Programming in C

Chapter Notes

Structures, Union and Enumerated Data Types

Topics Covered

6.1 Structures

6.2 Unions

6.3 Enumerated Data Types

Prepared for Engineering Students

6.1 Structures

Introduction

A structure is a user-defined data type available in C that allows the grouping of variables of different data types under a single name. While an array can only hold a collection of elements of the same data type, a structure lets a programmer combine dissimilar data types, such as an integer, a float and a character array, into one logical unit. Structures are widely used in engineering applications to represent real-world entities, for example, a student record containing a roll number, name and marks, or an employee record containing an ID, name and salary.

Defining a Structure

A structure definition begins with the keyword struct followed by a structure tag (name) and a list of member declarations enclosed in braces. Defining a structure only creates a template or blueprint; it does not allocate any memory until a variable of that structure type is declared.

```
struct StudentTag {
    int roll_no;
    char name[30];
    float marks;
};
```

Here, StudentTag is the structure tag, and roll_no, name and marks are the members of the structure. The definition must always end with a semicolon.

Declaring and Accessing Structure Members

Once a structure has been defined, variables of that structure type can be declared in the same way as ordinary variables, by writing the keyword struct, the tag name, followed by the variable name(s).

```
struct StudentTag s1, s2;
```

It is also possible to declare structure variables at the same time the structure is defined, by listing the variable names just before the closing semicolon of the definition.

Members of a structure variable are accessed using the dot operator (also called the member access or period operator), written between the structure variable name and the member name.

```
s1.roll_no = 101;
strcpy(s1.name, "Aditi Sharma");
s1.marks = 87.5;

printf("%d %s %f", s1.roll_no, s1.name, s1.marks);
```

> *Note: Each structure variable gets its own independent copy of the members. Changing s1.marks has no effect on s2.marks.*

Typedef Declaration

The typedef keyword is used to create a new, often shorter, name (an alias) for an existing data type, including structure types. This avoids the need to repeatedly write the struct keyword every time a variable of that type is declared, and makes the code more readable.

```
typedef struct {
    int roll_no;
    char name[30];
    float marks;
} Student;

Student s1, s2;   /* struct keyword no longer needed */
```

typedef does not create a new data type in the strict sense; it simply introduces a synonym for an already existing type. It is commonly combined with an anonymous structure definition (a structure with no tag name) as shown above.

Initialization of Structure

A structure variable can be initialized at the time of declaration by supplying values for its members inside braces, separated by commas, in the same order in which the members are declared.

```
struct StudentTag s1 = {101, "Aditi Sharma", 87.5};
```

From the C99 standard onward, designated initializers may also be used, which allow specific members to be initialized by name, in any order:

```
struct StudentTag s1 = {.name = "Aditi", .roll_no = 101, .marks = 87.5};
```

> *Note: If fewer values are supplied than the number of members, the remaining members are initialized to zero (for numeric types) or empty (for character arrays).*

Arrays of Structure

Just as arrays of basic data types such as int or float can be created, arrays of structures can also be declared. An array of structures is extremely useful when a program needs to store and process many records of the same kind, such as a list of students in a class or a list of employees in a department.

```
struct StudentTag class[3] = {
    {101, "Aditi", 87.5},
    {102, "Rahul", 76.0},
    {103, "Meena", 91.2}
};

for (int i = 0; i < 3; i++)
    printf("%d %s %.2f\n", class[i].roll_no, class[i].name, class[i].marks);
```

Each element of the array class is itself a complete structure variable, so individual members are accessed by combining array indexing with the dot operator, as in class[i].roll_no.

Nested Structure

A structure can contain another structure as one of its members; this is known as a nested structure. Nesting is useful for representing hierarchical data, such as an employee structure that contains a date-of-birth structure within it.

```
struct DateTag {
    int day, month, year;
};

struct EmployeeTag {
    int emp_id;
    char name[30];
    struct DateTag doj;   /* nested structure member */
};

struct EmployeeTag e1;
e1.doj.day = 15;
e1.doj.month = 8;
e1.doj.year = 2020;
```

Members of a nested structure are accessed by chaining the dot operator, first to reach the outer member and then to reach the inner member, as shown in e1.doj.day.

Structures and Functions

Structures can be used with functions in three principal ways:

- Passing individual structure members to a function, just like ordinary variables.
- Passing an entire structure variable by value, which copies the whole structure into the function's local parameter.
- Passing the address of a structure variable (a pointer to structure) to a function, which is more memory-efficient because no copy of the structure is made.

A function can also return a structure as its result. Passing large structures by value can be inefficient because the entire structure is copied onto the stack; for this reason, passing structures by pointer is generally preferred in real applications.

```
void display(struct StudentTag s) {          /* pass by value */
    printf("%d %s\n", s.roll_no, s.name);
}

struct StudentTag getStudent() {              /* returning a structure */
    struct StudentTag temp = {104, "Karan", 65.0};
    return temp;
}
```

Pointer to a Structure

A pointer variable can be declared to store the address of a structure variable, in the same way a pointer stores the address of any other variable. Structure pointers are declared using the struct keyword, the tag name, an asterisk, and the pointer name.

```
struct StudentTag s1 = {101, "Aditi", 87.5};
struct StudentTag *ptr = &s1;
```

When a pointer to a structure is used, members are accessed using the arrow operator (->) instead of the dot operator. The expression ptr->roll_no is equivalent to (*ptr).roll_no.

```
printf("%d", ptr->roll_no);      /* preferred, arrow operator */
printf("%d", (*ptr).roll_no);   /* equivalent, dereference then dot */
```

Passing a pointer to a structure into a function allows the function to directly modify the original structure's members, since it is operating on the same memory location rather than a copy.

Self-Referential Structure

A self-referential structure is a structure that contains, among its members, a pointer to a structure variable of its own type. Such structures cannot contain the structure type itself as a direct member (which would require infinite memory), but they can contain a pointer to it, because a pointer has a fixed, known size regardless of the type it points to.

```
struct Node {
    int data;
    struct Node *next;   /* pointer to the same structure type */
};
```

Self-referential structures form the foundation of many important data structures in computer science, including linked lists, stacks, queues and trees. In the example above, the next pointer allows one Node to be linked to another, and this chaining of nodes is precisely how a singly linked list is built.

6.2 Unions

Introduction and Defining a Union

A union is a user-defined data type, similar in syntax to a structure, that allows different data types to be stored in the same memory location. Unlike a structure, where each member has its own separate storage, all members of a union share a single, common block of memory. A union is defined using the keyword union, followed by a tag name and a list of member declarations.

```
union DataTag {
    int i;
    float f;
    char c[4];
};
```

The compiler allocates memory for a union equal to the size of its largest member, since only one member can hold a meaningful value at any given time.

Declaring and Accessing Union Members

Union variables are declared in the same manner as structure variables, and their members are accessed using the same dot operator (or the arrow operator, when accessed through a pointer).

```
union DataTag d1;
d1.i = 10;
printf("%d", d1.i);   /* prints 10 */

d1.f = 3.14;
printf("%f", d1.f);   /* prints 3.14, but d1.i is now meaningless */
```

> *Note: Assigning a new value to one member of a union overwrites the value previously stored, because all members occupy the same memory address. Only the most recently assigned member holds a valid, usable value.*

Initialization of Union

A union variable can be initialized at the time of declaration, but only its first member can be initialized using ordinary brace initialization, since all members share the same memory.

```
union DataTag d1 = {10};   /* initializes member i, the first member, to 10 */
```

To initialize a member other than the first, a designated initializer must be used:

```
union DataTag d2 = {.f = 3.14};
```

Arrays of Union Variables

Similar to structures, an array of union variables can be declared when many union-typed values need to be stored and processed together. Each element of such an array independently reserves memory equal to the size of the union's largest member.

```
union DataTag arr[5];
arr[0].i = 25;
arr[1].f = 9.81;
```

Nested Union

A union can be nested inside a structure or another union, meaning one of its members can itself be a union or structure type. This is useful when a single field within a larger record may legitimately hold one of several different data types.

```
union Inner {
    int x;
    float y;
};

union Outer {
    union Inner in;
    char ch;
};
```

Union Under Structure

A very common and practical use of unions is embedding a union as a member inside a structure. This allows a structure to have some fields that are always present (stored normally, with independent memory) together with one field that can flexibly hold different types of data depending on context, without wasting memory on unused type variants.

```
struct Item {
    int type;             /* 0 = integer, 1 = float */
    union {
        int i_val;
        float f_val;
    } value;               /* union embedded within a structure */
};

struct Item it;
it.type = 1;
it.value.f_val = 45.6;
```

Here, the type field always occupies its own memory, while value shares memory between i_val and f_val, and the program uses type to keep track of which member is currently valid.

Differences Between Structure and Union

Basis	Structure	Union
Keyword	struct	union
Memory allocation	Separate memory for each member; total size is the sum of all members (plus padding)	Common shared memory; total size equals the size of the largest member

Basis	**Structure**	**Union**
Value storage	All members can hold valid values simultaneously	Only one member holds a valid value at any given time
Memory efficiency	Less memory efficient when only one field is needed at a time	More memory efficient for mutually exclusive data
Use case	Grouping related but independent data, such as a record	Representing data that can be one of several types
Initialization	All members may be initialized together	Only the first member (or a designated member) may be initialized

6.3 Enumerated Data Types

Introduction

An enumerated data type, declared with the keyword enum, is a user-defined type consisting of a set of named integer constants, called enumerators. Enumerations are used to make a program more readable and self-documenting by giving meaningful names to a fixed set of related values, instead of using plain, unexplained integer literals scattered through the code.

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

By default, the compiler assigns integer values to the enumerators starting from 0 and increasing by 1 for each subsequent name, so in the example above, SUN is 0, MON is 1, and so on up to SAT which is 6.

Assigning and Accessing Enumerated Variables

A variable of an enumerated type can be declared and assigned any one of the named constants belonging to that enumeration. It is also possible to override the default values by explicitly assigning integers to one or more enumerators; any enumerator that follows an explicitly assigned value continues counting upward from that value.

```
enum Day today;
today = WED;
printf("%d", today);          /* prints 3 */

enum Month {JAN = 1, FEB, MAR, APR};  /* FEB=2, MAR=3, APR=4 */
```

Enumerator names must be unique within their scope, but it is legal for two different enumerators to be assigned the same integer value.

Enumeration Type Conversion

Internally, the C compiler treats enumerators purely as integer constants, and an enum variable is implemented using an integer type. Because of this, an enum value can be implicitly converted to an int wherever an integer is expected, such as in arithmetic expressions or printf statements using the %d format specifier.

The reverse conversion, from an int to an enum type, is not performed automatically and requires an explicit type cast, because the compiler does not check whether the integer value being assigned actually corresponds to one of the enumeration's valid named constants.

```
enum Day d;
int x = 2;
d = (enum Day) x;     /* explicit cast required, sets d to TUE */

int y = MON + 1;      /* implicit conversion to int, y becomes 2 */
```

> *Note: C performs no runtime bounds checking on enum variables, so it is possible to assign an out-of-range integer value to an enum variable through a cast, which can lead to logically invalid states if not handled carefully by the programmer.*

Comparing and I/O Operations on Enumerated Types

Because enumerators are internally integers, values of an enumerated type can be compared directly using the standard relational operators such as ==, !=, <, > and so on. This makes enumerations convenient for use in conditional statements and switch statements.

```
if (today == SUN || today == SAT)
    printf("Weekend");
else
    printf("Weekday");

switch (today) {
    case MON: printf("Monday"); break;
    case TUE: printf("Tuesday"); break;
    default:  printf("Some other day");
}
```

For input and output operations, C provides no direct way to print or read the symbolic name of an enumerator; the standard library functions such as printf and scanf work only with its underlying integer value using the %d format specifier. To display the enumerator's name as text, a programmer typically writes a lookup, such as an array of strings indexed by the enum value, or an explicit switch or if-else statement that maps each integer back to its corresponding name.

```
char *dayNames[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
printf("%s", dayNames[today]);   /* prints the name corresponding to today */
```

Multiple Choice Questions (MCQs)

Answer keys are provided at the end of this section.

1. Which keyword is used to define a structure in C?
   (a) union (b) struct (c) typedef (d) enum

2. Which operator is used to access a member of a structure through a structure variable (not a pointer)?
   (a) -> (b) . (c) :: (d) &

3. Which operator is used to access a structure member through a pointer to that structure?
   (a) . (b) * (c) -> (d) %

4. What is the main purpose of the typedef keyword?
   (a) To delete a data type (b) To create an alias for an existing data type (c) To initialize a structure (d) To allocate memory

5. In an array of structures such as struct Student s[5], how is the marks member of the third element accessed?
   (a) s.marks[3] (b) s[3].marks (c) s[2].marks (d) marks[2].s

6. A structure that contains another structure as a member is called a:
   (a) Self-referential structure (b) Nested structure (c) Union structure (d) Recursive structure

7. A structure containing a pointer to a structure of its own type is called a:
   (a) Nested structure (b) Self-referential structure (c) Recursive union (d) Anonymous structure

8. Self-referential structures are most commonly used to implement:
   (a) Arrays (b) Linked lists (c) Enumerations (d) Unions

9. When a structure is passed to a function by value, the function receives:
   (a) The address of the structure (b) A copy of the entire structure (c) Only the first member (d) Nothing

10. Which keyword is used to define a union in C?
    (a) struct (b) enum (c) union (d) typedef

11. The amount of memory allocated to a union variable is equal to:
    (a) The sum of the sizes of all members (b) The size of its smallest member (c) The size of its largest member (d) Always 4 bytes

12. In a union, at any given time:
    (a) All members hold valid values (b) Only the most recently assigned member holds a valid value (c) No member holds a value (d) Only the first member can ever be used

13. When a union variable is initialized using ordinary brace initialization without a designator, which member is initialized?
    (a) The largest member (b) The last member (c) The first member (d) All members

14. By default, the first enumerator in an enum declaration is assigned the integer value:
    (a) 1 (b) -1 (c) 0 (d) Undefined

15. Converting an int value into an enum type in C requires:

(a) No conversion, it is always automatic (b) An explicit type cast (c) The use of typedef (d) It is not possible under any circumstances

Answer Key

1-b 2-b 3-c 4-b 5-c 6-b 7-b 8-b 9-b 10-c 11-c 12-b 13-c 14-c 15-b

Broad / Descriptive Questions

1. Define a structure in C. Explain, with a suitable example, how a structure is defined, declared and how its members are accessed.

2. What is the purpose of the typedef keyword? Explain with an example how typedef is used along with structures.

3. Explain the initialization of structure variables in C. How does designated initialization differ from ordinary initialization? Illustrate with examples.

4. What is meant by an array of structures? Write a C program to store and display the records of five students using an array of structures.

5. What is a nested structure? Explain with an example how members of a nested structure are accessed.

6. Explain the different ways in which structures can be used with functions. Discuss the advantages of passing a structure by pointer over passing it by value.

7. What is a self-referential structure? Explain its significance in implementing linked data structures such as linked lists.

8. Define a union. Explain how memory is allocated for a union variable, and how this differs from memory allocation for a structure.

9. Distinguish between structure and union with respect to memory allocation, value storage and typical use cases. Support your answer with a comparison table.

10. What is an enumerated data type? Explain how enumerated variables are assigned and accessed, and describe how type conversion, comparison and I/O operations are handled for enum types in C.