Structs in C

Md. Aminul Islam Shazid

26 Nov 2025

Outline

1. Introduction
2. Defining Types with `typedef`
3. Pointers to Structs

Introduction

Motivation

Many real-world things involve multiple pieces of information of different types. Examples:

- A student has a name, ID, and CGPA
- A point has x and y coordinates
- A book has a title, author, and price

Arrays cannot store such heterogeneous data, so C provides structures to group diverse data elements.

Introduction

- A structure (also known as struct) is a user-defined composite data type
- It bundles several variables (called members) into a single logical unit
- A structure definition:
  - Specifies the names and types of fields
  - Describes a layout template
  - Does not allocate memory by itself
  - Memory is allocated only when structure variables are declared

Defining a Structure

```c
struct Student {
    char name[50];
    int id;
    float cgpa;
};
```

This creates a new type layout called `struct Student`.

Declaring Structure Variables

Declaring structure variables allocates memory.
Examples:

```
struct Student s;
struct Student t, class_monitor;
```

Accessing Members (Dot Operator)

```
s.id = 101;
s.cgpa = 3.8;
```

The dot operator accesses fields inside a structure variable.

Initialization Examples

```
struct Student a = { "Alice", 1001, 3.90 };

struct Student b = {
    .name = "Bob",
    .id = 1002,
    .cgpa = 3.75
};
```

Can also keep one or more fields uninitialzied:

```
struct Student c = {
    .id = 1003,
    .cgpa = 3.80
};
```

Arrays of Structs

```c
struct Student class[50];

class[0].name = "Alice";
class[0].id = 1001;
class[0].cgpa = 3.90;
```

Useful for records, lists, and tables.

Nested Structures

A structure may contain another structure as a member:

```
struct Address {
    char city[30];
    int zip;
};

struct Person {
    char name[50];
    struct Address home;
};
```

Defining Types with `typedef`

The `typedef` Keyword

- A structure type name often becomes long and repetitive
- Using `typedef` helps create cleaner and shorter type names
- Without typedef: `struct Student s;`
- With typedef: `Student s;`

This improves readability, especially in function prototypes and pointer-heavy code.

typedef Example

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

Date today;
today.day = 18;
```

When typedef is Helpful

- When the struct type is used frequently
- When building abstract data types
- When designing libraries or APIs
- When working with pointers to structs

When not mandatory:

- Very simple, rarely used struct types
- When emphasizing that a type is a struct (some coding styles prefer this)

Pointers to Structs

Struct Pointers

A pointer to a struct is used when:

- Passing to functions efficiently
- Modifying the original struct
- Allocating structs dynamically
- Implementing dynamic data structures (linked lists, trees)

Pointer to a Struct

```c
struct Student s, *p;

p = &s;
p->cgpa = 3.70;    // pointer access
```

Dot vs Arrow

- Dot: use with actual struct variables
- Arrow: use with struct pointers

```
(*p).id = 20;
p->id = 20;   // preferred form when using with pointers
```

Passing Structs to Functions

Pass by value:

```
void show(struct Student s);
```

Pass by pointer (preferred for modification):

```
void update(struct Student *s) {
    s->cgpa += 0.1;
}
```

Returning a Struct

```c
typedef struct { int x, y; } Point;

Point make_point(int a, int b) {
    Point p = {a, b};
    return p;
}
```

Useful for small structures like coordinates, dates, and configuration options.

Using const with Struct Pointers

```c
void print_student(const struct Student *s);
```

Declares that the function will not modify the struct. Good practice for safety and clear intent.

Example: Basic Student Struct

Example: Pointer to Struct

Example: Array of Structs

Example: Nested Struct Example

Example: Passing Struct by Pointer

Example: Returning a Struct

Example: Using typedef with Struct