Unit: Unit II

# Course Material: Unit II - User Defined Data Types
## 1. Brief Introduction
User-defined data types in programming languages allow developers to create more complex types that better reflect the structure of data being modeled. This unit will cover three prominent user-defined data types: **Structures**, **Unions**, and **Enumerated Data Types**. Understanding these concepts is essential for effective memory management, data organization, and implementing complex data structures in programs.
## 2. Key Concepts
### A. Structures
- **Definition**: A structure is a composite data type that groups together different data types under a single name.
- **Basic Operations**:
- **Declaring Structures**: Syntax to create a structure type.
- **Structure Variables**: Instantiation of a structure to create variables.
- **Accessing Members**: Using the dot (.) operator to access individual elements.
- **Initialization**: Assigning values to a structure’s members at the time of declaration.
- **Comparisons and Copying**: Mechanisms to compare structures and copy their values.
- **Typedef**: Creating an alias for a structure type for cleaner code.
### B. Union
- **Definition**: A union is a special data type that allows storing different data types in the same memory location. Only one member can contain a value at any given time.
- **Basic Operations**: Declaring unions, accessing members, and performing operations on them.
### C. Enumerated Data Type
- **Definition**: An enumerated type is a user-defined data type consisting of a set of named integer constants to represent distinct values.
- **Purpose**: Provides a way to organize and work with a set of related constants.
### D. Bit Fields
- **Definition**: A bit field allows the allocation of a specific number of bits for a particular member of a structure, providing control over memory usage.
### E. Advanced Structures
- **Nested Structures**: Structures that contain other structures as members.
- **Arrays within Structures**: Including arrays as members of structures, allowing for the grouping of related data.
- **Arrays of Structures**: Creating an array where each element is a structure.
- **Structures with Functions**: Passing structures to functions and returning structures.
- **Structures with Pointers**: Using pointers to manipulate structures for dynamic memory management and efficiency.
## 3. Examples
### A. Example of a Structure
```c
struct Student {
char name[50];
int rollNumber;
float GPA;
};
// Declaring a structure variable
struct Student student1;
// Accessing and initializing members
strcpy(student1.name, "Alice");
student1.rollNumber = 101;
student1.GPA = 3.5;
```
### B. Example of a Union
```c
union Data {
int intVal;
float floatVal;
char charVal;
};
// Declaring a union variable
union Data data;
```