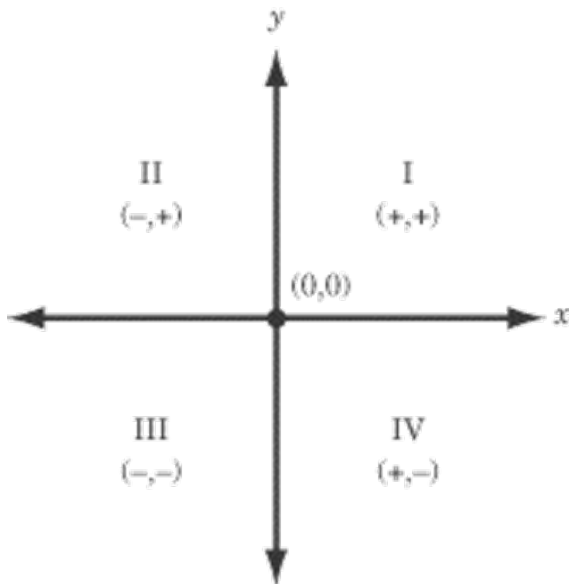
#include <stdbool.h>

typedef struct {
    int x, y;
} Point;

int FindQuadrant(Point);

bool SameQuadrants(Point, Point);
```

The function FindQuadrant will return 1/2/3/4 based on the quadrant the point lies in, and 0 if it lies on the axes. Hint: see the figure given below.



We can implement the FindQuadrant function as follows:

```
int FindQuadrant(Point p) {
    if(p.x > 0 && p.y > 0) return 1;
    if(p.x < 0 && p.y > 0) return 2;
    if(p.x < 0 && p.y < 0) return 3;
    if(p.x > 0 && p.y < 0) return 4;
    return 0;
}
```

Now, we can use this function to implement the SameQuadrants function.

```
bool SameQuadrants(Point p1, Point p2) {
    if(FindQuadrant(p1) == FindQuadrant(p2)) return true;
    return false;
}
```

Point to ponder: This function will return true even when both points are on the coordinate axes. How would you change the function if the result should be false for points on coordinate axes?

Answer:

```
bool SameQuadrants(Point p1, Point p2) {
    if(FindQuadrant(p1) == 0 || FindQuadrant(p2) == 0) return false;
    if(FindQuadrant(p1) == FindQuadrant(p2)) return true;
    return false;
}
```

Question: Add the following functionality to the Point program created above.

- What is the distance between two given points?
- Are the three given points collinear?

$$D(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Recall that the distance between two points is given by the equation:

We can implement this in code as follows (remember that we need to include math.h header file to use the sqrt function!):

```
#include <math.h>

float dist(Point p1, Point p2) {
    int d1 = (p1.x - p2.x) * (p1.x - p2.x);
    int d2 = (p1.y - p2.y) * (p1.y - p2.y);
    return sqrt(d1 + d2);
}
```

Also, recall that three points are collinear if the area of the triangle formed by them is 0. The area of the triangle formed by three points is equal to:

$$A(p_1, p_2, p_3) = \frac{1}{2} |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$$

We just need to check if this area is 0. Here is how to do it:

```
bool collinear(Point p1, Point p2, Point p3) {
    int area = p1.x*(p2.y - p3.y) + p2.x*(p3.y - p1.y) + p3.x*(p1.y -
p2.y);
    return (area == 0); // true if area == 0, else false.
}
```

Let us now write an example program to show how the code we just wrote would work. Try to write the program in your local/online compiler and see if the outputs match.

```

int main() {
    Point p1 = {0, 0}, p2 = {5, 5}, p3 = {10, 10}, p4 = {-5, 0}, p5 =
{-5, -5};

    printf("Point p2 lies in quadrant = %d\n", FindQuadrant(p2));
    printf("Point p5 lies in quadrant = %d\n", FindQuadrant(p5));
    if(SameQuadrants(p2, p3))
        printf("Points p2 and p3 lie in the same quadrants.\n");
    else
        printf("Points p2 and p3 don't lie in the same quadrants.\n");
    printf("Distance between p3 and p5 is = %f\n", dist(p3, p5));
    printf("Collinearity of p1, p2, p3 = %d\n", collinear(p1, p2, p3));
    printf("Collinearity of p1, p2, p4 = %d\n", collinear(p1, p2, p4));
}

```

Output:

Point p2 lies in quadrant = 1

Point p5 lies in quadrant = 3

Points p2 and p3 lie in the same quadrants.

Distance between p3 and p5 is = 21.213203

Collinearity of p1, p2, p3 = 1

Collinearity of p1, p2, p4 = 0

Case Study 2: Student Database

Question: Define a structure for representing an array of students. Each student must have a name, ID, and fields for marks in two subjects. Using the structure, write functions to:

- Print the name and ID of all students whose average marks are greater than 70.

Answer:

```

#include <stdio.h>

const int NUM = 4;

typedef struct {
    char name[20];
    int id;
    int marks[2];
} Student;

int main() {
    Student list[4] = {
        {"Abhinav", 1, {100, 100}},
        {"Manish", 2, {10, 5}},
        {"Shivam", 3, {66, 80}},
        {"Harsh", 4, {85, 60}}
    };

    for(int i = 0; i < NUM; i++) {
        float avg = (list[i].marks[0] + list[i].marks[1])/2.0;
        if(avg > 70) {
            printf("%s %d %f\n", list[i].name, list[i].id, avg);
        }
    }
}

```

Output:

Abhinav 1 100.000000

Shivam 3 73.000000

Harsh 4 72.500000

Question: Modify the Student structure to store the date of joining for each student. Now, write the functionality to:

- Print the name and ID of students whose date of joining is 2020 or later.

Hint: Recall how to use nested structures!

Answer: Date is not easy to represent using primitive data types. It is a combination of date, month, and year. It makes sense to make a new structure for the date. We can now define our structures as follows:

```
typedef struct {
    int day, month, year;
} Date;
```

```
typedef struct {
    char name[20];
    int id;
    int marks[2];
    Date date;
} Student;
```

The rest of the program is fairly straightforward after this:

```
int main() {
    Student list[4] = {
        {"Abhinav", 1, {100, 100}, {21, 1, 2021}},
        {"Manish", 2, {10, 5}, {15, 9, 2020}},
        {"Shivam", 3, {66, 80}, {7, 12, 2020}},
        {"Harsh", 4, {85, 60}, {12, 4, 2012}}
    };

    for(int i = 0; i < NUM; i++) {
        if(list[i].date.year >= 2020) {
            printf("%s %d %d-%d-%d\n", list[i].name, list[i].id,
                list[i].date.day, list[i].date.month, list[i].date.year);
        }
    }
}
```

Output:

Abhinav 1 21-1-2021

Manish 2 15-9-2020

Shivam 3 7-12-2020

Reading Summary:

- Examples of various real-world applications of structures in C.
- In the first example, you represented points on a two-dimensional (2D) cartesian plane as structures and performed various operations of coordinate geometry on them.
- In the second example, you created a student database to store various details of students and performed various queries on the database

Lesson 1: Structures in a C Program

Chapter 14, Sections 14.1–14.4

14 User-Defined Data Types

WHAT TO LEARN

- How *structures* extend the capabilities of arrays.
- The two-part sequence of declaring and defining structures.
- The restricted operations that can be performed on structures.
- How structures differ from arrays when used in functions.
- When a structure requires to be accessed by a pointer.
- How a *union* interprets a section of memory in multiple ways.
- The use of *bit fields* for handling data in units of bits rather than bytes.
- Enhance the readability of programs using *enumerators*.

14.1 STRUCTURE BASICS

C offers a basket of data types for handling different levels of complexity in the organization of data. The primary data types (like `int` and `float`) are meant for using simple and small amounts of data. The array is offered as a derived data type for handling a group of data items of the same type. Real-life situations often involve the use of heterogeneous data that need a mix of data types to be handled as a single unit. Structures and unions meet this requirement perfectly.

A group of data items are often logically related to one another. For instance, the attributes of an employee include the name, address, date of birth, salary, and so forth. Each of these attributes can be encapsulated as a *member* or *field* in a structure which can be manipulated as a *single* unit. Each member or field, however, retains its separate identity in the structure. That is not all; a member representing a date, for instance, can itself be a structure comprising the day, month and year as its members.

A *structure* is a data type that is designed by the user. However, unlike the other data types, a structure variable is based on a *template* that must be created first. Each member of the structure is accessed by the notation *structure.member*, and can be used in practically the same way as a variable. A structure

member can be a primary data type, an array or even another structure (or union). To accommodate multiple structures having the same type, C also supports an array of structures.

The advantage of using a structure is strongly felt when it is used as an argument to a function. It is more convenient to pass a structure containing all employee details as a single argument to a function rather than pass its members as separate arguments. A function can also return a structure, and if copying an entire structure bothers you, you can even pass a pointer to a structure as a function argument. Structures present the best of both worlds as you'll see soon.

14.2 DECLARING AND DEFINING A STRUCTURE

Because the components of a structure are determined by the user and not by C, a structure variable is created in two steps which may or may not be combined:

- Declare the structure to create a template which specifies the components of the structure. Declaration simply makes type information available to the compiler. However, the compiler doesn't allocate memory by seeing the declaration.
- Define a variable that conforms to the template. The compiler allocates memory for the variable and creates an *instance* of the template.

Declaration uses the **struct** keyword, optionally followed by the name of the structure and a set of specifications for each member or field. The following generalized syntax declares a structure named *struct_name*:

```
struct struct_name {
    data_type_1 member_name_1;           First member
    data_type_2 member_name_2;           Second member
    ...
    data_type_n member_name_n;           Last member
};
```

struct_name represents the *structure tag*. It is followed by the structure body enclosed in a set of curly braces terminated with a semicolon. The body contains the names of the members preceded by their data types. For instance, *member_name_1* has the data type *data_type_1*, which could be a primary, derived or user-defined type (including another structure). Each member specification is actually a declaration statement, the reason why it is terminated with a semicolon.

On seeing the declaration, the compiler determines how to organize the structure members in memory. It's only when you actually create structure variables (by definition), that memory is allocated by the compiler for the structure. The syntax of the definition also specifies the keyword **struct**, the structure tag (*struct_name*) and the variable name (*struct_var*):

```
struct struct_name struct_var;
```

Let's now declare and define a structure comprising three members that contain information of an audio CD sold in a store. This information is represented by the title, the quantity in stock and the price. The following declaration creates a structure named `music_cd`:

```
struct music_cd {
    char title[30];
    short qty;
    float price;
};
```

*music_cd is the structure tag
First member*

Declaration ends with ;

The structure tag (here, `music_cd`) is simply an identifier that is needed to create variables of this type. The names of the three members (`title`, `qty` and `price`) are preceded by their types and followed by the `;` terminator. Like `int` and `float`, `music_cd` is a data type, but a user-defined one.

You can now define one or more structure variables based on the created template. Just as a variable definition begins with the data type (`int x;`), the same is true for the definition of a structure:

```
struct music_cd disk1;
struct music_cd disk1, disk2;
```

*Allocates memory for the members
Can define multiple variables*

The compiler allocates memory for the variables `disk1` and `disk2`, which have `struct music_cd` as their data type. These two words representing the type must precede every definition of this structure. However, C supports an abbreviation feature called **typedef** (14.4.2) that can create a simple synonym for the words `struct music_cd`.



Takeaway: Declaration of a structure creates a template but doesn't allocate memory for the members. Memory is allocated when variables based on this template are created by definition.



Note: The terms *declaration* and *definition* are often used interchangeably with structures. This book, however, follows the convention adopted by Kernighan and Ritchie to create a template by declaration and an instance (i.e., an actual object) by definition.

14.2.1 Accessing Members of a Structure

The variable `disk1` (or `disk2`) is a real-life object of type `struct music_cd`. The individual members are connected to their respective variables with a dot, the member operator. Thus, the members of `disk1` are accessed as `disk1.title`, `disk1.qty` and `disk1.price`. These members can be treated like any other variable, which means that you can assign values to them in the usual manner:

```
strcpy(disk1.title, "Mozart");
disk1.qty = 3;
disk1.price = 10.75;
```

Note that it is not possible to use `disk1.title = "Mozart";` because `title` is an array which signifies a constant pointer. You can also assign values to these members using **scanf**:

```
scanf("%s", disk1.title);
scanf("%hd", &disk1.qty);
```

The same dot notation must be used to print the values of these members. For instance, **printf("%f", disk1.price);** displays the value of the member named `price`.

Another structure in the program may also have a member named `title`, but the name `disk1.title` will be unique in the program. The naming conventions used for simple variables apply equally to structure members. This means that the name can't begin with a digit but can use the underscore.

14.2.2 Combining Declaration, Definition and Initialization

The preceding definitions created the variables `disk1` and `disk2` without initializing them. It is possible to combine declaration and definition by adding the variable name after the closing curly brace (Form 1). It is also possible to simultaneously initialize the variable (Form 2):

| | |
|---|--|
| <pre>struct music_cd { char title[30]; short qty; float price; } disk1;</pre> | <pre>struct music_cd { char title[30]; short qty; float price; } disk2 = {"Beethoven", 3, 12.5};</pre> |
|---|--|

Form 1

Form 2

For an initialized structure, the variable name is followed by an `=` and a list of comma-delimited values enclosed by curly braces. These initializers obviously must be provided in the right order. In Form 2, "Beethoven" is assigned to `title` and 3 to `qty`.

You can create multiple variables and initialize them at the same time, as shown in the following snippets which represent the last line of declaration:

```
} disk1, disk2, disk3;
} disk1 = {"Beethoven", 3, 12.5}, disk2 = {"Mahler", 4, 8.75};
```

Like with arrays, you can partially initialize a structure. If you initialize only `title`, the remaining members, `qty` and `price`, are automatically assigned zeroes.

By default, uninitialized structures have junk values unless they are global. A structure defined before `main` is a global variable, which means that the members are automatically initialized to zero or `NULL`.

14.2.3 Declaring without Structure Tag

If declaration and definition are combined, the structure tag can be omitted. This is usually done when no more structure variables of that type require to be defined later in the program. The following statement has the tag missing:

```
struct {
    char title[30];
    short qty;
    float price;
} disk4 = {"Borodin", 5}, disk5 = {"Schubert"};
```

No tag specified

You can't subsequently create any more variables of this type. Here, both `disk4` and `disk5` are partially initialized, which means that `disk4.price`, `disk5.qty` and `disk5.price` are automatically initialized to zeroes.



Note: The structure tag is necessary when a structure is declared and defined at two separate places.

14.3 intro2structures.c: AN INTRODUCTORY PROGRAM

Program 14.1 declares a structure named `music_cd` containing three members. One of them is an array of type `char` meant to store a string. In this program, four variables are created and assigned values using different techniques:

- `disk1` Declared, defined and initialized simultaneously.
- `disk2` Defined separately but initialized fully.
- `disk3` Defined separately and initialized partially; two members are assigned values by `scanf`.
- `disk4` Defined separately but not initialized at all; its members are assigned individually.

```
/* intro2structures.c: Declares and initializes structures. */
#include <stdio.h>
#include <string.h>

int main(void)
{
    struct music_cd {                /* Declares structure for music_cd */
        char title[27];             /* Member can also be an array */
        short qty;
        float price;
    } disk1 = {"Bach", 3, 33.96};     /* Defines and initializes ...
                                     ... structure variable disk1 */
    printf("The size of disk1 is %d\n", sizeof disk1);

    /* Three more ways of defining structure variables */
    struct music_cd disk2 = {"Bruckner", 5, 6.72};
    struct music_cd disk3 = {"Handel"}; /* Partial initialization */
    struct music_cd disk4;

    /* Assigning values to members */
    strcpy(disk4.title, "Sibelius");
    disk4.qty = 7;
    disk4.price = 10.5;

    printf("The four titles are %s, %s, %s and %s\n",
           disk1.title, disk2.title, disk3.title, disk4.title);

    /* Using scanf */
    fputs("Enter the quantity and price for disk3: ", stdout);
    scanf("%hd%f", &disk3.qty, &disk3.price);
    printf("%s has %hd pieces left costing %.2f a piece.\n",
           disk3.title, disk3.qty, disk3.price);
    return 0;
}
```

PROGRAM 14.1: **intro2structures.c**

```

The size of disk1 is 36
The four titles are Bach, Bruckner, Handel and Sibelius
Enter the quantity and price for disk3: 3 7.75
Handel has 3 pieces left costing 7.75 a piece.

```

PROGRAM OUTPUT: **intro2structures.c**

Note that **sizeof** computes the total memory occupied by a structure. This is not necessarily the sum of the sizes of the members. For reasons of efficiency, the compiler tries to align one or more members on a word boundary. Thus, on this machine having a 4-byte word, `title` occupies 28 bytes (7 words) even though it uses 27 of them. `qty` and `price` individually occupy an entire word (4 bytes each), but `qty` uses two of these bytes. `disk1` thus needs 33 bytes ($27 + 2 + 4$) but it actually occupies 36 bytes.

Alignment issues related to structures lead to the creation of *slack bytes* or *holes* in the allocated memory segment (Fig. 14.1). If you reduce the size of the array to 26, no space is wasted and **sizeof disk1** evaluates to 32.

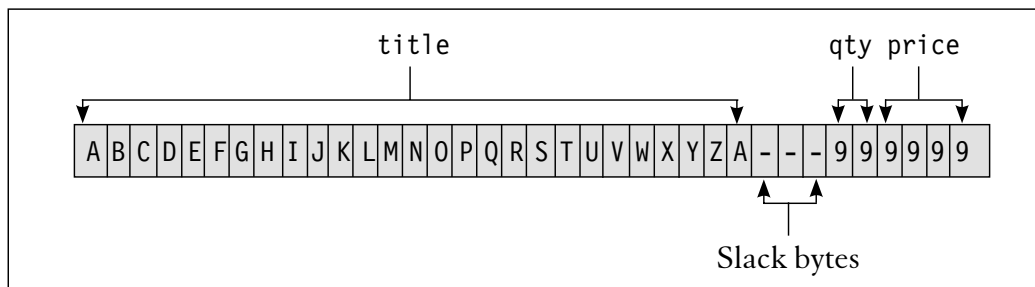


FIGURE 14.1 Memory Layout of Members of `music_cd`

14.4 IMPORTANT ATTRIBUTES OF STRUCTURES

Because a structure can include any data type as a member (including pointers, unions and other structures), this diversity leads to restrictions on the operations that can be performed on them. Be prepared for a surprise or two when you consider the following features of structures:

- *There are no name conflicts between structure templates, their instantiated variables and members.*
It is thus possible to have the same names for all of them as shown in the following:

```

struct x {
    char x[30];
    short y;
} x, y;

```

Name of template is the same ...
... as a member and ...

... a structure variable.

Even though the compiler finds nothing wrong in this declaration and definition, a programmer would do well to avoid naming structures and members in this manner.

- *The =, dot, -> and & are the only operators that can be used on structures.* Two of them (= and &) retain their usual significance, while the dot and -> are exclusively meant for use with structures.

- *A structure can be copied to another structure provided both objects are based on the same template.* The first statement in the following code segment defines a variable `disk2` based on a template declared earlier. The second statement uses the assignment operator to copy all members of `disk2` to `disk3`:

```
struct music_cd disk2 = {"mozart", 20, 9.75};
struct music_cd disk3 = disk2;
```

This feature is not available with arrays; all array elements have to be copied individually.

- *No arithmetic, relational or logical operations can be performed on structures even when the operation logically makes sense.* It is thus not possible to add two structures to form a third structure even if the members are compatible.
- *It is not possible to compare two structures using a single operator even though such comparison could be logically meaningful.* Each member has to be compared individually as shown by the following code:

```
if (strcmp(disk2.title, disk3.title) == 0
    && disk2.qty == disk3.qty
    && disk2.price == disk3.price)
    fputs("disk2 and disk3 are identical structures\n", stdout);
```

If a structure contains 20 members, you need to use 20 relational expressions to test for equality. Unfortunately, C doesn't support a better option.

- *When a structure is passed by name as an argument to a function, the entire structure is copied inside the function.* This doesn't happen with arrays where only a pointer to the first element of the array is passed. However, copying can be prevented by passing a pointer to a structure as a function argument.
- *A member of a structure can contain a reference to another structure of the same type.* This property of *self-referential* structures is used for creating *linked lists*.

Barring the last attribute in the list, the other attributes will be examined in this chapter. Self-referential structures are discussed in Chapter 16.



Takeaway: An array and structure differ in three ways: 1) A structure variable can be assigned to another structure variable of the same type. 2) The name of a structure doesn't signify a pointer. 3) When the name of a structure is passed as an argument to a function, the entire structure is copied inside the function.

14.4.1 `structure_attributes.c`: Copying and Comparing Structures

Program 14.2 demonstrates the use of (i) the `=` operator to copy a four-member structure named `cricketer`, (ii) a set of four relational expressions to individually compare the members of two structure variables, `bat1` and `bat2`. `bat1` is declared, defined and initialized simultaneously and is then copied to `bat2` before they are compared.

```

/* structure_attributes.c: Copies a structure and tests whether two
                           structures are identical. */
#include <stdio.h>
#include <string.h>
int main(void)
{
    struct cricketer {
        char name[30];
        short runs;
        short tests;
        float average;
    } bat1 = {"Don Bradman", 6996, 52, 99.94};
    printf("bat1 values: %s, %hd, %hd, %.2f\n",
          bat1.name, bat1.runs, bat1.tests, bat1.average);

    struct cricketer bat2;
    bat2 = bat1;                      /* Copies bat1 to bat2 */
    printf("%s scored %hd runs in %hd tests at an average of %.2f.\n",
          bat2.name, bat2.runs, bat2.tests, bat2.average);

    /* Compares members of two structures */
    if (strcmp(bat2.name, bat1.name) == 0 && bat2.runs == bat1.runs
        && bat2.tests == bat1.tests && bat2.average == bat1.average)
        fputs("The two structures are identical\n", stdout);
    else
        fputs("The two structures are not identical\n", stdout);
    return 0;
}

```

PROGRAM 14.2: structure_attributes.c

```

bat1 values: Don Bradman, 6996, 52, 99.94
Don Bradman scored 6996 runs in 52 tests at an average of 99.94.
The two structures are identical

```

PROGRAM OUTPUT: structure_attributes.c

14.4.2 Abbreviating a Data Type: The typedef Feature

The **typedef** keyword is used to abbreviate the names of data types. Using a data type LL instead of long long involves less typing. Also, use of meaningful names makes programs easier to understand. The syntax of **typedef** is simple; follow **typedef** with the existing data type and its proposed synonym:

```
typedef data_type new_name;
```

You can now use *new_name* in place of *data_type*. Some of the declarations we have used previously can now be abbreviated using **typedef**:


```
typedef unsigned int UINT;           UINT is synonym for unsigned int
typedef unsigned long ULONG;        ULONG is synonym for unsigned long
```

These **typedef** statements are usually placed at the beginning of the program, but they must precede their usage. You can now declare variables of the `UINT` and `ULONG` types:

```
UINT int_var;                       int_var is of type unsigned int
ULONG long_var;                     long_var is of type unsigned long
```

Use of uppercase for synonyms is not mandatory but is recommended because you can then instantly spot the “typedef’d” data types. **typedef** is commonly used for creating synonyms to names of structures, their pointers and enumerators. The synonym can be formed when creating the template or after, as shown by the following forms shown side by side:

| | |
|--|--|
| <pre>typedef struct student { char name[30]; int dt_birth; short roll_no; short total_marks; } EXAMINEE; EXAMINEE stud1;</pre> | <pre>struct student { char name[30]; int dt_birth; short roll_no; short total_marks; } typedef struct student EXAMINEE; EXAMINEE stud1, stud2;</pre> |
| Form 1 | Form 2 |

Form 1 combines the creation of the synonym named `EXAMINEE` with the declaration of `student`, which is preceded by the keyword **typedef**. In this form, the structure tag (`student`) is optional and can be dropped. But the second form creates the synonym *after* the declaration, so the tag is necessary. `EXAMINEE` is now a replacement for `struct student`, so you can create variables of type `EXAMINEE`.



Note: **typedef** was not designed simply to replace data types with short names. It was designed to make programs portable by choosing the right data types without majorly disturbing program contents. If you need 4-byte integers, you can use **typedef int INT32;** to create a synonym in a special file and then create variables of type `INT32` in all programs. If the programs are moved to a machine where `int` uses 2 bytes, you can simply change the statement to **typedef long INT32;** without disturbing the contents of the programs. (The `long` data type uses a minimum of 4 bytes.)

14.4.3 structure_typedef.c: Simplifying Use of Structures

Program 14.3 demonstrates the use of **typedef** in abbreviating a primary data type and two structures. The structure named `film` is given the synonym `MOVIE1` at the time of declaration. An identical but separate structure named `cinema` is typedef’d to `MOVIE2` after declaration. Observe the dual use of **typedef** with `cinema`; the data type of `year` is typedef’d to `USHORT` before `cinema` is typedef’d to `MOVIE2`.

Lesson 2: More Structures

Chapter 14, Sections 14.5–14.8.1.

454 Computer Fundamentals & C Programming

```
/* structure_typedef.c: Demonstrates convenience of abbreviating
                        data types with typedef. */
#include <stdio.h>

int main(void)
{
    typedef struct film {                /* Declaration also creates synonym */
        char title[30];
        char director[30];
        unsigned short year;
    } MOVIE1;                            /* Synonym MOVIE1 created as a data type */

    MOVIE1 release1 = {"The Godfather", "Francis Ford Coppola", 1972};
    printf("release1 values: %s, %s, %hd\n",
           release1.title, release1.director, release1.year);

    typedef unsigned short USHORT;       /* Synonym USHORT created here ... */
    struct cinema {
        char title[30];
        char director[30];
        USHORT year;                    /* ... and used here */
    };


    typedef struct cinema MOVIE2;        /* Synonym created after declaration */
    MOVIE2 release2 = {"Doctor Zhivago", "David Lean", 1965};
    printf("release2 values: %s, %s, %hd\n",
           release2.title, release2.director, release2.year);

    return 0;
}
```

PROGRAM 14.3: **structure_typedef.c**

```
release1 values: The Godfather, Francis Ford Coppola, 1972
release2 values: Doctor Zhivago, David Lean, 1965
```

PROGRAM OUTPUT: **structure_typedef.c**

 **Note:** Even though film and cinema have identical members, they represent separate templates. Their variables are thus not compatible for copying operations. It's not possible to assign release1 (of type film) to release2 (of type cinema) or vice versa.

14.5 NESTED STRUCTURES

A structure member can have any data type—including another structure (but not of the same type though). The inner structure can contain another structure and so on. C supports *nested structures* and the following outer structure named student contains an inner structure named dt_birth as one of its four members:

```

struct student {
    char name[30];
    struct {
        short day;
        short month;
        short year;
    } dt_birth;
    int roll_no;
    short total_marks;
};
struct student stud1;

```

Member 1
Structure as member

Member 2
Member 3
Member 4

Here, the declaration of the inner structure forms part of the declaration of the outer one. Note that `dt_birth` is actually a variable to which its members are connected. *However, you can't separately create a structure variable of this type.* This restriction, however, doesn't apply if the two structures are declared separately. Define the inner structure before the outer one:

```

struct dob {
    short day;
    short month;
    short year;
};
struct student {
    char name[30];
    struct dob dt_birth;
    int roll_no;
    short total_marks;
};
struct student stud1;
struct dob dob1;

```

dob must be declared before ...

... it is used in student.

Encapsulating the three components of a date into a separate structure has two advantages over using separate “unstructured” members. First, you can pass this structure to a function as a single argument without losing the ability of individually accessing its members. Second, because `dob` can be used by multiple programs, its declaration could be stored in a separate file. A program that uses a date field as a three-member structure can simply include this file.

14.5.1 Initializing a Nested Structure

When it comes to initialization, nested structures resemble multi-dimensional arrays. For every level of nesting, a set of inner braces have to be used for members of the inner structure. This is how you initialize the structure of the preceding template:

```
struct stud1 = {"Oskar Barnack", {1, 11, 1879}, 3275, 555};
```

The three initializers inside the inner braces represent the values of `dob` members. The inner braces are optional but you must use them for two reasons. First, they provide clarity when associating initializers with their members. Second, they let you partially initialize a structure. For instance, you can drop the value for month or both month and year while initializing `stud1`.

14.5.2 Accessing Members

A member of an inner structure is connected by a dot to its immediately enclosing structure, which is connected by another dot to the outer structure. For a structure with a single level of nesting, the innermost member can be accessed as

outer_structure.inner_structure.member

In the present case, after student has been instantiated to form the variable stud1, month can be accessed as

stud1.dt_birth.month

scanf needs the & prefix

For every increment in the level of nesting, the number of dots increase by one. This dot-delimited notation is similar to the pathname of a file. On Windows and UNIX systems, the pathname a/b/c refers to a file named c having b as its parent directory, which in turn has a as its parent. For a structure member, the name a.b.c can be interpreted in a similar manner: a and b must be names of structures while c can never be one.

14.5.3 structure_nested.c: Program Using a Nested Structure

Program 14.4 uses a nested structure where the inner structure named dob is incorporated as a member of the outer structure named student. This data type is typedef'd to EXAMINEE and used to create two variables, stud1 and stud2. stud1 is initialized but stud2 is populated by **scanf**. Note the use of the flag 0 in the **printf** format specifier (%02hd). This flag pads a zero to the day and month members of dt_birth to maintain the 2-digit width.

```
/* structure_nested.c: Shows how to access each member of a nested structure
   using structure1.structure2.member notation. */
#include <stdio.h>
int main(void)
{
    struct dob {                                /* Must be declared before student */
        short day;
        short month;
        short year;
    };
    typedef struct student {
        char name[30];
        struct dob dt_birth;                    /* Member is a nested structure */
        int roll_no;
        short total_marks;
    } EXAMINEE;                                /* Synonym for struct student */

    EXAMINEE stud1 = {"Oskar Barnack", {1, 11, 1879}, 3275, 555};
    printf("Name: %s, DOB: %02hd/%02hd/%02hd, Roll No: %d, Marks: %hd\n",
           stud1.name, stud1.dt_birth.day, stud1.dt_birth.month,
           stud1.dt_birth.year, stud1.roll_no, stud1.total_marks);
```

```

EXAMINEE stud2;
fputs("Enter name: ", stdout);
scanf("%[^\\n]", stud2.name);          /* Possible to enter multiple words */

while (getchar() != '\\n')
    ;                                /* Clears newline from buffer */

fputs("Enter DOB (dd/mm/yyyy), roll no. and marks: ", stdout);
scanf("%2hd/%2hd/%4hd %d %hd", &stud2.dt_birth.day, &stud2.dt_birth.month,
    &stud2.dt_birth.year, &stud2.roll_no, &stud2.total_marks);

printf("Name: %s, DOB: %02hd/%02hd/%hd, Roll No: %d, Marks: %hd\\n",
    stud2.name, stud2.dt_birth.day, stud2.dt_birth.month,
    stud2.dt_birth.year, stud2.roll_no, stud2.total_marks);
return 0;
}

```

PROGRAM 14.4: **structure_nested.c**

```

Name: Oskar Barnack, DOB: 01/11/1879, Roll No: 3275, Marks: 555
Enter name: Carl Zeiss
Enter DOB (dd/mm/yyyy), roll no. and marks: 11/9/1816 5723 666
Name: Carl Zeiss, DOB: 11/09/1816, Roll No: 5723, Marks: 666

```

PROGRAM OUTPUT: **structure_nested.c**

14.6 ARRAYS AS STRUCTURE MEMBERS

An array can also be a member of a structure, which is evident from the way we used one of type char to represent the name or title in `music_cd`, `cricketer` and `student`. But structures in C support arrays of any type as shown in the following modified form of the student structure:

```

struct student {
    char name[30];
    int roll_no;
    short marks_pcb[3];          /* Member is an array of 3 elements */
};

```

Like nested structures, a variable of this type is initialized using a separate set of braces for the values related to `marks_pcb`. You are aware that the inner braces are not mandatory but they provide clarity:

```
struct student stud1 = {"Oskar Barnack", 3275, {85, 97, 99}};
```

We can access the individual elements of `marks_pcb` as `stud1.marks_pcb[0]`, `stud1.marks_pcb[1]` and so forth. These elements are handled as simple variables:

```

stud1.marks_pcb[0] = 90;
printf("Physics Marks: %hd\\n", stud1.marks_pcb[0]);
scanf("%hd", &stud1.marks_pcb[0]);

```

Could we not have used three short variables here? Yes, we could have, and added another two variables if we wanted to include five subjects. But would you like to use five variables with five statements or a single array in a loop to access all the marks?



Note: Besides providing clarity, the inner braces in the initializer segment allow the partial initialization of array elements and members of nested structures.

14.7 ARRAYS OF STRUCTURES

We used two structure variables, `stud1` and `stud2`, to handle data of two students. This technique won't work with 500 students. C supports an array of structures, whose definition may or may not be combined with the declaration of the structure. The following statement defines an array of type `struct student` that can hold 50 sets (or records) of student data:

```
struct student {
    char name[30];
    int roll_no;
    short marks;
} stud[50];
```

Memory allocated for 50 elements

Alternatively, you can separate the declaration and definition. You can also use **typedef** to replace `struct student` with `STUDENT`:

```
typedef struct student {
    char name[30];
    int roll_no;
    short marks;
} STUDENT;
STUDENT stud[50];
```

All elements now uninitialized

Because `stud` is an array, its elements are laid out contiguously in memory. The size of each array element is equal to the size of the structure in memory after accounting for slack bytes. Pointer arithmetic can easily be employed here to access each array element, and using a special notation (`->`), the members as well (14.9).

This array can be partially or fully initialized by enclosing the initializers for each array element in a set of curly braces. Each set is separated from its adjacent one by a comma, while the entire set of initializers are enclosed by outermost braces:

```
STUDENT stud[50] = {
    {"Benjamin Franklin", 1234, 90},
    {"Max Planck", 4321, 80},
    ...,
    {"Albert Einstein", 9876, 70}
};
```

Each member of each element of this array is accessed by using a dot to connect the member to its array element. For instance, the member `name` is accessed as `stud[0].name`, `stud[1].name` and so on. If you have not initialized the array, then you'll have to separately assign values in the usual manner:

```
strcpy(stud[5].name, "Emile Berliner");
stud[5].roll_no = 3333;
stud[5].marks = 75;
```

You can use **scanf** to input values to each member using pointers to these variables (like `&stud[i].marks`). A simple **for** loop using the array index as the key variable prints the entire list:

```
for (i = 0; i < 50; i++)
    printf("%s %d %hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);
```

An array of structures is akin to *records* with *fields*. In the early days of commercial data processing, an employee's details were held as individual fields in a record. The fields of each record were processed before the next record was read. The same principle now applies to an array of structures and the two programs that are taken up next clearly demonstrate this.



Tip: An array of structures can take up a lot of memory, so make sure that you set the size of the array to a realistic value. On a Linux system, **sizeof stud** evaluated to 2000 bytes, i.e., 40 bytes for each element.

14.7.1 array_of_structures.c: Program for Displaying Batting Averages

Program 14.5 features an array of structures of type `cricketer`. Each element of the array variable, `bat`, stores the batting average and number of tests played by an individual. `bat` is partially initialized with the values of two players. The third array element is assigned separately while the fourth element is populated with **scanf**. The complete list is printed using **printf** in a loop.

The program also uses the **FLUSH_BUFFER** macro for ridding the buffer of character remnants that are left behind every time **scanf** reads a string. Note the use of the `-` symbol in the format specifier of the last **printf** statement. The `-` left-justifies the name and country—the way string data should be printed.

```
/* array_of_structures.c: Demonstrates techniques of defining, initializing
                           and printing an array of structures. */
#include <stdio.h>
#include <string.h>
#define FLUSH_BUFFER while (getchar() != '\n');

int main(void)
{
    short i, imax;
    struct cricketer {
        char name[20];
        char team[15];
        short tests;
        float average;
    } bat[50] = {
        {"Don Bradman", "Australia", 52, 99.94},
        {"Greame Pollock", "South Africa", 23, 60.97}
    };

    i = 2;
```

```

strcpy(bat[i].name, "Wally Hammond");
strcpy(bat[i].team, "England");
bat[i].tests = 85; bat[i].average = 58.46f;

i++;
fputs("Enter name: ", stdout);
scanf("%s", bat[i].name);

FLUSH_BUFFER
fputs("Enter team: ", stdout);
scanf("%s", bat[i].team);

fputs("Enter tests and average: ", stdout);
scanf("%d %f", &bat[i].tests, &bat[i].average);

imax = i;
for (i = 0; i <= imax; i++)
    printf("%-20s %-15s %3hd %.2f\n",
           bat[i].name, bat[i].team, bat[i].tests, bat[i].average);
return 0;
}

```

PROGRAM 14.5: `array_of_structures.c`

```

Enter name: Sachin Tendulkar
Enter team: India
Enter tests and average: 200 53.79
Don Bradman      Australia      52  99.94
Greame Pollock   South Africa   23  60.97
Wally Hammond    England       85  58.46
Sachin Tendulkar India        200  53.79

```

PROGRAM OUTPUT: `array_of_structures.c`

14.7.2 `structure_sort.c`: Sorting an Array of Structures

Program 14.6 fully initializes an array of five structures. The structures are then sorted in descending order on the marks field, using the selection sort algorithm that has been discussed previously (10.8). The program prints the student details before and after sorting.

```

/* structure_sort.c: Sorts an array of structures on the marks field.
   Algorithm used: selection sort */
#include <stdio.h>
#define COLUMNS 5
int main(void)
{
    short i, j, imax;

```



```

struct student {
    char name[30];
    int roll_no;
    short marks;
} temp, stud[COLUMNS] = {
    {"Alexander the Great", 1234, 666},
    {"Napolean Bonaparte", 4567, 555},
    {"Otto von Bismark", 8910, 999},
    {"Maria Teresa", 2345, 777},
    {"Catherine The Great", 6789, 888}
};

printf("Before sorting ...\n");
for (i = 0; i < COLUMNS; i++)
    printf("%-20s %4d %4hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);

printf("\nAfter sorting ...\n");
imax = i;
for (i = 0; i < imax - 1; i++)          /* Selection sort algorithm ... */
    for (j = i + 1; j < imax; j++)      /* ... explained in Section 10.8 */
        if (stud[j].marks > stud[i].marks) {
            temp = stud[i];
            stud[i] = stud[j];
            stud[j] = temp;
        }

for (i = 0; i < COLUMNS; i++)
    printf("%-20s%5d%5hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);

return 0;
}

```

PROGRAM 14.6: structure_sort.c

Before sorting ...

| | | |
|---------------------|------|-----|
| Alexander the Great | 1234 | 666 |
| Napolean Bonaparte | 4567 | 555 |
| Otto von Bismark | 8910 | 999 |
| Maria Teresa | 2345 | 777 |
| Catherine The Great | 6789 | 888 |

After sorting ...

| | | |
|---------------------|------|-----|
| Otto von Bismark | 8910 | 999 |
| Catherine The Great | 6789 | 888 |
| Maria Teresa | 2345 | 777 |
| Alexander the Great | 1234 | 666 |
| Napolean Bonaparte | 4567 | 555 |

PROGRAM OUTPUT: **structure_sort.c**

It's impractical to provide large amounts of data in the program. In real-life, structure data are saved in a file, with each line (or record) representing the data of members stored in an array element. Chapter 15 examines the standard functions that read and write files and how they can be used in handling data of structures.

14.8 STRUCTURES IN FUNCTIONS

The importance of structures can be strongly felt when they are used in functions. Functions use structures in practically every conceivable way, and this versatility makes structures somewhat superior to arrays when used in functions. A structure-related data item can take on the following forms when used as a function argument:

- An element of a structure. The function copies this element which is represented in the form *structure.member*.
- The structure name. The function copies the entire structure. *It doesn't interpret the name of the structure as a pointer.* This is not the case when the name of an array is passed to a function.
- A pointer to a structure or a member. This technique is normally used for returning multiple values or avoiding the overhead of copying an entire structure inside a function.

A function can also return any of these three objects. Structures present the best of both worlds because a function can be told to either copy the entire structure or simply its pointer. The former technique is memory-intensive but safe while the latter could be unsafe if not handled with care.

We declare a function before **main** for its signature to be visible throughout the program. This means that the declaration of a structure used by a function must precede the declaration of the function, as shown in the following:

```
struct date {                                Structure declared before main ...
    short day;
    short month;
    short year;
};
void display(struct date d);                  ... and before function declaration
```

The **display** function returns nothing but it could have an implementation similar to this:

```
void display(struct date d)                  Function definition
{
    printf("Date is %hd/%hd/%hd\n", d.day, d.month, d.year);
    return;
}
```

After the date template has been instantiated, you can invoke the function inside **main** using the structure variable as argument:

```
struct date today = {29, 2, 2017};
display(today);                             Displays Date is 29/2/2017
```

Note that the **display** function copies the entire structure, **today**. However, this copy disappears after the function has returned. Can we use this copy to change the original structure? We'll soon learn that we can.



Takeaway: A function using a structure as argument copies the entire structure. If the structure contains an array, it too will be copied.



Caution: A program may run out of memory when using structures as arguments. A structure containing numerous members and large arrays can lead to stack overflow and cause the program to abort.

14.8.1 `structure_in_func.c`: An Introductory Program

Program 14.7 uses two functions that use a three-member structure named `time` as an argument. The `display_structure` function simply outputs the values of the three members. The `time_to_seconds` function returns the time after conversion to seconds. Note that we have now moved the structure declaration before `main` and before the function declarations. `time` is now a global template even though its variable `t` is not global.

```
/* structure_in_func.c: Use a structure as a function argument. */
#include <stdio.h>

struct time {
    short hour;
    short min;
    short sec;
};

void display_structure(struct time f_time);
int time_to_seconds(struct time f_time);
int main(void)
{
    struct time t;
    fputs("Enter a time in hh:mm:ss format: ", stdout);
    scanf("%hd:%hd:%hd", &t.hour, &t.min, &t.sec);
    display_structure(t);
    int value = time_to_seconds(t);
    printf("Value of t in seconds: %d\n", value);
    return 0;
}

void display_structure(struct time f_time)
{
    printf("Hours: %hd, Minutes: %hd, Seconds: %hd\n",
        f_time.hour, f_time.min, f_time.sec);
    return;
}

int time_to_seconds(struct time f_time)
{
    return f_time.hour * 60 * 60 + f_time.min * 60 + f_time.sec;
}
```

PROGRAM 14.7: `structure_in_func.c`

Lesson 3: Example Cases

Chapter 14, Sections 14.8.2–14.8.3.

464

Computer Fundamentals & C Programming

```
Enter a time in hh:mm:ss format: 9:15:45
Hours: 9, Minutes: 15, Seconds: 45
Value of t in seconds: 33345
```

PROGRAM OUTPUT: `structure_in_func.c`

14.8.2 `time_difference.c`: Using a Function that Returns a Structure

A function can also return a structure which often has the same type that is passed to it. In the following declaration, the `time_diff` function accepts two arguments of type `struct time` and also returns a value of the same type:

```
struct time time_diff(struct time t1, struct time t2);
```

Program 14.8 computes the difference between two times that are input to `scanf` in the form `hh:mm:ss`. The values are saved in two structure variables, `t1` and `t2`, that are typedef'd to `TIME`. A third variable, `t3`, stores the returned value. Note that the `mins` operand is shared by the equal-priority operators, `--` and `dot`, but because of L-R associativity, no parentheses are needed.

```
/* time_difference.c: Uses a function to compute and return
the difference between two times as a structure. */
#include <stdio.h>

typedef struct time {
    short hours;
    short mins;
    short secs;
} TIME;

TIME time_diff(TIME t1, TIME t2);          /* Function returns a structure */

int main(void)
{
    TIME t1, t2, t3;

    fputs("Enter start time in hh:mm:ss format: ", stdout);
    scanf("%hd:%hd:%hd", &t1.hours, &t1.mins, &t1.secs);

    fputs("Enter stop time in hh:mm:ss format: ", stdout);
    scanf("%hd:%hd:%hd", &t2.hours, &t2.mins, &t2.secs);

    t3 = time_diff(t1, t2);
    printf("Difference: %hd hours, %hd minutes, %hd seconds\n",
           t3.hours, t3.mins, t3.secs);
    return 0;
}
```

```

TIME time_diff(TIME t1, TIME t2)
{
    TIME diff;
    if (t1.secs > t2.secs) {
        t2.mins--;
        t2.secs += 60;
    }
    if (t1.mins > t2.mins) {
        t2.hours--;
        t2.mins += 60;
    }
    diff.secs = t2.secs - t1.secs;
    diff.mins = t2.mins - t1.mins;
    diff.hours = t2.hours - t1.hours;
    return diff;
}

```

PROGRAM 14.8: `time_difference.c`

```

Enter start time in hh:mm:ss format: 6:45:50
Enter stop time in hh:mm:ss format: 8:35:55
Difference: 1 hours, 50 minutes, 5 seconds

```

PROGRAM OUTPUT: `time_difference.c`

Since `t3` now has three new values, it can be said that `time_diff` has “returned” three values. This property of structures breaks the well-known maxim that a function can return a single value using the **return** statement. We’ll use this property to provide an alternative solution to the swapping problem.



Note: The dot has the same precedence as the increment and decrement operators, but it has L-R associativity. Thus, in the expression `t2.mins--`, the dot operates on `mins` before the `--` does, so we don’t need to use `(t2.mins)--` in the program.

14.8.3 `swap_success2.c`: Swapping Variables Revisited

The programs, `swap_failure.c` (Program 11.3) and `swap_success.c` (Program 12.5) taught us an important lesson: a function can interchange the values of two variables only by using their pointers as arguments. What if these two variables are members of a structure? Program 14.9 proves that it is possible to swap them without using pointers. The game-changer in this program is the statement, `s = swap(s);`, which merits some discussion.

The `swap` function here accepts and returns a two-member structure of type `two_var`. This function swaps the copies of `s.x` and `s.y` in the usual manner but it also returns a copy of the structure. The swapping operation succeeds here because this returned value is assigned to the same variable that was passed as argument (`s = swap(s);`). Using this technique, it is possible for a function to “return” multiple values without using pointers.

```
/* swap_success2.c: Successfully swaps two variables using a structure
and without using a pointer. */
#include <stdio.h>
struct two_var {
    short x;
    short y;
} s = { 1, 10};
struct two_var swap(struct two_var);
int main(void)
{
    printf("Before swap: s.x = %hd, s.y = %hd\n", s.x, s.y);
    s = swap(s); /* The game changer */
    printf("After swap: s.x = %hd, s.y = %hd\n", s.x, s.y);
    return 0;
}
struct two_var swap(struct two_var z)
{
    short temp;
    temp = z.x;    z.x = z.y;    z.y = temp;
    return z;
}
```

PROGRAM 14.9: swap_success2.c

Before swap: s.x = 1, s.y = 10
After swap: s.x = 10, s.y = 1

PROGRAM OUTPUT: **swap_success2.c**



Takeaway: The assignment property (=) of structures, because of which one structure variable can be assigned to another, allows a function to change the original values of structure members using their copies.