Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #10
User Defined Type – Structures in C

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to

Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)

Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: User Defined Type – Structures in C

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

A structure in C is a **user-defined data type** that allows **grouping variables of different types under a single name.** It helps in **creating complex and meaningful data types** by combining related data elements. Using structures, you can conveniently refer to a **collection of diverse but related items through one identifier**. This makes them **especially useful when multiple pieces of data need to be organized together**—such as storing records from a database or maintaining contact details in an address book.

.

Why Do We Need Structures in C?

Let’s understand the need for structures through the following program. Suppose we want to store the **roll number, total marks, and names of 20 students**.

Coding Example_1: Here the approach is to use separate arrays for each data item

```
char names[20][15];
int marks[20];
int roll_num[20];
for(int i = 0; i < 20; i++) {
   printf("Enter Roll Number, Marks, and Name:\n");
   scanf("%d", &roll_num[i]);
   scanf("%d", &marks[i]);
   scanf(" %[^\n]s", names[i]);  // Note the space before % to consume newline
}
```

Although we have stored the data, there is **no built-in connection between a student's roll number, marks, and name.** We're simply assuming that the i-th entry in all three arrays belongs to the same student. This **implicit relationship is error-prone and hard to manage.** To **explicitly group related information** and ensure that all details belong to the same student record, we use structures. They allow us to **bind related variables of different types together**, making our code more **logical, readable, and maintainable**. This **grouping of related attributes into one logical entity** is what we call a structure. When such a type is created, it **also determines the binary layout in memory, including the order of fields and the total size of any variable declared using that type**.

Key Characteristics of Structures in C

- Structures are **user-defined** (non-primitive/secondary) data types.
- They consist of **one or more named components**, commonly referred to as **data members**.
- The data members can be of **homogeneous** (same) or **heterogeneous** (different) types.
- The **memory size** of a structure is typically **at least** the total of its members' sizes, but it may vary depending on **compiler-specific alignment and padding rules**. The **offsets are determined at compile time**.
- **Structure members do not occupy memory** on their own; memory is allocated only when a **structure variable** is declared.
- **Compatible structures** (i.e., with the same layout and member types) can be **assigned directly** to each other.

NOTE: To define a new type/entity, use the **keyword – struct**

The format for declaring a structure is as below:

```
struct Tag
{
        Data_type member1;
        data_type member2;
        …..
        data_type membern;
};      // Don't forget semicolon here
```

where Tag is the name of the entire structure and Members are the elements within the struct.

Example: User defined type Student entity is created.

```
struct Student
{
        int roll_no;
        char name[20];
        int marks;
};//No memory allocation for declaration/description of the structure.
```

Accessing members of a structure

All the data members inside a structure are accessible to the functions defined outside the structure. To access the data members in a function including main, we need to create a structure variable. We can **access the members of a structure only when we create instance variables of that type.** If struct Student is the type, we can create the instance variable as:

struct Student s1; // s1 is the instance variable of type struct Student. **Now memory gets allocated for data members**

struct Student* s2; // s2 is the instance variable of type struct Student*. Now memory gets allocated for pointer. **s2 is pointer to structure**

Two operators used.

1. Dot operator (.)

Any member of a structure can be accessed using the structure variable as:

structure_variable_name.member_name

Suppose, s1 is the structure variable name and we want to access roll_no member of s1. Then, it can be accessed as**:** **s1.roll_no**

2. Arrow operator (->)

Any member of a structure can be accessed using the pointer to a structure as:

pointer_variable->member_name

Suppose, s2 is the pointer to structure variable and we want to access roll_no member of s2. Then, it can be accessed as: **s2->roll_no**

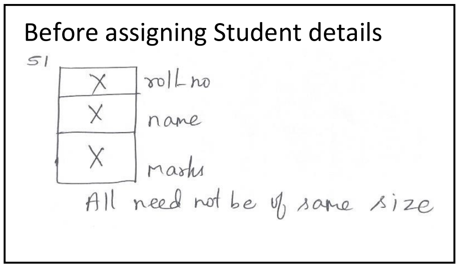
Coding Example_2: Basic Declaration, Assignment and Display. Use this structure for all programming examples listed below.

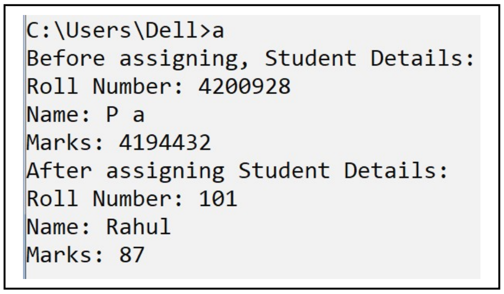
```
struct student   // Template declaration: defines what a 'student' looks like
{
   int roll_no;       // Member 1: roll number
   char name[20];     // Member 2: name (character array)
   int marks;         // Member 3: marks
};
```

Note: Declaration does not allocate memory. Only defines the blueprint of the type.

```
#include <stdio.h>
int main()
{
    struct student s1;  // Declare a structure variable of type 'student'
    printf("Before assigning, Student Details:\n");
    printf("Roll Number: %d\n", s1.roll_no);
    printf("Name: %s\n", s1.name);
    printf("Marks: %d\n", s1.marks);

    // Assigning values to members
    s1.roll_no = 101;
    s1.marks = 87;
    strcpy(s1.name, "Rahul");  // Use string handling function to assign name
    // Displaying values
    printf("After assigning Student Details:\n");
    printf("Roll Number: %d\n", s1.roll_no);
    printf("Name: %s\n", s1.name);
    printf("Marks: %d\n", s1.marks);
    return 0;
}
```

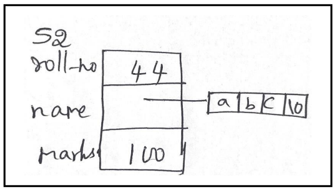

Before assigning Student details



```
C:\Users\Dell>a
Before assigning, Student Details:
Roll Number: 4200928
Name: P a
Marks: 4194432
After assigning Student Details:
Roll Number: 101
Name: Rahul
Marks: 87
```

Coding Example_3: Initialization of the structure while defining it

```
struct student s2 = {44, "abc", 100};  // Initialization
printf("%d %d %s", s2.marks, s2.roll_no, s2.name);
// 100 44 abc
```

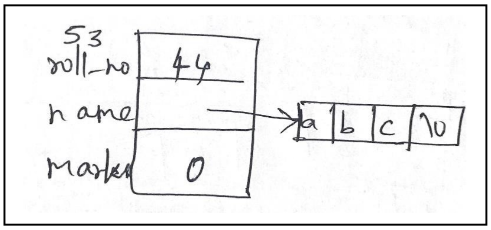


Partial Initialization of structures

If you initialize the given structure partially, **only the specified members get values**, and the rest are **automatically initialized to zero** (or null for pointers/arrays).

Coding Example_4: Partial Initialization of Structures

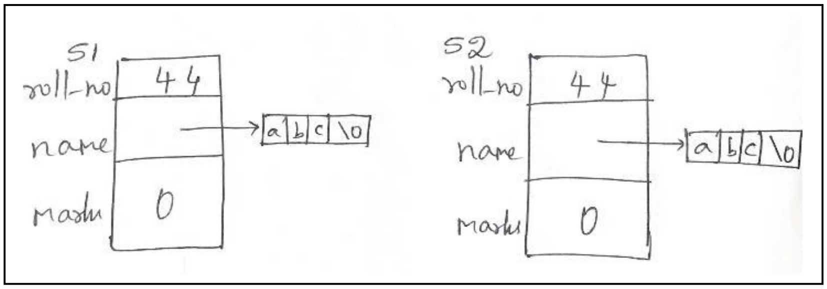
```
struct student s3 = {44, "abc"};          // Partial Initialization
// Explicitly few members are initialized. others are initialized to default value
printf("%d %d %s", s3.marks, s3.roll_no, s3.name); // 0 44  abc
```



Designated initializers allow you to **specify values for specific members** by name, **in any order**. Very useful in large structures with many fields. It requires **C99 or later C standard. Not supported** in old C standards like C89 unless enabled with compiler extensions (e.g., -std=c99 in GCC). Specifythe name of a field to initialize with '.member_name =' or 'member_name:' before the element value. Others are initialized to default value.

Coding Example_5: Designated initializers in C

```
struct student s1 = {.name = "abc", .roll_no = 44};
struct student s2 = {name : "abc", roll_no : 44};
printf("%d %d %s\n", s1.roll_no, s1.marks, s1.name);         // 44 0 abc
printf("%d %d %s", s2.roll_no, s2.marks, s2.name); // 44 0 abc
```



Memory Allocation for Structure Variable

Every data type in C has alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

More details, Refer to below links.

https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data- packing/

https://www.geeksforgeeks.org/data-structure-alignment/

NOTE:

- What is the minimum number of bytes allocated for the structure variable in each of these cases? – Sum of the sizes of all the data members. Size of data members is implementation specific.
- What is the maximum number of bytes allocated for the structure variable in each of these cases? **Implementation Specific.**

Coding Example_6: Below programs are run on Windows7 machine and 32 bit GCC compiler

```
struct test{     int i;      char j;       };
struct test1{    char j;       int i;       };
struct test2{      char k;     char j;    int i;     };
struct test3{    int i;       char k;      int j;     };
int main()
{        printf("size of the structure is %lu\n",sizeof(struct test));    //8bytes      4+4
         struct test t;
         printf("size of the structure is %lu\n",sizeof(t));        // 8bytes          4+4
         printf("size of the structure is %lu\n",sizeof(struct test1));  //8bytes            4+4
         printf("size of the       structure is %lu\n",sizeof(struct test2));  //8bytes      4+4
         printf("size of the structure is %lu\n",sizeof(struct test3));  //12bytes          4+4+4
         return 0;
}
```

#pragma in Struct Memory Management

#pragma is a **preprocessor directive** used to provide special instructions to the compiler. It's behavior is **compiler-specific.** This directive can be used with structures to **control memory alignment and padding.** By default, compilers may insert padding between structure members to align data in memory for efficient access. However, in some cases—like memory-mapped hardware, binary file structures, or network packets—you need precise control over the layout of the structure. **The #pragma pack() directive allows you to specify byte alignment to ensure that the compiler does not add unnecessary padding, or aligns members** in a specified way. The **#pragma pack(n)** directive in C is used to control **alignment** by specifying a **byte boundary** (n). This tells the compiler to align structure members on addresses that are multiples of n. The value **n must be a power of 2**: 1, 2, 4, 8, 16, 32... (up to the compiler's supported maximum).

Coding Example_7:

Version_1: Without #pragma, normally, the compiler adds padding between char a and int b to align b on a 4-byte boundary.

```
#include <stdio.h>
struct NormalStruct {
  char a;     // 1 byte                 // 3 bytes padding
  int b;     // 4 bytes (usually aligned at 4-byte boundary)
};
int main() {
  struct NormalStruct ns;
  printf("Size of NormalStruct: %zu bytes\n", sizeof(ns));   // 8 bytes
  return 0;
}
```

Version_2: With #pragma pack(1) -> Disables padding, so the structure becomes tightly packed.

```
#include <stdio.h>
#pragma pack(1)
struct PackedStruct {
  char a;    // 1 byte
```

```
    int b;      // 4 bytes (no padding!)
};
#include<stdio.h>
#pragma pack()
int main() {
    struct PackedStruct ps;
    printf("Size of PackedStruct: %lu bytes\n", sizeof(ps)); // 5 bytes
    return 0;
}
```

Comparison of Structure variables

In C, you **cannot directly compare two structure variables using** == or != like you do with primitive types. **Always prefer field-by-field comparison for accuracy and safety**. memcmp() should be used with caution due to padding or non-deterministic behavior.

Coding Example_8: Field-by-field comparison of data members of structure variables

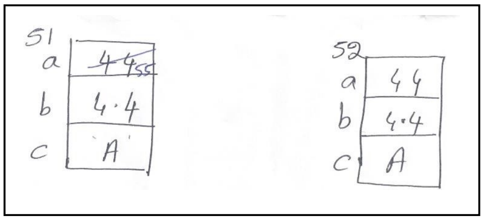
```
int main() {
    struct Student s1 = {101, "sindhu", 85};
    struct Student s2 = {101, "sindhu", 85};
    /*if (s1 == s2) {  // Error: invalid operands to binary ==
    // ...
    } */
    if (s1.roll_no == s2.roll_no &&  strcmp(s1.name, s2.name) == 0 && s1.marks == s2.marks)
    {
        printf("The structures are equal.\n");
    }
   else
   {     printf("The structures are different.\n");   }
   return 0;
}
```

Member - wise copy

Structures of **same type are assignment compatible**. When you assign one structure variable to another structure variable of same type, member- wise copy happens. **Both of them do not point to the same memory location**. The **compiler generates the code for member – wise copy.**

Coding Example_9:

```
struct testing {          int a; float b; char c;       };
int main(){
        struct testing s1 = {44,4.4, A'};
        struct testing s2 = s1;
        s1.a = 55;
        printf("%d\n",s1.a);
        printf("%d\n",s2.a);
        return 0;
    }
```



Few points to think:

- Structures can be assigned even if the structure contains an array. Is it True?
- If the structure contains pointer, how member-wise copy happens?

Usage of typedef

The typedef is a keyword which is used to **give a type, a new name**. By convention, uppercase letters are used for these definitions to remind the programmer that the type name is really a symbolic abbreviation, but we can use lowercase as well. Also, used to **assign alternative names to existing data types.** It is used to provide an interface for the client. Once a new name is created using typedef, the client may use this without knowing the underlying type. It is limited to giving symbolic names to types only. The typedef interpretation is performed by compiler.

I want to store the age of a person. Best way to define a variable is int age = 80;

Alternatively, you can also say, **typedef int age;** //another name for int is age.

Then **age a = 80;**

Can we say age age = 80?   // Think about this.

If we know how to declare a variable, then prefixing the declaration with typedef makes that a new name.

```
        int b[10];        // b is an array variable to store 10 integers
        typedef int c[10];        // c is a name for an array of 10 integers
```

NOTE: typedef is mostly used with user defined data types when names of the data types become slightly longer and complicated to use in programs.

Coding Example_10:

Version 1:

```
struct player
{        int id;   char name[20];   };
typedef struct player player_t;        // player_t is a new name to struct player.
int main()
{       player_t p1;       // p1 is a variable
}
```

Version 2: You can include typedef when defining a new type itself.

```
typedef struct player
{        int id;  char name[20];  } player_t;      // player_t is a new name to struct player.
int main()
{        player_t p1;     // p1 is a variable
}
```

Nested Structures in C

A nested structure is a **structure that contains another structure as a member**. This allows grouping related data hierarchically, improving organization and code clarity for complex data. N**ested structures** can be used in a few different **styles**, depending on how and where the inner structure is defined. Broadly, there are **3 common types** of nested structures.

A. Named Structure inside another (Standard Nested Structure)

One named structure is used as a member of another. Advantage is to reuse the structure in multiple outer structures.

Coding Example_11:

```
struct Date {    int day, month, year;    };
struct Student {
   int roll_no;
   struct Date dob;  // nested named structure
};
int main()
{          struct Student s1;
           s1.roll_no = 91;
           s1.dob.day = 31;
           s1.dob.month = 1;
           s1.dob. year = 2025;                              // Output: 91 ---  31/1/2025
           printf("%d --- %d/%d/%d", s1.roll_no, s1.dob.day, s1.dob.month, s1.dob.year);
           return 0;
}
```

B. Anonymous (Unnamed) Structure inside Another

The **inner structure is defined directly inside the outer structure without a name.** Good for one-time use. But not reusable outside the structure.

Coding Example_12:

```
struct Student {
   int roll_no;
   struct {
      int day, month, year;
   } dob;  // anonymous structure
};
int main()
{          struct Student s1;
```

```
        s1.roll_no = 91;
        s1.dob.day = 31;
        s1.dob.month = 1;
        s1.dob. year = 2025;                         // Output: 91 ---  31/1/2025
        printf("%d --- %d/%d/%d", s1.roll_no, s1.dob.day, s1.dob.month, s1.dob.year);
        return 0;
}
```

C. Structure within Structure as Pointer (Nested via Pointer)

Instead of embedding the structure directly, a pointer to a structure is used. **This allows dynamic memory allocation for the nested part.** Very useful in memory-optimized or linked data structures.

Coding Example_13:

```
struct Date {
   int day, month, year;    };
struct Student {
   int roll_no;    struct Date *dob;  // pointer to nested structure
};
int main()
{
        struct Student s1;
        s1.roll_no = 91;
        s1.dob = malloc(sizeof(struct Date));
        (s1.dob)->day = 31;
        (s1.dob)->month = 1;
        (s1.dob)->year = 2025;                  //Output: 91 ---  31/1/2025
        printf("%d --- %d/%d/%d", s1.roll_no, (s1.dob)->day, (s1.dob)->month, (s1.dob)->year);
        return 0;
}
```

Happy Coding with Structures!!!