

UNIT 10

Structure

(5 Hours) — C Programming (CACS151), BCA I/II

10.1 Introduction

An array can store multiple values, but only of the same data type. In real life, however, related data often has different types — for example, a student's roll number (int), name (string), and percentage (float). A structure is a user-defined data type in C that groups together variables of different data types under a single name, allowing them to be treated as one logical unit.

Structures are defined using the struct keyword and are especially useful for representing real-world records — such as a student, an employee, a book, or a point in 2D space — where multiple attributes must be handled together.

Memory Layout of a Structure Variable

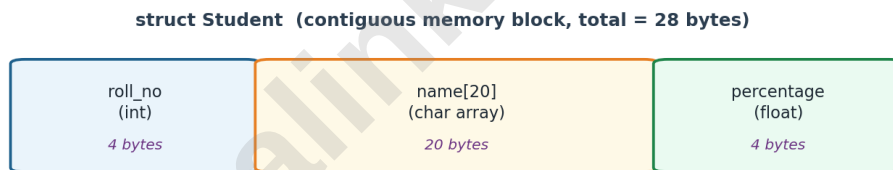


Fig 10.1 — Memory layout of a structure variable

Note: Unlike an array, a structure's members can be of completely different data types, but they are still stored in one contiguous block of memory, member after member.

10.2 Declaration

Declaring a structure involves two steps: first defining its template (blueprint), and then creating actual structure variables based on that template.

Syntax — Structure Template

```
struct StructureName
{
    data_type member1;
    data_type member2;
    ...
};
```

Example 10.1 — Declaring a Structure

```
struct_decl.c
struct Student
{
    int roll_no;
    char name[20];
    float percentage;
};
int main()
{
    struct Student s1, s2; /* declaring structure variables */
    return 0;
}
```

Structure variables can also be declared at the same time as the template, and a typedef can be used to create a shorter alias for the structure type, avoiding the need to repeat the struct keyword.

Alternative Declaration Styles

```
/* Declaring variables along with the template */
struct Student
{
    int roll_no;
    char name[20];
} s1, s2;
/* Using typedef for a shorter alias */
typedef struct Student
{
    int roll_no;
    char name[20];
} Student;
Student s3; /* 'struct' keyword no longer needed */
```

Accessing a member of a structure variable requires the dot (.) operator, written as: `variable_name.member_name`

10.3 Initialization

A structure variable can be given initial values at the time of declaration, similar to arrays, by listing the values in the same order as the members inside curly braces.

Example 10.2 — Initializing and Accessing a Structure

```
struct_init.c
#include <stdio.h>
struct Student
{
    int roll_no;
    char name[20];
    float percentage;
};
int main()
{
    struct Student s1 = {101, "Aarav", 88.5};
    printf("Roll No: %d\n", s1.roll_no);
    printf("Name: %s\n", s1.name);
    printf("Percentage: %.2f", s1.percentage);
    return 0;
}
```

Output

Roll No: 101

Name: Aarav

Percentage: 88.50

Individual members can also be assigned values after declaration, one at a time, using the dot operator:

Member-wise Assignment

```
struct Student s2;
s2.roll_no = 102;
strcpy(s2.name, "Bina"); /* strings need strcpy(), not = */
s2.percentage = 91.25;
```

Note: A member that is a character array (string) cannot be assigned directly with = after declaration — the strcpy() function from string.h must be used instead, since C does not allow array assignment with =.

10.4 Nested Structure

A nested structure occurs when one structure is declared as a member inside another structure. This is useful for grouping related sub-records — for example, an employee's date of birth is itself a logical group of day, month, and year, so it can be modeled as its own structure and embedded inside the Employee structure.

Nested Structure — struct Date inside struct Employee

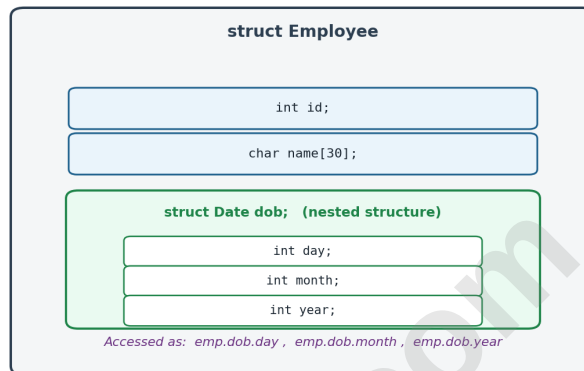


Fig 10.2 — A nested structure (struct Date inside struct Employee)

Example 10.3 — Nested Structure

```
nested_struct.c
#include <stdio.h>
struct Date
{
    int day, month, year;
};
struct Employee
{
    int id;
    char name[30];
    struct Date dob;    /* nested structure */
};
int main()
{
    struct Employee emp = {1, "Kiran", {15, 8, 1998}};
    printf("Name: %s\n", emp.name);
    printf("DOB: %d/%d/%d", emp.dob.day, emp.dob.month, emp.dob.year);
    return 0;
}
```

Note: Members of a nested structure are accessed by chaining the dot operator: `outer_variable.inner_member.inner_inner_member` — e.g. `emp.dob.day`.

10.5 Array of Structure

Just as an array can hold multiple values of a basic data type, an array of structures can hold multiple records of the same structure type — for example, storing data for an entire class of students using a single array variable.

Example 10.4 — Array of Structures

```
array_of_struct.c
#include <stdio.h>
struct Student
{
    int roll_no;
    char name[20];
};
int main()
{
    struct Student class[3] = {
        {1, "Aarav"},
        {2, "Bina"},
        {3, "Chetan"}
    };
    int i;
    for (i = 0; i < 3; i++)
        printf("Roll %d: %s\n", class[i].roll_no, class[i].name);
    return 0;
}
```

Output

Roll 1: Aarav

Roll 2: Bina

Roll 3: Chetan

Note: An array of structures is declared exactly like a normal array — `struct Student class[3]` — and each element `class[i]` is a complete structure whose members are accessed with `class[i].member`.

10.6 Array Within Structure

The reverse relationship is also possible: an array can be a member inside a structure, letting a single record hold a whole list of related values — for example, storing marks in five subjects for one student.

Example 10.5 — Array as a Structure Member

array_within_struct.c

```
#include <stdio.h>
struct Student
{
    char name[20];
    int marks[5];      /* array WITHIN the structure */
};
int main()
{
    struct Student s1 = {"Divya", {78, 85, 92, 67, 88}};
    int i, total = 0;
    for (i = 0; i < 5; i++)
        total += s1.marks[i];
    printf("%s's total marks = %d", s1.name, total);
    return 0;
}
```

Both concepts can even be combined — an array of structures where each structure itself contains an array member — to build more complex records, such as marks for many students across many subjects.

10.7 Passing Structure & Array of Structure to Function

Like ordinary variables, structures can be passed to functions. C allows two approaches: passing the entire structure by value (a full copy is made) or passing a pointer to the structure (the function works directly on the original data). An array of structures is passed the same way an ordinary array is passed — via its base address.

Passing a Structure by Value

Example 10.6 — Passing a Structure to a Function

```
pass_struct.c
#include <stdio.h>
struct Student
{
    int roll_no;
    float percentage;
};
void display(struct Student s) /* receives a COPY of the structure */
{
    printf("Roll: %d, Percentage: %.2f", s.roll_no, s.percentage);
}
int main()
{
    struct Student s1 = {105, 76.5};
    display(s1);
    return 0;
}
```

Note: When a structure is passed by value, any changes made to its members inside the function do NOT affect the original structure in main() — exactly like passing an ordinary variable by value.

Passing an Array of Structures to a Function

Example 10.7 — Passing an Array of Structures

pass_array_of_struct.c

```
#include <stdio.h>
struct Student
{
    int roll_no;
    char name[20];
};
void displayAll(struct Student arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d: %s\n", arr[i].roll_no, arr[i].name);
}
int main()
{
    struct Student class[2] = {{1, "Aarav"}, {2, "Bina"}};
    displayAll(class, 2); /* array name decays to base address */
    return 0;
}
```

bcalink.com

10.8 Structure & Pointer

A pointer to a structure stores the address of a structure variable. This is the more efficient way to pass large structures to functions, since only an address (typically 4 or 8 bytes) is copied instead of the entire structure's data.

When accessing structure members through a pointer, C provides the special arrow operator (->), which combines dereferencing and member access in one step.

Access Method	Syntax	Meaning
Direct (structure variable)	s1.member	Access member directly using the dot operator
Via pointer, using arrow	ptr->member	Shorthand for (*ptr).member — the preferred style
Via pointer, using dot	(*ptr).member	Explicitly dereference the pointer first, then access the member

Example 10.8 — Structure Pointer and Passing by Reference

```
struct_pointer.c
#include <stdio.h>
struct Student
{
    int roll_no;
    float percentage;
};
void updatePercentage(struct Student *ptr, float newVal)
{
    ptr->percentage = newVal; /* modifies the ORIGINAL structure */
}
int main()
{
    struct Student s1 = {110, 70.0};
    struct Student *ptr = &s1; /* pointer to structure */
    printf("Before: %.2f\n", ptr->percentage);
    updatePercentage(&s1, 92.5);
    printf("After: %.2f", s1.percentage); /* prints: 92.50 */
    return 0;
}
```

Note: Passing a pointer to a structure (call by reference) is generally preferred over passing the whole structure by value, especially for large structures, because it avoids the overhead of copying all the member data.

10.9 Bit Fields

Normally, even a variable that only needs to store 0 or 1 (like a true/false flag) occupies at least one full byte of memory. A bit field is a special structure member that lets the programmer specify the exact number of bits it should occupy, allowing several small flags to be packed tightly into a single byte or word — saving memory in situations with many small, limited-range values.

Syntax — Bit Field Declaration

```
struct StructureName
{
    data_type member_name : number_of_bits;
};
```

Example 10.9 — Using Bit Fields

bitfields.c

```
#include <stdio.h>
struct Flags
{
    unsigned int isMarried : 1; /* uses only 1 bit (0 or 1) */
    unsigned int age       : 7; /* uses only 7 bits (0-127) */
    unsigned int gender    : 1; /* uses only 1 bit (0 or 1) */
};
int main()
{
    struct Flags person = {0, 25, 1};
    printf("Married: %u, Age: %u, Gender: %u",
           person.isMarried, person.age, person.gender);
    printf("\nSize of struct Flags = %lu bytes", sizeof(person));
    return 0;
}
```

Without bit fields, the three members above (isMarried, age, gender) would ordinarily occupy 3 separate int-sized blocks; using bit fields, all three are packed into a single word of memory.

Point	Explanation
Purpose	Saves memory when many small-range flags or values need to be stored together
Declared with	A colon (:) followed by the number of bits, after the member name
Typical base type	unsigned int (or int) — the type that the bits are carved from
Limitation	Bit field members cannot have their address taken with the & operator, and cannot form arrays of bit fields

Point	Explanation
Common use	Flags/status registers, hardware interfacing, protocol headers, compact data storage

bcalink.com

10.10 Union and Its Importance

A union is a user-defined data type, declared using the union keyword, that looks syntactically almost identical to a structure — but behaves very differently in memory. In a union, all members share the same memory location, so the union's total size equals the size of its largest member, not the sum of all members as in a structure. Only one member can hold a valid value at any given time; storing a new value into one member overwrites whatever was stored in the others.

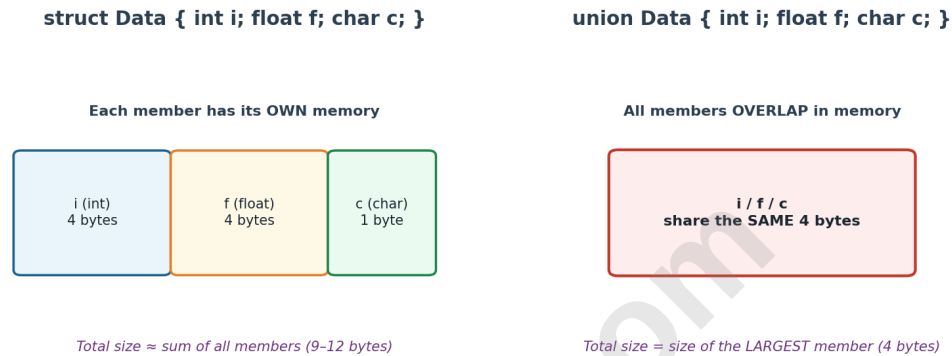


Fig 10.3 — Memory allocation: structure vs. union

Syntax — Union Declaration

```
union UnionName
{
    data_type member1;
    data_type member2;
    ...
};
```

Example 10.10 — Union Members Sharing Memory

```

union_demo.c
#include <stdio.h>
union Data
{
    int i;
    float f;
    char c;
};
int main()
{
    union Data d;
    d.i = 65;
    printf("d.i = %d\n", d.i);
    d.f = 3.14;          /* overwrites d.i's memory */
    printf("d.f = %.2f\n", d.f);
    printf("Size of union Data = %lu bytes", sizeof(d));
    return 0;
}

```

Explanation

- d.i, d.f, and d.c all occupy the SAME 4 bytes of memory.
- Setting d.f overwrites the value previously stored in d.i, since they physically share the same location.
- sizeof(d) equals 4 — the size of the largest member (float), not the sum $4 + 4 + 1 = 9$ as it would be for a structure.

Importance / Uses of Union

- Memory efficiency: Saves space when a variable needs to hold different types of data at different times, but never more than one at once.
- Type-punning / interpretation: Allows the same memory to be viewed and interpreted as different data types (commonly used in low-level and hardware programming).
- Tagged/variant data: Used with an extra "type" flag to build variant records that can hold one of several possible data types.
- Embedded and systems programming: Useful in memory-constrained environments where every byte matters.

10.11 Structure vs. Union

Basis	Structure	Union
Keyword	struct	union

Basis	Structure	Union
Memory allocation	Separate memory for each member	Shared memory for all members
Total size	Sum of the sizes of all members (plus possible padding)	Size of the largest member only
Simultaneous access	All members can hold valid values at the same time	Only ONE member holds a valid value at any given time
Modifying one member	Does not affect other members	Overwrites the value of any other member sharing that memory
Typical use case	Grouping distinct, related attributes of a record (e.g. a Student)	Saving memory when a variable represents only one of several possible types at a time

Note: As a quick rule of thumb: use a structure when you need to store several pieces of data together and access all of them at once; use a union when you only ever need one of several possible values stored in the same space at any given time.

Key Takeaways

- ✓ A structure groups variables of different data types under one name, stored in contiguous memory, and accessed with the dot (.) operator.
- ✓ Structures can be nested, combined with arrays (array of structures or array within a structure), and initialized like arrays.
- ✓ Structures may be passed to functions by value (a copy) or by pointer/reference (direct access via the -> arrow operator).
- ✓ Bit fields let structure members occupy a specified number of bits, saving memory for small-range flag-like values.
- ✓ A union allocates one shared block of memory for all its members, so its size equals its largest member's size, and only one member is valid at a time.
- ✓ The key difference between structure and union is memory allocation: structures allocate separate memory per member; unions share a single memory block among all members.

— End of Unit 10 —