GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING FOR WOMEN
(AUTONOMOUS)
(Affiliated to Andhra University, Visakhapatnam)
I B. Tech I Semester Regular Examinations, January 2025
Problem Solving Using C - 24RT11RC02
(Common to CSE, CSM, ECE and IT)

……..

SOLUTION SCHEME OF VALUATION

UNIT-I

1 a. Discuss the basic structure of a C program and explain the role of each component

A basic C program consists of several components that work together to define the structure, functionality, and behavior of the program. Here's a breakdown of the key components and their roles:

1. Preprocessor Directives

- **Purpose:** Preprocessor directives are instructions that are processed before the actual compilation of the code begins. They provide instructions to the preprocessor, which modifies the code or includes external code.
- **Common Directives:**
- #include: Used to include header files. For example, #include <stdio.h> includes the standard input/output library, which is used for input/output operations.
- #define: Defines constants or macros.
  Example:
  #include <stdio.h>

2. Global Declarations

- **Purpose:** Global variables and function prototypes are declared outside of functions but within the program. These declarations can be accessed by any function within the program.
- **Example:**
  int x; // Global variable declaration

3. The main() Function

- **Purpose:** The main() function is the entry point of any C program. When the program is executed, the execution starts from the main() function.
- **Return Type:** It returns an integer value, usually 0 to indicate successful execution. A non-zero return value indicates an error or abnormal termination.
- **Example:**
  int main() {
  // Program logic goes here
  return 0;

```
}
```

4. Local Declarations

- **Purpose:** Inside the main() function (or other functions), local variables are declared. These variables are only accessible within the function in which they are declared.
- **Example:**

```
int a = 5;  // Local variable
```

5. Statements and Expressions

- **Purpose:** Statements define the operations that are to be performed. They can be assignments, function calls, loops, conditionals, etc. Expressions calculate values that may be assigned to variables or returned.
- **Example:**

```
printf("Hello, World!\n");  // A statement that prints output

a = a + 1;  // An expression that modifies the value of a
```

6. Functions

- **Purpose:** Functions are blocks of code that perform specific tasks and can be called from the main() function or other functions. They help in modularizing the code.
- **Example:**

```
void printHello() {
   printf("Hello from a function!\n");
}
```

7. Comments

- **Purpose:** Comments are used for documentation purposes. They are ignored by the compiler but are useful for code explanation, making the code easier to understand and maintain.
- **Single-line comment:**

```
// This is a single-line comment
```

- **Multi-line comment:**

```
/* This is a
  multi-line comment */
```

8. Control Structures

- **Purpose:** Control structures like loops (for, while), conditionals (if, switch), and others control the flow of execution in the program.
- **Example:**

```
if (a > 0) {
   printf("Positive number\n");
}
```

9. Return Statement

- **Purpose:** The return statement is used to exit from a function and optionally send a value back to the function's caller. In the case of the main() function, it returns an integer value (commonly 0).
- **Example:**
  return 0; // Returns 0 from the main function
  Basic C Program Example:

```
#include <stdio.h>  // Preprocessor directive to include standard I/O functions
// Global declaration
int x = 10;
void greet() {  // Function definition
   printf("Hello, welcome to the C program!\n");  // Function call
}
int main() {
   int a = 5;  // Local variable declaration
   // Statement to print output
   printf("The value of a is: %d\n", a);
   // Calling the function greet
   greet();
   // Control structure (if statement)
   if (a < 10) {
      printf("a is less than 10\n");
   }

   // Returning from main function
   return 0;
}
```

1 b. Illustrate the purpose of the main() function in a C program with an example.

The main() function in a C program serves as the entry point for the program. When a C program is executed, the execution begins from the main() function. It is the function that the operating system calls to start the program's execution.

Key Points about main() Function:

1. Entry Point: It's the first function to be executed when a C program is run.
2. Return Value: It typically returns an integer value, which is used to indicate the success or failure of the program. A return value of 0 usually signifies successful execution, while a non-zero value indicates an error or abnormal termination.
3. Parameters: The main() function can optionally accept command-line arguments, typically defined as int argc (argument count) and char *argv[] (array of argument strings).

Basic Example to Illustrate main() Function

Here's a simple C program with a main() function that prints a message:

```
#include <stdio.h>  // Preprocessor directive to include the standard I/O library
// main function - Entry point of the program
int main() {
      printf("Hello, World!\n"); // Print a message to the console
   // Return 0 to indicate successful execution
   return 0;
}
```

Explanation:

1. #include <stdio.h>: This directive includes the standard input/output library, which allows us to use the printf() function to print messages to the console.
2. int main(): This is the entry point of the program. The return type is int, which means that the function is expected to return an integer value when it finishes. In this case, it returns 0 at the end.
3. printf("Hello, World!\n");: Inside the main() function, this line prints the message "Hello, World!" to the console. The \n is a newline character, which moves the cursor to the next line after printing the message.
4. return 0;: This line returns 0 from the main() function, signaling that the program has executed successfully. Returning 0 is a standard convention to indicate successful completion.

OR

2 a. Identify the types of constants in C and provide examples for each

In C programming, constants are fixed values that do not change during the execution of the program. Constants are used to make code more readable, maintainable, and less prone to errors. C provides several types of constants, each suited for different data types and purposes.

Types of Constants in C

1. Integer Constants

- Definition: These constants represent integer values without any fractional or decimal component.
- Examples:

```
10    // Decimal constant
0xFF  // Hexadecimal constant (255 in decimal)
075   // Octal constant (61 in decimal)
0b1011 // Binary constant (11 in decimal, C99 and later)
```

- Details:

- Decimal: A standard integer in base 10 (e.g., 10).
- Hexadecimal: Prefixed with 0x or 0X (e.g., 0xFF represents 255 in decimal).
- Octal: Prefixed with 0 (e.g., 075 represents 61 in decimal).
- Binary: Prefixed with 0b in C99 and later (e.g., 0b1011 represents 11 in decimal).

2. Floating-point Constants

- Definition: These constants represent real numbers or numbers with fractional parts.
- Examples:

```
3.14   // Floating-point constant (double by default)
2.0    // Another floating-point constant
1.2e3  // Scientific notation (1.2 × 10^3 = 1200)
5.6E-2 // Scientific notation (5.6 × 10^-2 = 0.056)
```

- Details:
  - Floating-point constants can be written in decimal format (e.g., 3.14).
  - They can also use scientific notation (e.g., 1.2e3 or 5.6E-2).

3. Character Constants

- Definition: These constants represent individual characters enclosed in single quotes. Each character constant corresponds to a unique ASCII value.
- Examples:

```
'a'    // Character constant (ASCII value of 'a' is 97)
'1'    // Character constant (ASCII value of '1' is 49)
'A'    // Character constant (ASCII value of 'A' is 65)
'\n'   // Special character constant (newline)
'\t'   // Special character constant (tab)
```

- Details:
  - Characters are enclosed in single quotes (e.g., 'a').
  - Special characters like newline (\n), tab (\t), backslash (\\), etc., are used for non-printable characters.

4. String Constants (String Literals)

- Definition: A string constant is a sequence of characters enclosed in double quotes. It represents a string of characters stored in memory.
- Examples:

```
"Hello, World!"   // String constant
"C programming"    // Another string constant
```

"123" // A string containing digits, not an integer

- Details:
  - Strings are enclosed in double quotes (e.g., "Hello").
  - The string is null-terminated, meaning the last character is '\0' (null character).

5. Enumerated Constants (Enum)

- Definition: Enumerated constants are used to define a set of named integer constants, which makes code more readable.
- Examples:
  enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
- Details:
  - By default, the first value starts at 0, and each subsequent value increments by 1.
  - You can assign specific integer values to the constants if needed:

enum Day { Sunday = 1, Monday = 2, Tuesday = 3 };

6. Constant Variables

- Definition: These are variables declared as constants using the const keyword. The value of a const variable cannot be modified after initialization.
- Examples:

```
const int MAX_LIMIT = 100;   // Constant integer

const float PI = 3.14159;    // Constant float

const char* greeting = "Hello, World!";  // Constant pointer to a string
```

- Details:
  - const is used to declare variables whose values are fixed after initialization.
  - These constants can be of any data type (integer, floating-point, string, etc.).

7. Macro Constants

- Definition: These are constants defined using the #define preprocessor directive. Macro constants are often used for symbolic names or to define values that remain unchanged throughout the program.
- Examples:

```
#define PI 3.14159    // Defining a constant for Pi
#define MAX_BUFFER_SIZE 1024  // Defining a constant buffer size
#define MAX(x, y) ((x) > (y) ? (x) : (y))  // Macro for maximum of two values
```

- Details:
  - The #define directive is used to define symbolic constants.

- Macro constants can be numerical or symbolic and are typically used for configuration, mathematical constants, or code optimization.

2 b. Compare primitive data types and derived data types in C, providing examples

Data types define the type of data a variable can store. C provides primitive data types and derived data types, each serving different purposes in handling various kinds of data. Let's compare them and provide examples for each.

1. Primitive Data Types

Primitive data types, also known as basic data types, are built-in data types that represent simple values like numbers, characters, or truth values. These types are directly supported by the C language.

Common Primitive Data Types in C:

1. int (Integer)
   - Description: Represents whole numbers (positive, negative, or zero).
   - Size: Typically 4 bytes (depends on the system).
   - Example:

     ```
     int a = 10; // a is an integer
     ```

2. float (Floating-point number)
   - Description: Represents real numbers (numbers with a fractional part).
   - Size: Typically 4 bytes.
   - Example:

     ```
     float pi = 3.14; // pi is a floating-point number
     ```

3. double (Double-precision floating-point number)
   - Description: Represents real numbers with double precision (more storage and accuracy than float).
   - Size: Typically 8 bytes.
   - Example:

     ```
     double pi = 3.14159265358979; // pi with double precision
     ```

4. char (Character)
   - Description: Represents a single character.
   - Size: Typically 1 byte.
   - Example:

     ```
     char letter = 'A'; // letter is a character
     ```

5. void (No value)

- Description: Represents the absence of any data type, typically used for functions that don't return a value.
- Example:

```
void myFunction() {
   // Function does not return anything
}
```

6. _Bool (Boolean type in C99)

- Description: Represents a Boolean value (either true or false).
- Size: Typically 1 byte.
- Example:

```
_Bool flag = 1; // 1 represents true
```

Summary of Primitive Data Types:

- Integers: int
- Floating-point numbers: float, double
- Characters: char
- Boolean: _Bool (from C99)
- Void: void

2. Derived Data Types

Derived data types are types that are built using the primitive data types. They allow the creation of more complex data structures.

Common Derived Data Types in C:

1. Arrays

- Description: An array is a collection of elements of the same type stored in contiguous memory locations.
- Example:

```
int arr[5] = {1, 2, 3, 4, 5}; // An array of integers
char name[] = "Alice";        // An array of characters (string)
```

2. Pointers

- Description: A pointer is a variable that stores the memory address of another variable.
- Example:

```
int num = 10;
int *ptr = &num;  // ptr is a pointer to an integer
```

3. Structures

- Description: A structure (or struct) is a collection of variables (possibly of different types) grouped together under a single name.
- Example:

```
struct Person {
   char name[20];
   int age;
};
struct Person person1 = {"Alice", 30}; // Declaring and initializing a structure
```

4. Unions

- Description: A union is similar to a structure, but it allows different data types to share the same memory location. A union can store only one of its members at a time.
- Example:

```
union Data {
   int i;
   float f;
   char str[20];
};
union Data data;  // Declaring a union
data.i = 10;      // Only one member can be used at a time
```

5. Enums (Enumerations)

- Description: An enum is a user-defined data type consisting of named integer constants. It helps in organizing related values.
- Example:

```
enum Weekday { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Weekday today = Wednesday;  // today is assigned the value of 3 (since Sunday
#include <stdio.h>


// Structure definition (Derived data type)
struct Person {
   char name[30];  // String (array of characters)
   int age;        // Integer
};
```

```
int main() {
    // Primitive data types
    int num = 10;
    float pi = 3.14;
    char letter = 'A';

    // Derived data types
    struct Person person1;  // Structure
    int arr[3] = {1, 2, 3};  // Array
    int *ptr = &num;         // Pointer

    // Initialize structure
    person1.age = 25;
    snprintf(person1.name, sizeof(person1.name), "Alice");

    // Output the values
    printf("Primitive Data Types:\n");
    printf("num = %d, pi = %.2f, letter = %c\n", num, pi, letter);

    printf("\nDerived Data Types:\n");
    printf("Person Name: %s, Age: %d\n", person1.name, person1.age);
    printf("Array: %d, %d, %d\n", arr[0], arr[1], arr[2]);
    printf("Pointer: The value pointed by ptr is %d\n", *ptr);

    return 0;
}
```

Output:

```
Primitive Data Types:
num = 10, pi = 3.14, letter = A

Derived Data Types:
Person Name: Alice, Age: 25
Array: 1, 2, 3
Pointer: The value pointed by ptr is 10
```

UNIT-II

3 a. Implement a C program using if-else statements to check whether a number is positive or negative.

To check whether a number is positive or negative:

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

```
int main() {
   int num;
   // Ask the user for input
   printf("Enter a number: ");
   scanf("%d", &num);
   // Check if the number is positive or negative
   if (num > 0) {
      printf("The number is positive.\n");
   } else if (num < 0) {
      printf("The number is negative.\n");
   } else {
      printf("The number is zero.\n");
   }
   return 0;
}
Enter a number:  5
The number is positive.
Enter a number:  -3
The number is negative.
Enter a number: 0
The number is zero.
```

3 b. Use a switch statement in C to display the name of a day based on the number input

```
#include <stdio.h>
int main() {
   int dayNumber;
   // Ask the user for input
   printf("Enter a number (1-7) to display the corresponding day of the week: ");
   scanf("%d", &dayNumber);
   // Use switch statement to display the day based on the input
   switch (dayNumber) {
      case 1:
         printf("Sunday\n");
         break;
      case 2:
         printf("Monday\n");
         break;
     case 3:
         printf("Tuesday\n");
         break;
      case 4:
```

```
            printf("Wednesday\n");
            break;
        case 5:
            printf("Thursday\n");
            break;
        case 6:
            printf("Friday\n");
            break;
        case 7:
            printf("Saturday\n");
            break;
        default:
            printf("Invalid input! Please enter a number between 1 and 7.\n");
            break;
    }

    return 0;
}
```

```
Enter a number (1-7) to display the corresponding day of the week:  3
Tuesday
Enter a number (1-7) to display the corresponding day of the week:  6
Friday
Enter a number (1-7) to display the corresponding day of the week:  8
Invalid input! Please enter a number between 1 and 7.
```

OR

4. a Explain the structure of a for loop in C with an example.

Definition "for loop in C"

The for loop in C language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

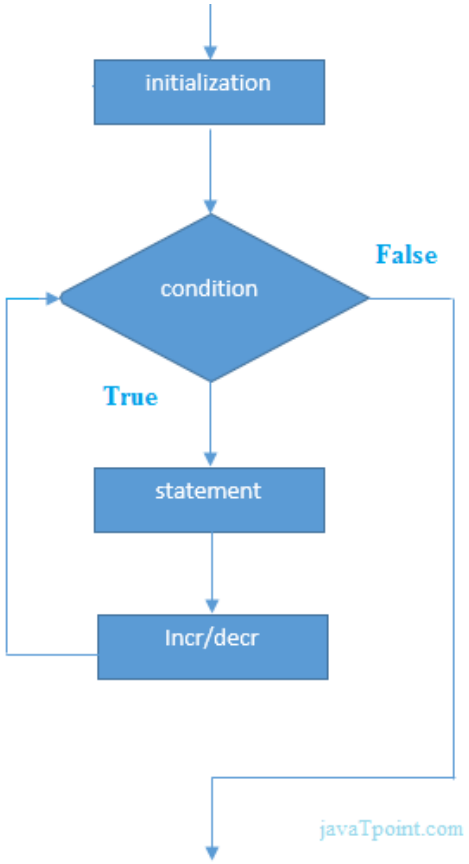
Syntax of for loop in C

The syntax of for loop in c language is given below:

```
for(Expression 1; Expression 2; Expression 3){
//code to be executed
}
```

Flowchart of for loop in C

for loop in c language flowchart



C for loop Example

Let's see the simple program of for loop that prints table of 1.

```
#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=10;i++){
printf("%d \n",i);
}
return 0;
}
```

Output

```
1
2
3
```

4
5
6
7
8
9
10

4 b) Develop a C program that uses a while loop to compute the sum of the first 10 natural numbers

```
#include <stdio.h>
int main() {
   int n=10, I, sum = 0;
   I = 1;
   while (I <= n) {
      sum += I;
      ++I;
   }
   printf("Sum = %d", sum);
   return 0;
}
```

Output:

Sum=55

UNIT-III

5. a. Declare and define a function in C to compute the factorial of a number.

```
#include <stdio.h>
// Function declaration
unsigned long long factorial(int n);

// Main function
int main() {
   int num;
   printf("Enter a number to calculate its factorial: ");
   scanf("%d", &num);
   if (num < 0) {
      printf("Factorial of a negative number is not defined.\n");
   } else {
      printf("Factorial of %d is %llu\n", num, factorial(num));
   }
   return 0;
```

```
}
// Function definition
unsigned long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

5 b. Contrast call by value and call by reference, using examples to demonstrate each.

call by value and **call by reference** are two different ways in which arguments can be passed to functions. Let's contrast them with definitions, examples, and explanations.

1. Call by Value

In **call by value**, the actual value of the argument is passed to the function. This means that any changes made to the parameter inside the function do not affect the original argument in the calling function.

Characteristics of Call by Value:

- The function operates on a copy of the actual argument.
- Changes made to the parameter inside the function do not affect the original variable.
- It is the default method of passing arguments in C.

Example of Call by Value:

```
#include <stdio.h>
// Function to add 10 to the value of the argument
void addTen(int num) {
    num = num + 10;  // Change is made to the local copy
    printf("Inside addTen function: %d\n", num);
}

int main() {
    int x = 5;
    printf("Before calling addTen function: %d\n", x);
    addTen(x);  // Passing x by value
    printf("After calling addTen function: %d\n", x);  // x is unchanged
    return 0;
}
```

Output:

```
Before calling addTen function: 5
Inside addTen function: 15
```

After calling addTen function: 5

2. Call by Reference

In **call by reference**, the address (or reference) of the actual argument is passed to the function. This means that the function can modify the original value of the argument because it operates on the memory location of the argument.

Characteristics of Call by Reference:

- The function works directly with the memory address of the argument.
- Changes made to the parameter inside the function affect the original variable in the calling function.
- Pointers are used to pass arguments by reference in C.

Example of Call by Reference:

```
#include <stdio.h>
// Function to add 10 to the value of the argument using pointers
void addTen(int *num) {
   *num = *num + 10;  // Dereferencing pointer to modify the original value
   printf("Inside addTen function: %d\n", *num);
}
int main() {
   int x = 5;
   printf("Before calling addTen function: %d\n", x);
   addTen(&x);  // Passing the address of x (by reference)
   printf("After calling addTen function: %d\n", x);  // x is modified
   return 0;
}
```

Output:

Before calling addTen function: 5
Inside addTen function: 15
After calling addTen function: 15

Comparison: Call by Value vs Call by Reference

Feature	**Call by Value**	**Call by Reference**
Definition	Passes a copy of the value to the function.	Passes the memory address of the argument.
Modification of Argument	The original argument remains unchanged.	The original argument can be modified.

Feature	Call by Value	Call by Reference
Memory	Uses more memory (for the copy of the argument).	More memory efficient (no copying, works with references).
Performance	May be slower for large data types (since a copy is made).	More efficient for large data types (like large structures) since it avoids copying.
Use Case	Used when you don't want the function to modify the original value.	Used when you want to modify the original value or for efficiency with large data.
Syntax	function(arg)	function(&arg) (using pointers)

OR

6 a. Implement a C program to demonstrate dynamic memory allocation using malloc() and free()

C malloc()
The name "malloc" stands for memory allocation.
The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.
Syntax of malloc()
ptr = (castType*) malloc(size);
Example
ptr = (float*) malloc(100 * sizeof(float));
The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.
The expression results in a NULL pointer if the memory cannot be allocated.

C free()
Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.
Syntax of free()
free(ptr);
This statement frees the space allocated in the memory pointed by ptr.

Example 1: malloc() and free()
// Program to calculate the sum of n numbers entered by the user

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, i, *ptr, sum = 0;
```

```
  printf("Enter number of elements: ");
  scanf("%d", &n);
  ptr = (int*) malloc(n * sizeof(int));
   // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);
    // deallocating the memory
  free(ptr);
  return 0;
}
```

Output
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
Here, we have dynamically allocated the memory for n number of int.

6 b. Describe pointer operations and demonstrate the use of pointers to access and modify variable values

pointers are variables that store the memory address of another variable. Pointers provide a powerful way to manipulate data, pass large structures efficiently, and work with dynamic memory. There are several operations associated with pointers, such as **dereferencing**, **address-of operator**, and pointer arithmetic.

Key Pointer Operations:

1. **Address-of operator (&)**:

- This operator is used to get the memory address of a variable.
- Example: &x gives the memory address of variable x.

2. **Dereferencing operator (*)**:

- This operator is used to access the value stored at the memory address pointed to by a pointer.
- Example: If ptr is a pointer, *ptr accesses the value stored at the memory address ptr is pointing to.

3. **Pointer Arithmetic**:

- Pointers can be incremented or decremented, and pointer arithmetic is useful for traversing arrays or accessing different memory locations.
- Example: ptr++ moves the pointer to the next memory address based on the size of the type it points to.

4. **Pointer to Pointer**:

- You can have pointers that point to other pointers. This is useful for multi-level pointer dereferencing.

Example 1: Using Pointers to Access and Modify Variable Values

```
#include <stdio.h>
int main() {
   int num = 10;      // Initialize an integer variable
   int *ptr = &num;   // Pointer ptr holds the address of num
   // Accessing the value of num using the pointer
   printf("Value of num: %d\n", num);
   printf("Value of num using pointer: %d\n", *ptr);  // Dereferencing the pointer

   // Modifying the value of num using the pointer
   *ptr = 20;  // Dereferencing ptr to change the value of num
   printf("\nAfter modifying using pointer:\n");
   printf("Value of num: %d\n", num);  // num is modified
   printf("Value of num using pointer: %d\n", *ptr);  // The pointer reflects the change
   return 0;
}
```

Output:

```
Value of num: 10
Value of num using pointer: 10
After modifying using pointer:
Value of num: 20
Value of num using pointer: 20
```

UNIT-IV

7 a. Define a structure in C and use it to display the information of a student (name, roll number, marks).

A **structure** is a user-defined data type that allows grouping different types of variables (called members or fields) under one name. Structures are useful for representing a record of data that logically belongs together, such as a student record containing a name, roll number, and marks.

Defining a Structure in C

To define a structure, we use the struct keyword. Each member of the structure can have a different data type, such as int, char[], float, etc.

Example: Structure for Student Information

In this example, we will define a structure to store the student's **name**, **roll number**, and **marks** and use it to display the student information.

C Program:

```
#include <stdio.h>
#include <string.h>
// Define a structure for student information
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};
int main() {
    // Declare a variable of type struct Student
    struct Student student1;
    // Input the student's information
    printf("Enter student's name: ");
    fgets(student1.name, sizeof(student1.name), stdin);  // Read name
    student1.name[strcspn(student1.name, "\n")] = 0;  // Remove trailing newline character
    printf("Enter student's roll number: ");
    scanf("%d", &student1.rollNumber);  // Read roll number
    printf("Enter student's marks: ");
    scanf("%f", &student1.marks);  // Read marks
    // Display the student's information
    printf("\nStudent Information:\n");
    printf("Name: %s\n", student1.name);
    printf("Roll Number: %d\n", student1.rollNumber);
    printf("Marks: %.2f\n", student1.marks);
    return 0;
}
```

Example Output:

```
Enter student's name: John Doe
Enter student's roll number: 101
Enter student's marks: 89.5
Student Information:
Name: John Doe
Roll Number: 101
Marks: 89.50
```

7 b. Implement a union in C and explain how it differs from a structure with an example.

A **union** in C is a special data type that allows storing different data types in the same memory location. Unlike a **structure**, which allocates memory for each member separately, a union shares the same memory for all its members. This means that the size of a union is equal to the size of its largest member, and only one member can hold a value at any given time.

Key Differences Between a Union and a Structure:

Feature	**Union**	**Structure**
Memory Allocation	All members share the same memory location. The size of the union is the size of its largest member.	Each member gets its own memory location, and the size of the structure is the sum of the sizes of all its members.
Storage	Only one member can hold a value at a time.	All members can hold values simultaneously.
Use Case	Used when you need to store different types of data, but only one value at a time.	Used when you need to store multiple values of different types at the same time.

Example of a Union and Structure in C:

Union Example:

```
#include <stdio.h>
// Define a union
union Data {
   int i;
   float f;
   char str[20];
};

int main() {
   // Declare a variable of type union Data
   union Data data;
   // Assign an integer value to the union
   data.i = 42;
   printf("data.i = %d\n", data.i);
   // Assign a float value to the union (this will overwrite the integer value)
   data.f = 3.14;
   printf("data.f = %.2f\n", data.f);
   // Assign a string value to the union (this will overwrite the float value)
   snprintf(data.str, sizeof(data.str), "Hello, Union!");
   printf("data.str = %s\n", data.str);
   // Note: Only the last assigned value is valid, previous values are overwritten
   return 0;
}
```

Output:

```
data.i = 42
```

data.f = 3.14
data.str = Hello, Union!

Structure Example:

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

int main() {
   // Declare a variable of type struct Person
   struct Person person;
   // Assign values to the structure members
   snprintf(person.name, sizeof(person.name), "John Doe");
   person.age = 30;
   person.height = 5.9;
   // Display the structure's information
   printf("Name: %s\n", person.name);
   printf("Age: %d\n", person.age);
   printf("Height: %.2f\n", person.height);
   return 0;
}
```

Output:

Name: John Doe
Age: 30
Height: 5.90

OR

8 a. Define and demonstrate the use of bit-fields in C

Bit-field in C allows you to specify the exact number of bits that should be allocated to a particular member of a structure. This is particularly useful when you need to save memory and represent data compactly, especially when the data takes fewer bits than the usual int or char types.

Bit-fields are primarily used when memory is constrained or when representing data that has a fixed number of bits, such as flags or packed data.

Syntax for Defining Bit-Fields

Bit-fields are defined within a structure, and you specify the number of bits for each field:

```
struct {
   type member_name : number_of_bits;
};
```

- type: The data type of the member (usually int, unsigned int, char, etc.).
- member_name: The name of the member.

- number_of_bits: The number of bits allocated for the member (less than or equal to the size of the type).

Example of Bit-Fields in C

In the example below, we will define a structure that uses bit-fields to store a set of flags and a small number of integers efficiently.

C Program with Bit-Fields:

```
#include <stdio.h>
// Define a structure with bit-fields
struct Flags {
   unsigned int isActive : 1;   // 1 bit for active flag
   unsigned int isValid : 1;    // 1 bit for validity flag
   unsigned int errorCode : 5;  // 5 bits for error code (0-31)
   unsigned int priority : 3;   // 3 bits for priority (0-7)
};
int main() {
   // Declare a variable of type struct Flags
   struct Flags flags = {1, 1, 15, 4};  // Initializing bit-fields
   // Print the values of the bit-fields
   printf("isActive: %u\n", flags.isActive);
   printf("isValid: %u\n", flags.isValid);
   printf("errorCode: %u\n", flags.errorCode);
   printf("priority: %u\n", flags.priority);
   // Modify the values of the bit-fields
   flags.isActive = 0;
   flags.errorCode = 25;  // Error code within the 5-bit limit (0-31)
   // Print the modified values
   printf("\nAfter modification:\n");
   printf("isActive: %u\n", flags.isActive);
   printf("errorCode: %u\n", flags.errorCode);
   return 0;
}
```

Output:

```
isActive: 1
isValid: 1
errorCode: 15
priority: 4
After modification:
isActive: 0
errorCode: 25
```

8 b. Write a C program to access individual elements in an array of structures

An array of structures is a collection of structures stored in contiguous memory locations. To access individual elements in an array of structures, you can use the index of the array along with the dot operator (.) to access the members of the structure.

Example: Accessing Individual Elements in an Array of Structures

Let's create a C program that defines a structure for storing information about students (name, roll number, and marks), then initializes an array of structures, and finally accesses and displays individual elements.

```
#include <stdio.h>
#include <string.h>
// Define a structure to store student information
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};
int main() {
    // Declare an array of 3 Student structures
    struct Student students[3];
    // Initialize the students array
    strcpy(students[0].name, "John Doe");
    students[0].rollNumber = 101;
    students[0].marks = 85.5;
    strcpy(students[1].name, "Jane Smith");
    students[1].rollNumber = 102;
    students[1].marks = 92.3;
    strcpy(students[2].name, "Alice Johnson");
    students[2].rollNumber = 103;
    students[2].marks = 78.4;

    // Access and print individual elements of the array of structures
    for (int i = 0; i < 3; i++) {
        printf("Student %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].rollNumber);
        printf("Marks: %.2f\n\n", students[i].marks);
    }
    return 0;
}
```

Output:

```
Student 1:
Name: John Doe
Roll Number: 101
Marks: 85.50
```

Student 2:
Name: Jane Smith
Roll Number: 102
Marks: 92.30

Student 3:
Name: Alice Johnson
Roll Number: 103
Marks: 78.40

UNIT-V

9 a. List the modes of file operations in C and explain their use with examples.

File operations are performed using standard library functions that allow reading from and writing to files. The file modes define the type of access you have to the file and how the file is opened. These modes are specified when using the fopen() function.

File Operation Modes in C

Here are the different modes used for file operations in C:

1. **"r"** (Read Mode):

- Opens the file for reading.
- The file must exist. If the file doesn't exist, the program will return NULL and set an error.

Example: Open a file for reading:

```
FILE *fp = fopen("file.txt", "r");
if (fp == NULL) {
   printf("File not found.\n");
}
```

2. **"w"** (Write Mode):

- Opens the file for writing.
- If the file exists, it truncates (empties) the file.
- If the file doesn't exist, a new file is created.

Example: Open a file for writing:

```
FILE *fp = fopen("file.txt", "w");
if (fp == NULL) {
   printf("Unable to open the file for writing.\n");
}
```

3. **"a"** (Append Mode):

- Opens the file for appending data at the end.
- If the file doesn't exist, a new file is created.
- Data is written at the end of the file without modifying its existing content.

Example: Open a file for appending:

```
FILE *fp = fopen("file.txt", "a");
if (fp == NULL) {
   printf("Unable to open the file for appending.\n");
}
```

4. **"r+"** (Read/Write Mode):

- Opens the file for both reading and writing.
- The file must exist. If the file doesn't exist, the program returns NULL.

Example: Open a file for reading and writing:

```
FILE *fp = fopen("file.txt", "r+");
if (fp == NULL) {
   printf("File does not exist.\n");
}
```

5. **"w+"** (Write/Read Mode):

- Opens the file for both reading and writing.
- If the file exists, it truncates the file. If the file doesn't exist, a new file is created.

Example: Open a file for reading and writing, with truncation:

```
FILE *fp = fopen("file.txt", "w+");
if (fp == NULL) {
   printf("Unable to open the file for reading and writing.\n");
}
```

6. **"a+"** (Append/Read Mode):

- Opens the file for both reading and appending data at the end.
- If the file doesn't exist, a new file is created.

Example: Open a file for reading and appending:

```
FILE *fp = fopen("file.txt", "a+");
if (fp == NULL) {
   printf("Unable to open the file for reading and appending.\n");
}
```

7. **"b"** (Binary Mode):

- This is not a standalone mode but can be appended to other modes (e.g., "rb", "wb").
- It is used for reading/writing binary files instead of text files.
- In binary mode, data is read or written in its raw binary form without any transformation (such as newline character conversions).

Example: Open a binary file for reading:

```
FILE *fp = fopen("image.bmp", "rb");
if (fp == NULL) {
   printf("Unable to open binary file.\n");
}
```

- Similarly, use "wb" for writing binary files.

Example Program Demonstrating File Operations

Here is a complete example program that demonstrates the use of different file operation modes ("r", "w", "a", "r+").

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

```
#include <stdlib.h>
int main() {
   FILE *fp;
   char data[100];
   // "w" mode: Writing to a file
   fp = fopen("testfile.txt", "w");
   if (fp == NULL) {
      printf("Unable to open the file for writing.\n");
      return 1;
   }
   fprintf(fp, "Hello, world!\n");
   fclose(fp);
   // "r" mode: Reading from the file
   fp = fopen("testfile.txt", "r");
   if (fp == NULL) {
      printf("Unable to open the file for reading.\n");
      return 1;
   }
   while (fgets(data, sizeof(data), fp)) {
      printf("%s", data);  // Print content of file
   }
   fclose(fp);
   // "a" mode: Appending to the file
   fp = fopen("testfile.txt", "a");
   if (fp == NULL) {
      printf("Unable to open the file for appending.\n");
      return 1;
   }
   fprintf(fp, "Appended data.\n");
   fclose(fp);
   // "r+" mode: Reading and writing to the file
   fp = fopen("testfile.txt", "r+");
   if (fp == NULL) {
      printf("Unable to open the file for reading and writing.\n");
      return 1;
   }
   fgets(data, sizeof(data), fp);  // Read first line
   printf("First line read from the file: %s", data);
   fseek(fp, 0, SEEK_END);  // Move to end of file
   fprintf(fp, "End of file reached.\n");
   fclose(fp);
   return 0;
}
```

9 b. Develop a C program that reads from a text file and displays its contents

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *file;      // Declare a file pointer
   char ch;         // Variable to hold each character read from the file
   // Open the file in read mode
   file = fopen("example.txt", "r");
   // Check if the file was opened successfully
   if (file == NULL) {
      // If the file doesn't exist or cannot be opened, print an error and exit
      printf("Error: Could not open the file.\n");
      return 1;
   }
   // Read and display each character from the file
   printf("File contents:\n");
   while ((ch = fgetc(file)) != EOF) {
      putchar(ch);  // Print the character to the console
   }
   // Close the file after reading
   fclose(file);
   return 0;
}
```

Example Output:

Hello, this is a text file.
This is the second line.
End of file.
The output of the program would be:
File contents:
Hello, this is a text file.
This is the second line.
End of file.

OR

10 a. Implement a C program that reads the data from one file and writes it to another file.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *sourceFile, *destinationFile;  // File pointers for source and destination
   char ch;  // Variable to hold each character read from the source file
```

```
    // Open the source file in read mode
    sourceFile = fopen("source.txt", "r");
    if (sourceFile == NULL) {
        // If source file cannot be opened, print an error and exit
        printf("Error: Could not open source file.\n");
        return 1;
    }
    // Open the destination file in write mode
    destinationFile = fopen("destination.txt", "w");
    if (destinationFile == NULL) {
        // If destination file cannot be opened, print an error and exit
        printf("Error: Could not open destination file.\n");
        fclose(sourceFile);  // Close the source file before exiting
        return 1;
    }
    // Read the source file character by character and write to the destination file
    while ((ch = fgetc(sourceFile)) != EOF) {
        fputc(ch, destinationFile);  // Write the character to the destination file
    }
    // Close both the source and destination files
    fclose(sourceFile);
    fclose(destinationFile);
    printf("File content has been copied successfully.\n");
    return 0;
}
```

Output:

Hello, this is the source file.
Content will be copied to the destination file.
The program will copy this content into the destination.txt file. If the content of destination.txt is displayed afterward, it will be:
Hello, this is the source file.
Content will be copied to the destination file.

10 b. Use command-line arguments in a C program to accept two numbers and display their sum

```
#include <stdio.h>
#include <stdlib.h>  // For atoi() function
int main(int argc, char *argv[]) {
    // Check if the correct number of arguments is provided
    if (argc != 3) {
        printf("Usage: %s <number1> <number2>\n", argv[0]);
        return 1;
```

```
    }
    // Convert command-line arguments to integers
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    // Calculate the sum of the two numbers
    int sum = num1 + num2;
    // Display the result
    printf("The sum of %d and %d is: %d\n", num1, num2, sum);
    return 0;
}
```

Output:

If you run the program with 5 and 7 as the input:

The sum of 5 and 7 is: 12