Parul® University

03010501ES01 - Programming for Problem Solving

UNIT 5 - User-Defined Functions, Structure and Unions

DEPARTMENT OF AI & DS

Parul® University

Outline

- User-Defined Functions
- Structures
- Unions

Parul® University

What is Enumerators (Enums)?

- An enumerator is a user-defined data type that assigns names to a set of integral constants. (Counting/Calculation)
- It makes your code more readable and organized by replacing raw numbers with meaningful names.
- Enums are defined using the ***enum*** keyword and can be of different types based on their usage.

How can we use Enums?

Syntax :

```
enum EnumName {
    CONSTANT1, CONSTANT2,
CONSTANT3
};
```

Example :

```
enum TrafficLight { Red, Yellow, Green };

int main() {
    enum TrafficLight signal = Red;
    if (signal == Red) {
        printf("STOP\n");
    }
    return 0;
}
```

Parul®
University

When can we use Enums?

- When you ***have a fixed set of related constants***:
  - Examples:
    - Days of the week (Monday, Tuesday, ...).
    - Traffic lights (Red, Yellow, Green).
    - Status codes (Success, Error, Pending).

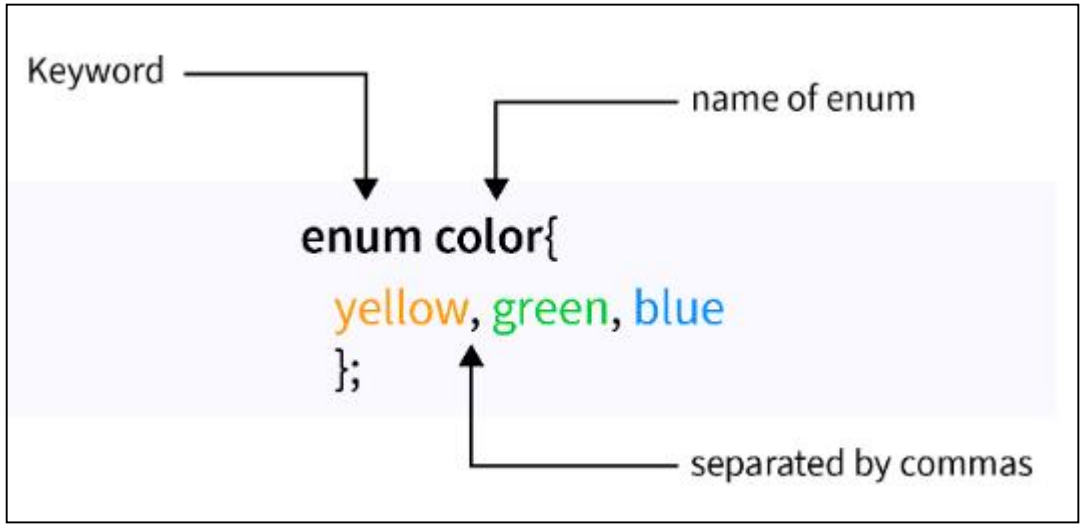
Enumeration in C

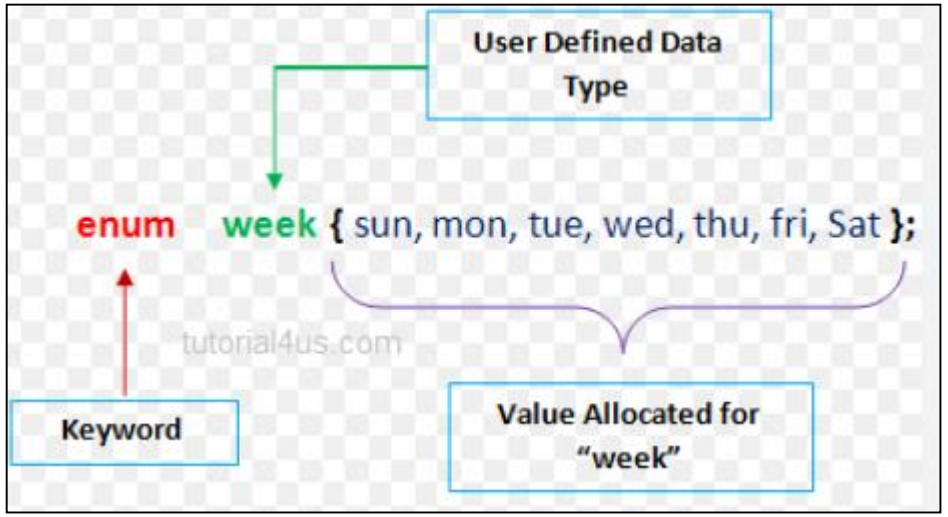
Syntax:

```
enum EnumName{
        //group of constants
}
```

Example:

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





Advantages of Using Enums:

- **Readability:** Enums make code easier to understand by using meaningful names instead of raw integer values.
- **Maintainability:** If you need to change the value of a constant, you only need to modify the enum definition, not search for all instances of the integer value throughout the code.
- **Simplified Switch Statements:** Enums can be used in switch statements, making the code more readable and easier to maintain.

Ex: Simplified Switch Statements:

- Enums can be used in switch statements, making the code more readable and easier to maintain.

```c
#include <stdio.h>

enum Days {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main() {
    enum Days today = MONDAY;
    switch (today) {
        case MONDAY:
            printf("Today is Monday.\n");
            break;
        case TUESDAY:
            printf("Today is Tuesday.\n");
            break;
        // ... other cases
    }

    return 0;
}
```

Parul®
University

Why do we use Enums?

1. Improves readability:Enums replace raw numbers with descriptive names, making the code easier to understand.

2. Reduces errors:Prevents the use of undefined or incorrect constants.

3. Enhances maintainability:If the values of constants change, you only update them in the enum declaration.

4. Better code organization:Groups related constants together under one name.

Parul® University

Example

```
enum DifficultyLevel { Easy, Medium, Hard };

void displayDifficulty(enum DifficultyLevel level) {
    if (level == Easy) printf("You selected EASY mode.\n");
    else if (level == Medium) printf("You selected MEDIUM mode.\n");
    else if (level == Hard) printf("You selected HARD mode.\n");
}

int main() {
    displayDifficulty(Medium);
    return 0;
}
```

Parul® University

Types / Usages of Enumerators (enums)

1. ***Basic Enumerators*** : A simple list of named constants where each constant is automatically assigned an integer value starting from 0 (by default).
2. ***Enumerators with Custom Values*** : Similar to basic enumerators, but you can manually assign specific integer values to some or all constants.
3. ***Enumerators with Arithmetic or Mixed Values*** : Enumerators where values are assigned using calculations or combinations of other constants.
4. ***Anonymous Enumerators*** : Enumerators without a name, used directly to define constants.
5. ***Typedef Enumerators*** : Enumerators defined with a typedef so you can use them without the enum keyword every time.
6. ***Bit-Flag Enumerators*** : Enumerators where each constant represents a power of 2, allowing multiple constants to be combined using bitwise operations.

Parul® University

Types / Usages of Enumerators (enums)

1. Basic Enumerators:

These are simple enumerations where constants are ***assigned integer values automatically starting from 0*** (default) or ***from a user-specified value.***

```
Syntax :
enum EnumName {
    CONSTANT1,
    CONSTANT2,
    CONSTANT3
};
```

```
#include <stdio.h>
enum DaysOfWeek {
    MONDAY,  // 0
    TUESDAY, // 1
    WEDNESDAY, // 2
    THURSDAY, // 3
    FRIDAY, // 4
    SATURDAY, // 5
    SUNDAY // 6
};

int main() {
    enum DaysOfWeek today = WEDNESDAY;
    printf("Today is day number: %d\n", today); // Outputs: 2
    return 0;
}
```

Parul® University

Types / Usages of Enumerators (enums)

2. Enumerators with Custom Values

You can assign specific integer values to the constants in the enum.
If a ***value isn't specified***, it will continue from the previous constant's value.

Syntax :

```
enum EnumName {
    CONSTANT1 = 10,
    CONSTANT2 = 20,
    CONSTANT3
};
```

```
#include <stdio.h>
enum ErrorCodes {
    SUCCESS = 0,
    ERROR_FILE_NOT_FOUND = 404,
    ERROR_ACCESS_DENIED = 403,
    ERROR_TIMEOUT = 408
};

int main() {
    enum ErrorCodes error = ERROR_FILE_NOT_FOUND;
    printf("Error Code: %d\n", error); // Outputs: 404
    return 0;
}
```

Real-World Analogy:
HTTP Status Codes: 200 for success, 404 for file not found, etc.

Parul® University

Types / Usages of Enumerators (enums)

3. Enumerators with Arithmetic or Mixed Values

Enums allow performing arithmetic operations during declaration, combining constants or offsets.

```
Syntax :
enum EnumName {
    CONSTANT1 = 10,
    CONSTANT2 = CONSTANT1 + 10,
    CONSTANT3 = CONSTANT2 * 2
};
```

```
#include <stdio.h>
enum MemorySizes {
    BYTE = 1,
    KILOBYTE = BYTE * 1024,
    MEGABYTE = KILOBYTE * 1024,
    GIGABYTE = MEGABYTE * 1024
};

int main() {
    printf("1 GB = %d bytes\n", GIGABYTE); // Outputs:
1073741824
    return 0;
}
```

Real-World Analogy:
Memory Units: 1 KB = 1024 Bytes, 1 MB = 1024 KB, 1 GB = 1024 MB.

Parul® University

Types / Usages of Enumerators (enums)

4. Anonymous Enumerators

Enums can be defined ***without a name*** when their constants are to be used directly.

Syntax :

```
enum {
    CONSTANT1 = 1,
    CONSTANT2,
    CONSTANT3
};
```

```
#include <stdio.h>

int main() {
    enum {
        LOW,
        MEDIUM,
        HIGH
    };

    int priority = HIGH;
    printf("Priority level: %d\n", priority); // Outputs: 2
    return 0;
}
```

Parul® University

Types / Usages of Enumerators (enums)

5. Typedef Enumerators

Enums can be defined with a ***typedef*** to simplify their usage.

Syntax :

```
typedef enum {
    CONSTANT1,
    CONSTANT2,
    CONSTANT3
} EnumAlias;
```

```
#include <stdio.h>

typedef enum {
    SPRING,
    SUMMER,
    FALL,
    WINTER
} Season;

int main() {
    Season currentSeason = FALL;
    printf("Current season: %d\n", currentSeason); //
Outputs: 2
    return 0;
}
```

Parul® University

Types / Usages of Enumerators (enums)

6. Bit-Flag Enumerators

Enums can be defined with a ***typedef*** to simplify their usage.

Syntax :

```
enum EnumName {
    FLAG1 = 1, // 1 << 0
    FLAG2 = 2, // 1 << 1
    FLAG3 = 4  // 1 << 2
};
```

```
#include <stdio.h>

enum Permissions {
    READ = 1,   // 0001
    WRITE = 2,  // 0010
    EXECUTE = 4 // 0100
};

int main() {
    int permissions = READ | WRITE; // Combine using
bitwise OR
    printf("Permissions: %d\n", permissions); // Outputs: 3
(0001 | 0010 = 0011)
    return 0;
}
```

Parul® University

Structure

What is a Structure in C?

- A structure is a ***user-defined data type*** that allows you to group multiple variables of different data types under one name.
- Structures are particularly useful when you ***need to represent a complex entity with various attributes***.
- For example, if you need to represent a "***Person***," you can group attributes like name (string), age (integer), and height (float) together using a structure.

Parul® University

Declaring a Structure in C

Syntax :

```
struct StructureName {
    dataType member1;
    dataType member2;
    // More members can be added as needed
};
```

Example :

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

Here:
name is a ***string (character array)*** to store the person's name.
age is an ***integer*** to store the person's age.
height is a ***float*** to store the person's height.

Parul® University

Accessing and Using Structures

```
#include <stdio.h>
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Create a structure variable
    struct Person person1;
    // Assign values to the members
    printf("Enter name: ");
    scanf("%s", person1.name);
    printf("Enter age: ");
    scanf("%d", &person1.age);
    printf("Enter height: ");
    scanf("%f", &person1.height);

    // Print the details
    printf("\nPerson Details:\n");
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f meters\n", person1.height);
    return 0;
}
```

Parul®
University

Key Points

- Structures allow combining variables of different types into one entity.
- Use the ***struct*** keyword to define a structure.
- Access members using the ***dot (.)*** operator.
- Structures are essential when dealing with real-world objects or entities.

Parul® University

Complex Structures

- A complex structure in C is a structure that ***contains other structures or arrays as members***.
- It allows you to create a hierarchy of related data, making it easier to represent and manage more complex entities.

A complex structure in C refers to a structure that contains:

- Nested structures: A structure as a member of another structure.
- Arrays within structures: Arrays as members of a structure.
- Pointers in structures: Members that are pointers.
- Structures with self-referential pointers: A structure having a pointer to the same structure type (used in linked lists, trees, etc.).

Parul® University

How to Define a Complex Structure?

- Include one structure inside another (nested structures).
- Use arrays within a structure.

Syntax for Nested Structures

```
struct Structure1 {
    dataType member1;
    dataType member2;
    // More members
};
struct Structure2 {
    struct Structure1 nestedStructure; // Structure inside
another
    dataType member3;
};
```

Syntax for Structures with Arrays

```
struct Structure {
    dataType arrayMember[arraySize];
    dataType member2;
};
```

Parul® University

Example: Nested Structures

```
#include <stdio.h>
// Define the Address structure
struct Address {
    char street[100];
    char city[50];
    int zipCode;
};
// Define the Employee structure
struct Employee {
    char name[50];
    int age;
    struct Address address; // Nested structure
};
```

```
int main() {
    struct Employee emp;
    // Input employee details
    printf("Enter name: ");
    scanf("%s", emp.name);
    printf("Enter age: ");
    scanf("%d", &emp.age);
    printf("Enter street: ");
    scanf("%s", emp.address.street);
    printf("Enter city: ");
    scanf("%s", emp.address.city);
```

```
 printf("Enter zip code: ");
    scanf("%d", &emp.address.zipCode);
    // Print employee details
    printf("\nEmployee Details:\n");
    printf("Name: %s\n", emp.name);
    printf("Age: %d\n", emp.age);
    printf("Address: %s, %s - %d\n",
emp.address.street, emp.address.city,
emp.address.zipCode);
    return 0;
}
```

Parul® University

Array of Structures

- An array of structures means we have multiple instances of a structure stored in an array.
- Each element in the array is a separate structure.
- Think of it like having multiple boxes, and each box can store information of a single item. All the boxes are of the same type.

```
struct Student {
    char name[50];
    int age;
    float marks;
};
struct Student students[3]; // Array of 3 Students
```

Example:
Imagine you are collecting information about students in a class:

- Each student has a name, age, and marks.
- You create a structure Student and an array of 3 students.

Parul® University

Example: Array of Structures

```
#include <stdio.h>
// Define the structure
struct Student {
    char name[50];
    int age;
    float marks;
};
int main() {
    // Declare an array of structures
    struct Student students[3]; // Array of 3 structures
```

```
// Input details for each student
    for (int i = 0; i < 3; i++) {
        printf("Enter details for student %d:\n", i + 1);
printf("Name: ");
        scanf("%s", students[i].name);
        printf("Age: ");
        scanf("%d", &students[i].age);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
        printf("\n");
    }
```

```
// Display details of each student
    printf("Student Details:\n");
    for (int i = 0; i < 3; i++) {
        printf("Student %d: Name: %s, Age: %d, Marks: %.2f\n",
                i + 1, students[i].name, students[i].age, students[i].marks);
    }

    return 0;
}
```

Parul® University

Array within a Structure

An array within a structure means the structure itself has an array as one of its members.
Think of it like having one big box, and inside this box, there is an array to store multiple related items

```
struct Library {
    char name[50];
    char books[3][50]; // Array to store 3 book names
};
```

Example:
Imagine you have a library:

- Each library has a name and multiple books.
- You create a structure Library that contains an array of books.

Parul® University

Example: Array within a Structure

```
#include <stdio.h>
// Define a structure for a Library
struct Library {
    char name[50];
    char books[3][50]; // Array to store 3 book names
};
```

```
int main() {
    struct Library library;
    // Input library details
    printf("Enter library name: ");
    scanf("%s", library.name);

printf("\nEnter names of 3 books:\n");
    for (int i = 0; i < 3; i++) {
        printf("Book %d: ", i + 1);
        scanf("%s", library.books[i]);
    }
```

```
// Display library details
    printf("\nLibrary Name: %s\n", library.name);
    printf("Books Available:\n");
    for (int i = 0; i < 3; i++) {
        printf("Book %d: %s\n", i + 1, library.books[i]);
    }
    return 0;
}
```

Parul® University

Structure and Functions

- Structures can be used with functions in C for better modularity and reusability.
- Users can pass a structure to a function, return a structure from a function, or manipulate structure members within a function.

Passing Structures to Functions

1. ***By Value***: The function receives a copy of the structure.
2. ***By Reference (Pointer)***: The function can modify the original structure.

```
void functionName(struct StructureName structureVariable);
```

```
void functionName(struct StructureName *structureVariable);
```

Parul® University

Example: Passing Structures to Functions (By Value)

```c
#include <stdio.h>

struct Student {
    char name[50];
    int rollNo;
    float marks;
};

// Function to modify student details
void updateMarks(struct Student *stu, float newMarks) {
    stu->marks = newMarks; // The arrow operator (->) in C is used to access members of a structure through a pointer
}
```

```c
int main() {
    struct Student student = {"Bob", 102, 76.5};

    printf("Before Update:\n");
    printf("Marks: %.2f\n", student.marks);

    // Update marks by passing structure pointer
    updateMarks(&student, 95.0);

    printf("After Update:\n");
    printf("Marks: %.2f\n", student.marks);

    return 0;
}
```

Parul® University

Example: Passing Structures to Functions(By Reference)

```c
#include <stdio.h>

struct Student {
    char name[50];
    int rollNo;
    float marks;
};

// Function to display student details
void displayStudent(struct Student stu) {
    printf("Name: %s\n", stu.name);
    printf("Roll No: %d\n", stu.rollNo);
    printf("Marks: %.2f\n", stu.marks);
}
```

```c
int main() {
    struct Student student = {"Alice", 101, 89.5};

    // Call function by passing structure
    displayStudent(student);

    return 0;
}
```

Parul® University

Anonymous Structures

- An anonymous structure is a structure declared without a name. It is primarily used for grouping related variables under one scope without requiring the structure to be explicitly named.

Key Features:

- No explicit name for the structure.
- Useful for simple and temporary groupings of data.
- Typically used in conjunction with typedef or as part of other structures.

```
struct {
    dataType member1;
    dataType member2;
    // More members
} variableName;
```

Parul® University

Example

```
#include <stdio.h>

int main() {
    struct {
        char name[50];
        int age;
    } person;
```

```
// Assign values
    printf("Enter name: ");
    scanf("%s", person.name);
    printf("Enter age: ");
    scanf("%d", &person.age);

    // Display values
    printf("Name: %s\n", person.name);
    printf("Age: %d\n", person.age);

    return 0;
}
```

Real-World Example: You want to quickly group and manage data for a temporary purpose, such as storing information about a one-time event (e.g., a single visitor's details).

Parul® University

Self-Referential Structures

- A self-referential structure is a structure that contains a pointer to an instance of the same structure type.
- These are commonly used for creating dynamic and linked data structures like linked lists, trees, or graphs.

Key Features:

- Contains a pointer to its own type.
- Allows the creation of dynamic data structures.
- Commonly used in linked data structures.

```
struct StructureName {
    dataType member;
    struct StructureName *pointerToSameStructure;
};
```

Parul® University

Example

```
#include <stdio.h>
#include <stdlib.h>

// Define a self-referential structure
struct Node {
    int data;
    struct Node *next; // Pointer to the next node
};
int main() {
    // Create two nodes
    struct Node *node1 = (struct Node
*)malloc(sizeof(struct Node));
    struct Node *node2 = (struct Node
*)malloc(sizeof(struct Node));
```

```
// Assign values and link nodes
    node1->data = 10;
    node1->next = node2;
    node2->data = 20;
    node2->next = NULL;
    // Display the list
    struct Node *current = node1;
    while (current != NULL) {
        printf("Data: %d\n", current->data);
        current = current->next;
    }
    // Free memory
    free(node1);
    free(node2);

    return 0;
}
```

Parul® University

Structure Padding

- Structure padding is a mechanism where the compiler adds extra bytes (padding) between structure members to align them according to the system's word size.
- This ensures better performance by aligning data in memory.

Key Features:

- Added bytes improve data access speed.
- Padding is compiler and architecture-dependent.
- Can lead to wasted memory.

Structure padding happens automatically, no specific syntax is needed

Parul® University

Example

```
#include <stdio.h>

struct Padded {
    char a;      // 1 byte
    int b;       // 4 bytes
    char c;      // 1 byte
};

int main() {
    printf("Size of struct Padded: %zu bytes\n",
sizeof(struct Padded));
    return 0;
}
```

Explanation:
On a 32-bit system, int must align to 4 bytes.
char occupies 1 byte.
The compiler adds padding bytes to ensure alignment:
1 byte for a + 3 bytes padding to align b.
4 bytes for b.
1 byte for c + 3 bytes padding for alignmen

Real-World Example:
When working with hardware devices or protocols where data needs to align with memory boundaries for efficiency.

Parul® University

Example

Avoiding Padding:
You can use the #pragma pack directive to avoid padding (not always recommended for performance reasons):

```
#pragma pack(1) // Disable padding
struct Compact {
    char a;
    int b;
    char c;
};
#pragma pack() // Re-enable default alignment

int main() {
    printf("Size of struct Compact: %zu bytes\n", sizeof(struct Compact));
    return 0;
}
```

Parul® University

Pointers in Structures

- A pointer in a structure is a member of a structure that holds the memory address of a specific data type (e.g., an integer, a float, a structure, or even another structure).
- Pointers in structures are very useful when dynamically managing memory or linking multiple structures together. Below, you'll find a detailed explanation, syntax, and example in simple terms.

```
struct StructName {
    int *ptr;          // Pointer to an integer
    float *fptr;       // Pointer to a float
    char *str;         // Pointer to a character
array (string)
};
```

Key Points

- **Pointer Member**: A pointer inside a structure can store the address of variables or arrays.
- **Dynamic Allocation**: You can dynamically allocate memory to these pointer members during runtime using malloc() or calloc().
- **Accessing Pointer Members:**
  - If you have a structure variable, use the dot operator (.) to access its members.
  - If you have a pointer to a structure, use the arrow operator (->) to access its members.

Parul® University

Example

```
#include <stdio.h>
#include <stdlib.h>

// Define a structure with a pointer
struct Student {
    char *name;
    int *marks;
};

int main() {
    struct Student s;
```

```
// Allocate memory for the name and marks dynamically

    s.name = (char *)malloc(50 * sizeof(char)); // Allocate memory for a string

    s.marks = (int *)malloc(sizeof(int)); // Allocate memory for an integer

    // Input student details
    printf("Enter student name: ");
    scanf("%s", s.name); // Store the name in dynamically allocated memory

printf("Enter student marks: ");
    scanf("%d", s.marks); // Store marks in dynamically allocated memory
```

```
// Display student details
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s.name);
    printf("Marks: %d\n", *s.marks);

    // Free allocated memory
    free(s.name);
    free(s.marks);

    return 0;
}
```

Parul® University

Unions

- A union is a user-defined data type in C that allows storing different data types in the same memory location.
- Unlike a structure, where each member gets its memory, all members of a union share the same memory.
- This means that a union can hold only one value at a time, even if it has multiple members.

Declaration of a Union :

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

Example :

```
union Data {
    int i;
    float f;
    char str[20];
};
```

Parul® University

Unions (Conti...)

Initialization of a Union : You can initialize a union when declaring it. However, since all members share the same memory, initializing one member overwrites any previously stored value.

```
union UnionName varName = {value};
```

```
union Data d1;
d1.i = 42;       // Assigning to the integer member
d1.f = 3.14;     // Overwrites the integer member
strcpy(d1.str, "Hello"); // Overwrites the float member
```

You can initialize a union directly during declaration:

```
union Data d2 = {42}; // Initialize integer
```

Parul® University

Bitfields in Unions

Bitfields in Unions : Bitfields are a way to specify that a member of a structure or union should occupy a specific number of bits in memory, rather than the default size for its type.
This is often used when working with hardware, protocols, or memory optimization.

Syntax :

```
struct {
    unsigned int bit1 : 1; // 1 bit
    unsigned int bit2 : 2; // 2 bits
    unsigned int bit3 : 3; // 3 bits
};
```

```
union Data d1;
d1.i = 42;       // Assigning to the integer member
d1.f = 3.14;     // Overwrites the integer member
strcpy(d1.str, "Hello"); // Overwrites the float member
```

Parul® University

Example

```
union Flags {
    struct {
        unsigned int option1 : 1;
        unsigned int option2 : 1;
        unsigned int option3 : 1;
    } bits;
    unsigned int allOptions;
};

int main() {
    union Flags f;
    f.bits.option1 = 1; // Enable option1
    f.bits.option2 = 0; // Disable option2
    f.bits.option3 = 1; // Enable option3

    printf("All options as bits: %d\n", f.allOptions);
    return 0;
}
```

Parul® University

Example of Bitfields

```
#include <stdio.h>

union DeviceStatus {
    struct {
        unsigned int powerOn : 1;
        unsigned int error : 1;
        unsigned int batteryLow : 1;
    } statusBits;
    unsigned int allStatus; // To view all status as a single integer
};

int main() {
    union DeviceStatus device;

    // Setting status
    device.statusBits.powerOn = 1;  // Device is powered on
    device.statusBits.error = 0;   // No error
    device.statusBits.batteryLow = 1; // Battery is low

    // Viewing status
    printf("Power On: %u\n", device.statusBits.powerOn);
    printf("Error: %u\n", device.statusBits.error);
    printf("Battery Low: %u\n", device.statusBits.batteryLow);

    // Combined status
    printf("All Status Bits: %u\n", device.allStatus);

    return 0;
}
```

Parul® University

What is typedef ?

- In simple terms, typedef is a keyword in C that allows you to create a new name (or alias) for an existing data type.
- Think of it as giving a nickname to a data type to make your code easier to read and understand.

Why Use typedef?

- Simplifies complex data types: You can give a shorter, more understandable name to complex types like pointers or structures.
- Improves code readability: Makes the code easier to understand for others (and yourself).
- Consistency: Helps maintain consistent naming for specific types.

Syntax : typedef existing_type new_name;

Parul® University

Example

Simple Type Alias

```
#include <stdio.h>

typedef unsigned int uint; // Creating an alias for unsigned int

int main() {
    uint age = 25; // Now, you can use 'uint' instead of 'unsigned int'
    printf("Age: %u\n", age);
    return 0;
}
```

Using typedef with Structures

```
#include <stdio.h>
typedef struct {
    int id;
    char name[50];
} Student; // Now, 'Student' is an alias for the struct

int main() {
    Student s1; // No need to write 'struct'
    s1.id = 1;
    strcpy(s1.name, "Dhruv");
    printf("ID: %d, Name: %s\n", s1.id, s1.name);
    return 0;
}
```

Parul® University

Example

Typedef with Pointers

```
#include <stdio.h>

typedef int* IntPtr; // Creating an alias for 'int*'

int main() {
    IntPtr ptr1, ptr2; // Both are now pointers to integers
    return 0;
}
```

Real-World Example : Program for a hospital system

```
#include <stdio.h>
typedef struct {
    int day;
    int month;
    int year;
} Date; // Alias for the date structure

int main() {
    Date today = {2, 2, 2025}; // Declaring a Date variable
    printf("Today's date: %02d/%02d/%d\n", today.day,
today.month, today.year);
    return 0;
}
```

Parul® University

Key Points

- ***typedef*** does not create new data types; it only creates an alias for existing ones.

- ***It is mainly used for*** :
  - Simplifying complex types.
  - Improving code readability.
  - Working with structures and pointers.

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in