BCA Semester- I

Programming in C (NBCA-102)

Unit II: C Programming Constructs

1. What are the main components of a C program? Describe the structure of a C program with an example.

A C program is structured into different sections, each of which performs a specific role. Understanding the organization of these components helps ensure proper compilation and execution. The main components of a C program include:

- **Preprocessor Directives**:
  - Preprocessor directives are commands that are processed before the actual compilation of the program begins. These usually include `#include` for including standard or user-defined header files, and `#define` for defining macros.
  - Example:

    ```
    #include <stdio.h>  // Includes the standard input/output library
    #define PI 3.14159  // Defines a macro constant
    ```

- **Global Declarations**:
  - Global variables and constants can be declared outside of the `main()` function. These are accessible to all functions in the program.
  - Example:

    ```
    int count = 0;  // Global variable
    ```

- **Main Function**:
  - The `main()` function is the entry point for any C program. This is where program execution begins. Every C program must contain a `main()` function.
  - Example:

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

- **Variable Declarations**:
  - Inside the `main()` function (or any other function), variables are declared to store data temporarily during the program's execution.
  - Example:

    ```
    int number;   // Variable declaration
    ```

- **Statements and Expressions**:
  - C programs consist of statements and expressions that define the logic and operations to be performed. These may include arithmetic operations, conditional statements (`if`, `switch`), loops (`for`, `while`), function calls, etc.
  - Example:

    ```
    number = 5 + 3;  // Expression
    printf("The sum is %d\n", number);  // Statement
    ```

- **Functions**:
  - Functions are self-contained blocks of code that perform specific tasks. A function can be called from the `main()` function or from other functions to execute a certain task.
  - Example:

    ```
    int sum(int a, int b) {
        return a + b;
    }
    ```

- **Return Statement**:
  - The `return` statement ends the function and returns control to the calling function. In the `main()` function, a return value of `0` typically indicates successful program execution.
  - Example:

    ```
    return 0;
    ```

Program Example:

```
#include <stdio.h>   // Preprocessor directive

// Global variable
int global_count = 10;

// Function to calculate the square of a number
int square(int num) {
    return num * num;
}

int main() {
    int num = 5;  // Local variable declaration
    int result;

    // Function call and assignment
    result = square(num);

    // Output the result
    printf("The square of %d is %d\n", num, result);

    return 0;   // End of the program
}
```

2. Explain operator precedence and associativity in C. Why are they important in program execution? Provide examples.

- **Operator precedence** refers to the rules that determine the order in which different operators are evaluated in expressions. For example, in an expression with multiple operators, some operators are given higher priority than others. Operators with higher precedence are evaluated first.
  - **Example**:

    ```
    int result = 5 + 3 * 2;  // result is 11, not 16
    ```

    In this example, multiplication (`*`) has a higher precedence than addition (`+`), so `3 * 2` is evaluated first, and then `5 + 6` is calculated to get the final result.

- **Operator associativity** determines the order in which operators of the same precedence level are evaluated. Associativity can be **left-to-right** or **right-to-left**.
  - **Left-to-right associativity**: Most operators in C (such as `+`, `-`, `*`, `/`) follow left-to-right associativity. This means that when multiple operators of the same precedence appear in an expression, they are evaluated from left to right.
    - **Example**:

      ```
      int result = 10 - 5 + 2;  // result is 7 (evaluated as (10
      - 5) + 2)
      ```

  - **Right-to-left associativity**: Some operators, like the assignment operator (`=`) and the conditional operator (`?:`), have right-to-left associativity.
    - **Example**:

      ```
      int a, b, c;
      a = b = c = 5;  // Right-to-left associativity, evaluated
      as a = (b = (c = 5))
      ```

- **Importance in Program Execution**:
  - **Precedence** ensures that expressions are evaluated in a logical and predictable manner. Without precedence, expressions like `3 + 5 * 2` could produce incorrect results if the operators were evaluated strictly left-to-right without regard for the priority of multiplication over addition.
  - **Associativity** is crucial when dealing with operators of the same precedence. For example, consider the expression `5 - 3 + 2`. If both subtraction and addition had no associativity rules, the result could be ambiguous.
  - Incorrect assumptions about precedence or associativity can lead to bugs or unintended behavior in programs.
- **Precedence and Associativity Table**:

Precedence	Operator	Associativity
1	`() [] -> .`	left-to-right
2	`! ~ ++ -- + -`	right-to-left
3	`* / %`	left-to-right

4	`+ -`	left-to-right
5	`<< >>`	left-to-right
6	`< <= > >=`	left-to-right
7	`== !=`	left-to-right
8	`&`	left-to-right
9	`^`	left-to-right
10	`|`	left-to-right
11	`&&`	left-to-right
12	`||`	left-to-right
13	`?:`	right-to-left
14	`= += -=`	right-to-left

3. What are storage classes in C? Discuss the different types and give an example for each storage class (automatic, register, static, and external).

In C, a **storage class** defines the scope (visibility), lifetime (duration of existence), and memory location of variables. There are four types of storage classes:

1. **Automatic Storage Class (`auto`)**:
   - The `auto` storage class is the default for local variables inside functions. Variables with the `auto` class have **block scope**, meaning they are created when the block (function) starts and destroyed when it ends.
   - **Example**:

     ```
     void func() {
         auto int x = 10;  // This variable 'x' is local to 'func'
         printf("%d\n", x);
     }
     ```

   - **Characteristics**:
     - Storage: **Memory**
     - Lifetime: Created when the function is called and destroyed when it exits.
     - Scope: Local to the block in which it is defined.
2. **Register Storage Class (`register`)**:
   - The `register` storage class requests that the variable be stored in a **CPU register** rather than in memory, making access to the variable faster. However, the actual placement of the variable in a register is not guaranteed and depends on the hardware.
   - **Example**:

     ```
     void func() {
         register int counter = 0;  // Request to store 'counter' in a
     CPU register
         counter++;
         printf("%d\n", counter);
     }
     ```

   - **Characteristics**:
     - Storage: **CPU register**
     - Lifetime: Same as `auto` (local variables)

        - Scope: Local to the block.
3. **Static Storage Class (`static`)**:
    - The `static` storage class extends the **lifetime** of a variable to the entire program. This means that the variable is created when the program starts and destroyed when it ends, but it retains **block scope** (i.e., it is only accessible within the function or block where it is declared). A static variable preserves its value between function calls.
    - **Example**:

```
void count() {
    static int x = 0;  // Static variable
    x++;
    printf("%d\n", x);
}
int main() {
    count();  // Output: 1
    count();  // Output: 2 (because x retains its value)
    return 0;
}
```

    - **Characteristics**:
        - Storage: **Memory**
        - Lifetime: Throughout the program's execution.
        - Scope: Local to the block, but retains value between function calls.
4. **External Storage Class (`extern`)**:
    - The `extern` storage class is used to declare a global variable or function in another file or later in the same file. It tells the compiler that the variable or function exists and is defined elsewhere. Variables declared with `extern` have **global scope** and can be accessed across different files in a multi-file program.
    - **Example**:

```
extern int global_var;  // Declares a global variable defined
elsewhere
```

    If `global_var` is defined in another file, it can be accessed using `extern`.

    - **Characteristics**:
        - Storage: **Memory**
        - Lifetime: Throughout the program's execution.
        - Scope: Global, accessible across multiple files.

4. Write a C program that demonstrates the use of `printf()` and `scanf()` for input and output. Explain how these functions work.

In C, **`printf()`** is used to display output to the console, and **`scanf()`** is used to read input from the user. These functions are part of the standard input/output library (`stdio.h`).

- **`printf()`**:

    - `printf()` is used to print formatted output. It takes a format string that specifies how the output should be displayed, followed by the values to print.
    - Example format specifiers:
        - `%d`: Integer
        - `%f`: Floating-point number
        - `%c`: Character
        - `%s`: String
        - `%.2f`: Floating-point number with 2 decimal places
- **`scanf()`**:
    - `scanf()` is used to read input from the user. It requires the format specifier for the type of input expected and the **address** of the variable where the input should be stored. The address operator (`&`) is used to pass the address of the variable.
- **Example Program**:

```
#include <stdio.h>

int main() {
    int age;
    float salary;

    // Taking input from the user
    printf("Enter your age: ");
    scanf("%d", &age);  // Reads an integer value into 'age'

    printf("Enter your salary: ");
    scanf("%f", &salary);  // Reads a float value into 'salary'

    // Displaying the output
    printf("Your age is %d and your salary is %.2f\n", age, salary);

    return 0;
}
```

- **Explanation**:
    1. **`printf("Enter your age: ");`**: This prints the string "Enter your age: " to the console.
    2. **`scanf("%d", &age);`**: This waits for the user to enter an integer value, which is then stored in the variable `age`. The `%d` format specifier tells `scanf()` to expect an integer. The `&age` passes the **address** of the `age` variable, so `scanf()` knows where to store the input.
    3. **`printf("Your age is %d and your salary is %.2f", age, salary);`**: This prints the values of `age` and `salary` to the console. `%d` is used for the integer `age`, and `%.2f` is used for the floating-point `salary`, rounded to two decimal places.

This program demonstrates the use of both `printf()` and `scanf()` for basic input/output operations in C.