STRUCTURES

Sercan Külcü | 21.06.2023

Contents

1 Structure declaration and initialization

In this chapter, we will dive into the fascinating realm of structure declaration and initialization. Structures provide a way to group related data items together under one name, making our programs more organized and efficient. So, let's get started and explore the wonders of structures!

In C programming, a structure is a user-defined data type that allows you to combine different types of variables under a single name. It provides a convenient way to represent complex entities by grouping related data together. Structures enable you to create custom data types that suit your specific programming needs, providing flexibility and clarity to your code.

To declare a structure, you need to define its blueprint using the struct keyword followed by a name of your choice. Inside the structure, you can declare multiple variables of different data types. The general syntax for declaring a structure is as follows:

```
struct structureName {
   dataType member1;
   dataType member2;
   // ...
   dataType memberN;
};
```

Here, structureName represents the name of the structure, and member1, member2, ..., memberN are the variables within the structure, each having its own data type. For example, let's consider a structure to represent a point in 2D space:

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

In this case, we have declared a structure named Point with two members x and y, both of type int.

Once you have declared a structure, you can initialize its variables using various techniques. Let's explore some of the commonly used methods:

1. Member-wise Initialization

Member-wise initialization involves assigning values to the structure members individually. You can use the dot (.) operator to access the members and assign values to them. Here's an example:

```
struct Point p1;

p1.x = 10;

p1.y = 5;
```

In this code snippet, we declare a structure variable p1 of type Point and assign values to its x and y members.

2. Initialization during Declaration

You can also initialize a structure during declaration by enclosing the member values within curly braces {}. Here's an example:

```
struct Point p2 = {20, 15};
```

In this case, we declare a structure variable p2 of type Point and initialize its x and y members with the values 20 and 15, respectively.

3. Designated Initializers (C99 and later)

C99 introduced designated initializers, which allow you to specify values for specific members of a structure, even in a different order from their declaration. This technique can be handy when dealing with large structures. Here's an example:

```
struct Point p3 = {.y = 7, .x = 12};
```

In this code snippet, we declare a structure variable p3 of type Point and initialize its y and x members with the values 7 and 12, respectively.

2 Accessing structure members

Now it's time to learn how to access the members of a structure. In this chapter, we will explore various techniques to access and manipulate structure members, allowing you to unleash the full potential of structures in your C programs. So, let's dive right in and discover the world of accessing structure members!

2.1 Dot Operator

To access the members of a structure, we use the dot (.) operator. The dot operator acts as a bridge between the structure variable and its members, allowing us to read or modify their values. The general syntax for accessing structure members is as follows:

```
structureVariable.memberName
```

Here, structureVariable represents the name of the structure variable, and memberName refers to the name of the member you want to access. For example, let's consider the previously defined Point structure:

```
struct Point {

   int x;

   int y;

};
```

To access the members x and y, you can use the dot operator as shown below:

```
struct Point p;

p.x = 10;

p.y = 5;
```

In this code snippet, we declare a structure variable p of type Point and assign values to its x and y members using the dot operator.

2.2 Structure Member Access within Functions

When working with functions, you can access structure members using pointers. This allows you to pass structure variables to functions and manipulate their members directly. Let's explore this concept through an example:

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

void printPoint(struct Point* p) {
    printf("Coordinates: (%d, %d)\n", p->x, p->y);
}

int main() {
    struct Point p;
    p.x = 10;
    p.y = 5;

    printPoint(&p);

    return 0;
}
```

In this code snippet, we define a function printPoint that takes a pointer to a Point structure as its parameter. By using the arrow (->) operator instead of the dot operator, we can access the members of the structure pointer passed to the function.

The arrow operator acts as a shorthand for dereferencing the pointer and accessing the member simultaneously.

2.3 Nested Structures

Structures can also be nested within other structures, allowing you to create complex data structures. In such cases, accessing nested structure members requires multiple dot operators. Let's consider an example involving nested structures:

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

struct Employee {
    char name[50];
    struct Date dateOfBirth;
};

int main() {
    struct Employee emp;
    emp.dateOfBirth.day = 10;
    emp.dateOfBirth.month = 5;
    emp.dateOfBirth.year = 1990;

    printf("Employee Date of Birth: %d/%d/%d\n", emp.dateOfBirth.day,
emp.dateOfBirth.month, emp.dateOfBirth.year);
```

```
    return 0;

}
```

In this code snippet, we define a structure Date representing a date and a structure Employee containing the name and date of birth of an employee. To access the date of birth within the Employee structure, we use the dot operator twice.

3 Nested structures and arrays of structures

In this chapter, we will explore the fascinating world of nested structures and arrays of structures. These advanced concepts allow you to create even more complex and dynamic data structures, opening up exciting possibilities for your programs. So, let's dive right in and discover the wonders of nested structures and arrays of structures!

3.1 Nested Structures

Nested structures refer to the concept of including one structure within another. By nesting structures, you can create hierarchical relationships and build more intricate data structures. Let's delve into an example to understand nested structures better:

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

struct Employee {
    char name[50];
    struct Date dateOfBirth;
};

int main() {
    struct Employee emp;
    emp.dateOfBirth.day = 10;
    emp.dateOfBirth.month = 5;

    emp.dateOfBirth.year = 1990;


    printf("Employee Date of Birth: %d/%d/%d\n", emp.dateOfBirth.day, emp.dateOfBirth.month, emp.dateOfBirth.year);


    return 0;

}
```

In this code snippet, we define two structures: Date representing a date and Employee representing an employee. The Employee structure includes a member of type Date called dateOfBirth. By nesting the Date structure within the Employee structure, we can access the date of birth of an employee as a sub-member.

3.2 Arrays of Structures

Arrays of structures provide a powerful mechanism to store multiple instances of structures. This allows you to work with collections of related data efficiently. Let's explore an example to illustrate arrays of structures:

```
struct Student {

    char name[50];

    int rollNumber;

};


int main() {

    struct Student class[3];


    for (int i = 0; i < 3; i++) {
        printf("Enter the name of student %d: ", i + 1);
        scanf("%s", class[i].name);

        printf("Enter the roll number of student %d: ", i + 1);
        scanf("%d", &class[i].rollNumber);
    }

    printf("Student Details:\n");
    for (int i = 0; i < 3; i++) {
        printf("Name: %s, Roll Number: %d\n", class[i].name, class[i].rollNumber);
    }

    return 0;
}
```

In this code snippet, we define a structure Student representing a student with a name and a roll number. We create an array of Student structures called class, which can hold up to three students. We use a loop to input the names and roll numbers of the students and then display their details using another loop.

3.3 Nested Structures and Arrays of Structures

Combining nested structures with arrays of structures allows you to create complex data structures that mimic real-world scenarios. You can nest structures within other structures and create arrays of those nested structures. Let's consider an example to illustrate this concept:

```
struct Course {
    char name[50];
    int credits;
};

struct Student {
    char name[50];
    int rollNumber;
    struct Course courses[3];
};

int main() {
    struct Student student;
    strcpy(student.name, "John Doe");
    student.rollNumber = 12345;

    strcpy(student.courses[0].name, "Mathematics");
    student.courses[0].credits = 3;

    strcpy(student.courses[1].name, "Physics");
    student.courses[1].credits = 4;

    printf("Student Details:\n");
    printf("Name: %s, Roll Number: %d\n", student.name, student.rollNumber);
    printf("Course 1: %s, Credits: %d\n", student.courses[0].name,
student.courses[0].credits);
    printf("Course 2: %s, Credits: %d\n", student.courses[1].name,
student.courses[1].credits);
```

```
    return 0;
}
```

In this code snippet, we define a structure Course representing a course with a name and credits. We then nest the Course structure within the Student structure. The Student structure includes an array of Course structures called courses, allowing us to store information about multiple courses for a student.