BCA-I Semester (NEP-2020)

NBCA-102 (Programing in C)

Important Questions

UNIT-1

Q1. Explain the character set and keywords in C language. How are keywords different from identifiers? Discuss the importance of keywords in programming with examples.

Ans1. The character set in C consists of the fundamental building blocks used to form programs, such as letters, digits, and special characters. The C language uses the ASCII character set, which includes:

- **Alphabets:** Uppercase (A-Z) and lowercase (a-z).
- **Digits:** 0-9.
- **Special characters:** +, -, *, /, =, &, #, etc.
- **Whitespace characters:** Space, newline, tab.
- **Control characters:** These are non-printing characters such as \n (newline), \t (tab), etc.

Keywords in C Language:

Keywords are reserved words predefined by the C language, having fixed meanings and purposes. They are fundamental to C syntax and cannot be used as identifiers (such as variable names or function names). Some common C keywords are: int, char, float, if, else, while, for, return, void, and const.

Difference between Keywords and Identifiers:

- **Keywords:** Predefined and reserved for specific purposes in the C language. They cannot be altered or redefined. Example: int, void, if.
- **Identifiers:** Names created by the programmer to represent variables, functions, or other user-defined items. Identifiers must follow naming rules, and they cannot be keywords. Example: sum, counter, calculateTotal.

Importance of Keywords in Programming:

Keywords form the backbone of the syntax and structure of a program. They are essential for defining data types, control flow, functions, and memory management. For example:

- `int main()` defines the entry point of a program.
- `if` and `else` control conditional execution.
- `return` indicates the result of a function.

Keywords ensure that programs are structured and interpreted correctly by the compiler, allowing for precise instructions and efficient execution.

Q2. What is a variable in C? Explain the declaration and initialization of variables with suitable examples. How does the scope of a variable affect its behavior in a program? Illustrate the difference between local and global scope with code.

Ans2. Variable in C:

A variable in C is a storage location in memory that holds a value. Variables have a specific data type, which determines the type of data they can store, such as `int`, `float`, or `char`. Each variable must have a name (identifier) to reference it.

Declaration and Initialization of Variables:

- **Declaration** is the process of specifying the type and name of a variable. Example:

  ```
  int age;  // Declaration of an integer variable named age
  ```

- **Initialization** assigns an initial value to a declared variable. Example:

  ```
  int age = 25;  // Declaration and initialization in a single statement
  ```

A variable can also be initialized after declaration:

```
int age;
age = 25;  // Initializing the variable after declaration
```

Scope of a Variable:

The scope of a variable defines the region in a program where the variable can be accessed. Variables can have either **local scope** or **global scope**.

- **Local Scope:** A variable declared inside a function or block (within {}) has a local scope and can only be accessed within that function or block.

- **Global Scope:** A variable declared outside all functions has a global scope and can be accessed by any function within the program.

Local vs. Global Scope:

Example: Local Scope

```
#include <stdio.h>

void function() {
   int x = 10;  // Local variable
   printf("Value of x inside function: %d\n", x);
}

int main() {
   function();
   // printf("%d", x);  // This will cause an error because x is not in scope here
   return 0;
}
```

In this example, x is a local variable to function(), so it cannot be accessed outside the function.

Example: Global Scope

```
#include <stdio.h>

int x = 10;  // Global variable

void function() {
   printf("Value of x inside function: %d\n", x);
}

int main() {
   printf("Value of x in main: %d\n", x);
   function();
   return 0;
}
```

Here, x is a global variable and can be accessed in both the main() and function() functions.

Difference:

- **Local variables** exist only within the block where they are declared and are destroyed after the block is executed.
- **Global variables** exist throughout the program's execution, making them accessible to all functions. However, excessive use of global variables can lead to tightly coupled code, making debugging and maintenance difficult.

Q3. Define constants in C. Discuss the different types of constants available in C programming (integer, floating-point, character, string). How do constants differ from variables? Write a program that demonstrates the use of different constants in C.

Ans3. Constants in C:

A constant in C refers to a value that cannot be altered during the execution of a program. Constants are fixed values that are defined and assigned during program compilation. They help maintain the integrity of data that shouldn't change throughout the program.

Types of Constants in C:

1. **Integer Constants:**
    - These represent whole numbers without any fractional part.
    - They can be in decimal, octal, or hexadecimal form.
    - Examples: 10, 0xA, 014
2. **Floating-point Constants:**
    - These represent real numbers that include a fractional part.
    - Examples: 3.14, -0.5, 2.0E3
3. **Character Constants:**
    - A single character enclosed in single quotes.
    - It represents the ASCII value of the character.
    - Example: 'A', '9'
4. **String Constants:**
    - A sequence of characters enclosed in double quotes.
    - It represents a string of characters stored as an array.
    - Example: "Hello", "123"

Difference between Constants and Variables:

- **Constants:** Their values remain unchanged during the execution of the program.
- **Variables:** Their values can change during program execution.

Program Demonstrating Different Constants in C:

```
#include <stdio.h>

#define PI 3.14159  // Constant defined using #define (macro)

int main() {
   const int AGE = 25;  // Integer constant using 'const'
```

```
    const float RATE = 4.5;  // Floating-point constant using 'const'
    const char GRADE = 'A';  // Character constant using 'const'
    const char NAME[] = "John";  // String constant using 'const'

    // Demonstrating the use of constants
    printf("Value of PI: %f\n", PI);  // Using macro constant
    printf("Age: %d\n", AGE);  // Integer constant
    printf("Rate: %f\n", RATE);  // Floating-point constant
    printf("Grade: %c\n", GRADE);  // Character constant
    printf("Name: %s\n", NAME);  // String constant

    // Trying to modify a constant would result in a compilation error
    // AGE = 30;  // Uncommenting this line will cause an error

    return 0;
}
```

Q4. Discuss the various data types supported in C language. How do data types ensure efficient memory usage and proper data handling in a C program? Give examples of each data type along with their sizes and usage.

Ans4. Data Types in C Language:

C supports a variety of data types that define the kind of data a variable can hold. These data types ensure efficient memory usage and proper data handling by allocating the appropriate amount of memory for the data type and determining the operations that can be performed on the data.

Primary Data Types:

1. **Integer (int):**
    - Used to store whole numbers.
    - Size: Typically 2 or 4 bytes (platform-dependent).
    - Example: `int count = 10;  // Integer variable`
    - Range: -32,768 to 32,767 (for 2 bytes) or -2,147,483,648 to 2,147,483,647 (for 4 bytes).
2. **Character (char):**
    - Used to store a single character.
    - Size: 1 byte.
    - Example: `char letter = 'A';  // Character variable`
    - Range: -128 to 127 (signed), 0 to 255 (unsigned).
3. **Floating-point (float):**
    - Used to store real numbers with decimal points.
    - Size: 4 bytes.
    - Example: `float temperature = 36.6;  // Floating-point variable`
    - Precision: 6-7 decimal digits.

4. **Double (double):**
    - Used to store large real numbers with more precision.
    - Size: 8 bytes.
    - Example: double pi = 3.14159265359; // Double variable
    - Precision: 15-16 decimal digits.
5. **Void (void):**
    - Used as a return type for functions that do not return a value.
    - Example:

```
void display() {

printf("Hello World");

}
```

Derived Data Types:

1. **Array:**
    - Collection of data elements of the same type.
    - Example: int numbers[5] = {1, 2, 3, 4, 5}; // Array of integers
2. **Pointer:**
    - Stores the address of a variable.
    - Example: int *ptr; // Pointer to an integer
3. **Structure (struct):**
    - Combines different types of data.
    - Example: struct Person {

```
  char name[20];
  int age;
};
```

4. **Union (union):**
    - Similar to structure but uses shared memory for all members.
    - Example: union Data {

```
  int i;
  float f;
};
```

5. **Enumerated (enum):**
    - Defines a set of named integer constants.
    - Example: enum week { Sunday, Monday, Tuesday };

Sizes and Usages of Each Data Type:

Data Type	Size (bytes)	Example Usage	Description
int	2 or 4	int age = 25;	Used to store integers (whole numbers).
char	1	char grade = 'A';	Used to store characters.
float	4	float weight = 55.5;	Used to store single-precision floating-point numbers.
double	8	double distance = 12345.6789;	Used to store double-precision floating-point numbers.
void	0	void functionName();	Represents an empty value, often used in functions.
short int	2	short int count = 100;	Used to store smaller range integers.
long int	4 or 8	long int population = 123456789;	Used for large integer values.
long double	10 or 12	long double pi = 3.141592653589;	Used for extended precision floating-point numbers.

Efficient Memory Usage and Proper Data Handling:

- **Memory Allocation:** C assigns different amounts of memory to different data types. For example, an int uses 2 or 4 bytes, while a char uses only 1 byte. Choosing the correct data type ensures that memory is not wasted. For instance, using char for storing characters saves space compared to using int.
- **Operations and Type Safety:** Data types determine the kind of operations that can be performed on variables. For example, arithmetic operations are valid for int and float, but not for char. This ensures proper data handling and reduces errors.

Example Program Demonstrating Different Data Types:

```
#include <stdio.h>

int main() {
    int age = 30;              // Integer data type
    char grade = 'A';          // Character data type
    float temperature = 98.6;  // Float data type
    double pi = 3.14159;       // Double data type

    printf("Age: %d\n", age);                  // %d for integer
    printf("Grade: %c\n", grade);              // %c for character
    printf("Temperature: %.1f\n", temperature); // %f for float
    printf("Value of Pi: %lf\n", pi);          // %lf for double

    return 0;
}
```

Output:

Age: 30
Grade: A
Temperature: 98.6
Value of Pi: 3.141590

Q5. Explain the different types of operators in C. Provide examples of unary, binary, and ternary operators. Discuss the significance of operator precedence and associativity in evaluating complex expressions.

Ans5. Operators in C:

Operators in C are symbols that perform operations on variables and values. They are categorized based on the number of operands they operate on and the types of operations they perform.

Types of Operators in C:

1. **Unary Operators**: Operate on a single operand.
2. **Binary Operators**: Operate on two operands.
3. **Ternary Operator**: Operates on three operands.

1. Unary Operators:

Unary operators act on a single operand. Common unary operators in C include:

- **Increment (++)**: Increases the value of a variable by 1.
- **Decrement (--)**: Decreases the value of a variable by 1.
- **Logical NOT (!)**: Reverses the logical state of an operand.
- **Bitwise NOT (~)**: Inverts all bits of the operand.
- **Unary minus (-)**: Negates the value of an operand.
- **Unary plus (+)**: Has no effect but indicates positivity.
- **Address-of (&)**: Returns the memory address of a variable.
- **Dereference (*)**: Accesses the value at the address stored in a pointer.

Example:

```
int x = 5;
int y = -x;      // Unary minus
x++;            // Increment operator
int z = !0;      // Logical NOT
```

2. Binary Operators:

Binary operators act on two operands. They are categorized into different types:

- **Arithmetic Operators**: +, -, *, /, % (modulus).

```
int a = 10, b = 3;
int sum = a + b;     // Addition
int diff = a - b;    // Subtraction
int prod = a * b;    // Multiplication
int div = a / b;     // Division
int mod = a % b;     // Modulus
```

- **Relational Operators**: >, <, >=, <=, ==, !=.

```
if (a > b) { /* do something */ }  // Greater than
```

- **Logical Operators**: && (AND), || (OR).

```
if (a > 0 && b > 0) { /* Both true */ }  // Logical AND
```

- **Bitwise Operators**: &, |, ^, <<, >>.

```
int bitwise And = a & b;  // Bitwise AND
```

- **Assignment Operators**: =, +=, -=, *=, /=, %=.
- **Conditional Operators**: Used in ternary operations.

3. Ternary Operator (?:):

The ternary operator is the only operator in C that operates on three operands. It is used for making decisions in a concise way.

- **Syntax:** condition ? expression1 : expression2
    - If the condition is true, expression1 is executed.
    - If false, expression2 is executed.

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // If a > b, max = a; otherwise, max = b
```

Significance of Operator Precedence and Associativity:

In C, operator precedence determines the order in which operators are evaluated in an expression. Associativity determines the direction (left-to-right or right-to-left) in which operators of the same precedence are evaluated.

Precedence Rules:

- **High precedence operators** are evaluated before lower precedence ones.
- **Associativity** resolves ties when multiple operators of the same precedence appear in an expression.

Example of Precedence:

```
int result = 5 + 3 * 2;  // result = 11
```

Here, multiplication (*) has higher precedence than addition (+), so 3 * 2 is evaluated first, followed by 5 + 6.

Associativity:

- **Left-to-right associativity**: Operators like +, -, *, /, % follow left-to-right evaluation.
- **Right-to-left associativity**: Operators like = (assignment) and ? : (ternary) follow right-to-left evaluation.

Example of Associativity:

```
int a = 5;
int b = 10;
int result = (a > 3 && b < 20) ? a : b;  // Right-to-left associativity for ternary operator
```

Complex Expression with Precedence and Associativity:

```
int a = 10, b = 20, c = 5;
int result = a + b * c / 2;  // result = 10 + (20 * 5) / 2 = 10 + 100 / 2 = 10 + 50 = 60
```

Q6. What are bitwise operators in C? Explain the working of bitwise AND, OR, XOR, NOT, left shift, and right shift operators with examples. How are bitwise operations useful in systems programming?

Ans6. Bitwise Operators in C:

Bitwise operators in C are used to perform operations at the bit level. They manipulate individual bits of integer data types (int, char, etc.) and are often employed in systems programming, embedded systems and low-level hardware manipulation where performance and memory efficiency are critical.

Types of Bitwise Operators:

1. **Bitwise AND (&)**
2. **Bitwise OR (|)**
3. **Bitwise XOR (^)**
4. **Bitwise NOT (~)**

5. **Left Shift (<<)**
6. **Right Shift (>>)**

1. Bitwise AND (&):

The bitwise AND operator compares each bit of its two operands. If both corresponding bits are 1, the result is 1; otherwise, the result is 0.

Example:

```
int a = 5;    // In binary: 0101
int b = 3;    // In binary: 0011
int result = a & b;  // 0101 & 0011 = 0001 (binary), result = 1
```

2. Bitwise OR (|):

The bitwise OR operator compares each bit of its two operands. If at least one of the corresponding bits is 1, the result is 1; otherwise, the result is 0.

Example:

```
int a = 5;    // In binary: 0101
int b = 3;    // In binary: 0011
int result = a | b;  // 0101 | 0011 = 0111 (binary), result = 7
```

3. Bitwise XOR (^):

The bitwise XOR (exclusive OR) operator compares each bit of its two operands. The result is 1 if the corresponding bits are different; if the bits are the same, the result is 0.

Example:

```
int a = 5;    // In binary: 0101
int b = 3;    // In binary: 0011
int result = a ^ b;  // 0101 ^ 0011 = 0110 (binary), result = 6
```

4. Bitwise NOT (~):

The bitwise NOT operator inverts all the bits of its operand. Each 1 becomes 0, and each 0 becomes 1. This is also known as the one's complement.

Example:

```
int a = 5;    // In binary: 0101
int result = ~a;  // ~0101 = 1010 (binary), result = -6 (in two's complement representation)
```

Note: The result is negative because C uses two's complement to represent negative numbers.

5. Left Shift (<<):

The left shift operator shifts all the bits of its first operand to the left by the number of positions specified by the second operand. Zeros are shifted in from the right.

Example:

```
int a = 5;    // In binary: 0101
int result = a << 1;  // 0101 becomes 1010 (binary), result = 10
```

Effect: Left shifting by n bits multiplies the number by 2^n.

6. Right Shift (>>):

The right shift operator shifts all the bits of its first operand to the right by the number of positions specified by the second operand. Zeros are shifted in from the left for unsigned numbers; for signed numbers, the behavior depends on the implementation (either logical shift or arithmetic shift).

Example:

```
int a = 5;    // In binary: 0101
int result = a >> 1;  // 0101 becomes 0010 (binary), result = 2
```

Effect: Right shifting by n bits divides the number by 2^n (truncating towards zero for positive numbers).

Usage of Bitwise Operations in Systems Programming:

Bitwise operations are crucial in systems programming for tasks that require low-level manipulation of data at the hardware or binary level. Some common use cases include:

1. **Manipulating Individual Bits:** Bitwise operations allow precise control over individual bits in data, enabling operations such as setting, clearing, or toggling specific bits. This is useful in scenarios like controlling hardware devices, setting configuration registers, or managing status flags.
    - **Example:** Setting a specific bit to 1:

      ```
      int x = 0b00000100;  // Binary representation: 00000100
      x = x | (1 << 2);    // Sets the 2nd bit, result = 00000100 (no change)
      x = x | (1 << 3);    // Sets the 3rd bit, result = 00001100
      ```

2. **Optimized Memory Usage:** Bitwise operations allow the use of a single variable to store multiple Boolean flags, saving memory space.
    - **Example:** Using an integer to store multiple on/off states for different flags:

      ```
      unsigned int flags = 0;  // All flags off
      ```

```
flags |= (1 << 0);      // Set flag 0
flags |= (1 << 3);      // Set flag 3
```

3. **Efficient Arithmetic Operations:** Bitwise shifts can be used as efficient substitutes for multiplication or division by powers of two, avoiding expensive division or multiplication operations in performance-critical code.
4. **Networking and Data Transmission:** Bitwise operations are essential for tasks like constructing and parsing data packets, bit-masking, and working with binary protocols or hardware interfaces.
5. **Cryptography and Error Detection:** Bitwise XOR is commonly used in cryptographic algorithms and for parity checks in error detection schemes.

Example Program Demonstrating Bitwise Operators:

```
#include <stdio.h>

int main() {
   int a = 5, b = 3;

   // Bitwise AND
   printf("Bitwise AND of %d & %d = %d\n", a, b, a & b);

   // Bitwise OR
   printf("Bitwise OR of %d | %d = %d\n", a, b, a | b);

   // Bitwise XOR
   printf("Bitwise XOR of %d ^ %d = %d\n", a, b, a ^ b);

   // Bitwise NOT
   printf("Bitwise NOT of %d = %d\n", a, ~a);

   // Left shift
   printf("Left shift of %d << 1 = %d\n", a, a << 1);

   // Right shift
   printf("Right shift of %d >> 1 = %d\n", a, a >> 1);

   return 0;
}
```

Output:

```
Bitwise AND of 5 & 3 = 1
Bitwise OR of 5 | 3 = 7
Bitwise XOR of 5 ^ 3 = 6
Bitwise NOT of 5 = -6
Left shift of 5 << 1 = 10
Right shift of 5 >> 1 = 2
```

Q7. Describe type conversion and typecasting in C. Differentiate between implicit and explicit typecasting with examples. How does typecasting affect the output of a program? Write a program to demonstrate both implicit and explicit typecasting.

Ans7. Type Conversion and Typecasting in C

Type conversion in C refers to converting the data type of a variable or expression from one type to another. This process can either happen automatically or be forced by the programmer.

- **Type Conversion**: The process of converting one data type to another.
- **Typecasting**: Forcing the conversion of a variable from one type to another.

There are two types of typecasting in C:

1. **Implicit Type Conversion** (Automatic Type Conversion)
2. **Explicit Type Conversion** (Manual Typecasting)

1. Implicit Type Conversion (Automatic Typecasting):

In implicit type conversion, the compiler automatically converts one data type to another when required. This usually happens when:

- Data of a smaller type is assigned to a larger type.
- Mixing different data types in expressions (e.g., adding an int to a float).

The conversion is done based on type hierarchy, where types like float, double, and long are considered "bigger" than int, char, etc.

Example of Implicit Conversion:

```
int a = 5;
float b = 2.5;
float result = a + b;  // 'a' is implicitly converted to float before addition
```

In this example, a (an integer) is automatically converted to float before adding to b, resulting in a float sum.

2. Explicit Type Conversion (Manual Typecasting):

Explicit typecasting is done manually by the programmer using the cast operator. It involves converting one data type to another forcibly, even when there might be a loss of data.

- **Syntax:**

  (type) expression

  The cast operator (type) specifies the data type to which you want to convert.

Example of Explicit Typecasting:

```
float x = 9.7;
int y = (int) x;  // Explicit typecasting from float to int, truncating decimal part
```

Here, x (a float) is explicitly cast to an int, truncating the decimal part and leaving y = 9.

Difference between Implicit and Explicit Typecasting:

Aspect	Implicit Typecasting	Explicit Typecasting
Performed By	Compiler automatically	Programmer manually (using cast operator)
Data Loss	Usually safe, no data loss	May lead to data loss (e.g., truncating decimal)
Ease of Use	Easier, automatic	Requires careful handling by the programmer
Example	int a = 10; float b = a;	float a = 10.5; int b = (int) a;

How Typecasting Affects Program Output:

Typecasting influences how data is stored and manipulated within a program. For instance:

- Implicit typecasting ensures no data loss but may promote smaller data types to larger ones.
- Explicit typecasting allows flexibility but can lead to data truncation or precision loss.

Example Program Demonstrating both Implicit and Explicit Typecasting:

```
#include <stdio.h>

int main() {
   // Implicit typecasting example
   int a = 5;
   float b = 4.5;
   float result_implicit = a + b; // 'a' is implicitly converted to float
   printf("Result of implicit typecasting (int to float): %.2f\n", result_implicit);

   // Explicit typecasting example
   float x = 9.8;
   int y = (int) x;  // Explicit typecasting from float to int (truncating)
   printf("Result of explicit typecasting (float to int): %d\n", y);

   // Mixing explicit and implicit typecasting
   int m = 10;
   int n = 3;
   float result_division = (float)m / n;  // Explicit typecasting of 'm' to float
   printf("Result of explicit typecasting in division: %.2f\n", result_division);

   return 0;
}
```

Output:

```
Result of implicit typecasting (int to float): 9.50
Result of explicit typecasting (float to int): 9
Result of explicit typecasting in division: 3.33
```

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.