UNIT - 1

Compilers, Compiling, and Executing a Program

1. Introduction to Compilers

A **compiler** is a specialized program that translates source code written in a high-level programming language into machine code, which is the set of instructions a computer's processor can execute directly. The compilation process is vital for converting human-readable code into a form that a computer can understand.

There are many C compilers available like GCC(GNU Compiler Collection), Intel C++ compiler etc. The GNU Compiler Collection (*GCC Compiler*) is used for compilation of programs written in C Language.

Compilation: The compilation is the process of converting an understandable human code into a Machine understandable code and checking the syntax, semantics of the code to determine any syntactical errors or warnings present in the C program.

2. Phases of Compilation

- **Lexical Analysis**: The source code is broken down into tokens, which are the smallest units of meaning (e.g., keywords, operators).
- **Syntax Analysis**: The structure of the code is checked against the grammar rules of the programming language.
- **Semantic Analysis**: The meaning of the code is evaluated to ensure logical consistency (e.g., variable type checking).
- **Intermediate Code Generation**: A lower-level, abstract representation of the source code is generated, which is easier to optimize.
- **Optimization**: The intermediate code is optimized for performance, such as reducing the number of instructions or improving execution speed.

- **Code Generation**: The final machine code is generated from the optimized intermediate code.
- **Linking**: The machine code is linked with other code libraries to produce an executable program.

3. Types of Compilers

- **Single-Pass Compiler**: Processes the source code in one pass.
- **Multi-Pass Compiler**: Processes the source code in multiple passes to improve error checking and optimization.
- **Cross Compiler**: Generates machine code for a platform different from the one on which the compiler is running.
- **Just-In-Time (JIT) Compiler**: Compiles code during execution rather than before execution, commonly used in environments like Java Virtual Machine (JVM).

Compiling and Executing a Program

- **Writing the Program**: The source code is written using a text editor or an Integrated Development Environment (IDE).
- **Compilation**: The source code file (e.g., program.c) is passed to the compiler, which translates it into machine code.
- **Linking**: The compiler links the machine code with external libraries to produce an executable file (e.g., program.exe).
- **Loading and Execution**: The operating system loads the executable into memory and starts execution.
- **Debugging**: If the program doesn't run as expected, debugging tools are used to identify and fix errors, followed by recompilation.

Representation of Algorithm:

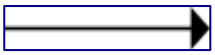
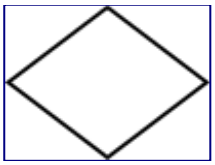
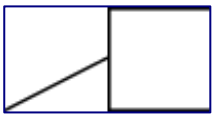
An **algorithm** is a step-by-step procedure or formula for solving a problem. Representing an algorithm clearly is crucial for implementing it in code. It takes a set of input(s) and produces the desired output.

Qualities of a Good Algorithm

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Flowchart:

A flowchart is a diagram that illustrates the steps, sequences, and decisions of a process or workflow.

Shape	Name	Description
	Flowline (Arrowhead)	Shows the process's order of operation. A line coming from one symbol and pointing at another.
	Terminal	Indicates the beginning and ending of a program or sub-process. They usually contain the word "Start" or "End". Represented as oval or rounded rectangle
	Process	Represents a set of operations that changes value. Represented as rectangle.
	Decision	Shows a conditional operation that determines which one of the two paths the program will take. The operation is commonly a yes/no question or true/false test. Represented as a diamond (rhombus)
	Input/Output	Indicates the process of inputting and outputting data, as in entering data or displaying results. Represented as parallelogram.
	Annotation (Comment)	Indicating additional information about a step in the program. Represented as an open rectangle with a dashed or solid line connecting it to the corresponding symbol in the flowchart.
	Predefined Process	Shows named process which is defined elsewhere. Represented as a rectangle with double-struck vertical edges.
	On-page Connector	Pairs of labeled connectors replace long or confusing lines on a flowchart page. Represented by a small circle with a letter inside.

Off-page Connector	A labelled connector for use when the target is on another page. Represented as pentagon.

Pseudocode:

Pseudocode is an informal way of writing algorithms or programming logic that combines natural language with some programming language-like constructs. It is not an actual programming language but serves as a bridge between human understanding and code implementation. By using pseudocode, programmers can outline the steps of an algorithm clearly before converting them into actual code.

Advantages of Pseudocode

1. **Eases Transition to Code**: Provides a clear and structured outline that can be easily translated into actual code.
2. **Improves Collaboration**: Makes it easier for team members with different levels of programming expertise to understand and contribute to the development process.
3. **Reduces Errors**: Helps in identifying potential issues or logical errors early in the design phase.

Example:

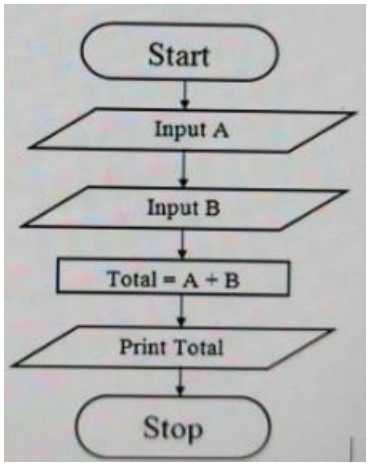
a. Algorithm to Find the Sum of Two Numbers

This is one of the simplest algorithms that introduces the concept of taking input, performing a calculation, and providing output.

Algorithm:

1. Start
2. Input two numbers, a and b.
3. Compute the sum: sum = a + b.
4. Output the sum.
5. End

Flowchart:



Pseudocode:

START

INPUT a, b

sum ← a + b

PRINT sum

END

b. Algorithm to Find the Maximum of Two Numbers

This algorithm teaches comparison operations and decision-making.

Algorithm:

1. Start
2. Input two numbers, a and b.
3. If a is greater than b, then output a is the maximum number.
4. Else output b is the maximum number.
5. End

Pseudocode:

START

INPUT a, b

```
        IF a > b THEN
            PRINT a
        ELSE
            PRINT b
        ENDIF
    END
```

c. Algorithm to Check if a Number is Even or Odd

Algorithm:

1. Start
2. Input a number n.
3. Compute remainder = n % 2.
4. If remainder is equal to 0, then the number is "Even".
5. Else the number is "Odd".
6. End

Pseudocode:

```
    START
        INPUT n
        remainder ← n % 2
        IF remainder = 0 THEN
            PRINT "Even"
        ELSE
            PRINT "Odd"
        ENDIF
    END
```

1. Finding Roots of a Quadratic Equation

The quadratic equation $ax^2 + bx + c = 0$ has roots that can be found using the formula:

- **Discriminant**: $D = b^2 - 4ac$
- **Algorithm**:
  1. Calculate D.
  2. If $D > 0$, the roots are real and different:

$$\text{root}_1 = \frac{-b + \sqrt{D}}{2a}, \quad \text{root}_2 = \frac{-b - \sqrt{D}}{2a}$$

  3. If $D = 0$, the roots are real and the same:

$$\text{root} = \frac{-b}{2a}$$

  4. If $D < 0$, the roots are complex:

$$\text{root}_1 = \frac{-b}{2a} + i\frac{\sqrt{-D}}{2a}, \quad \text{root}_2 = \frac{-b}{2a} - i\frac{\sqrt{-D}}{2a}$$

Flowchart/Pseudocode:

1. Input values for a, b, and c.
2. Compute the discriminant D = b^2 - 4ac.
3. If D > 0:
   a. The roots are real and different.
   b. Calculate root1 = (-b + sqrt(D)) / (2 * a).
   c. Calculate root2 = (-b - sqrt(D)) / (2 * a).
4. Else if D == 0:
   a. The roots are real and the same.
   b. Calculate root = -b / (2 * a).
5. Else (D < 0):
   a. The roots are complex.
   b. Calculate root1 = (-b / (2 * a)) + (i * sqrt(-D)) / (2 * a).
   c. Calculate root2 = (-b / (2 * a)) - (i * sqrt(-D)) / (2 * a).
6. Output the roots.

2. Finding Minimum and Maximum Numbers in a Set

Given a set of numbers, finding the minimum and maximum can be done with the following algorithm:

- **Algorithm**:
    1. Initialize min and max with the first element of the set.
    2. For each element in the set:
        - If the element is less than min, update min.
        - If the element is greater than max, update max.
    3. After iterating through the set, min holds the minimum value, and max holds the maximum value.

Flowchart/Pseudocode:

BEGIN

1. Initialize min and max with the first element of the set:

   SET min = firstElement

   SET max = firstElement

2. FOR each element in the set (starting from the second element):

   a. IF element < min THEN

      UPDATE min = element

   b. IF element > max THEN

      UPDATE max = element

3. After iterating through the set:

   PRINT "Minimum value:", min

   PRINT "Maximum value:", max

END

3. Checking if a Number is Prime

A **prime number** is a natural number greater than 1 that is not divisible by any number other than 1 and itself.

- **Algorithm**:

1. Start with the number n.
2. If $n \leq 1$, it is not prime.
3. For i from 2 to $\sqrt{n}$, check if n is divisible by i.
4. If n is divisible by any i, it is not prime. Otherwise, it is prime.

Flowchart/Pseudocode:

BEGIN

1. Input the number n.
2. IF n ≤ 1 THEN

   PRINT "Not prime"

   EXIT
3. FOR i = 2 to √n:

   IF n is divisible by i THEN

   PRINT "Not prime"

   EXIT
4. IF no divisors found THEN

   PRINT "Prime"

END

<u>Program Design and Structured Programming:</u>

Program design and structured programming emphasize breaking down a program into smaller, manageable modules or functions, each with a specific purpose.

1. Principles of Structured Programming

- **Top-Down Design**: Start with a broad overview of the program and break it down into smaller modules.
- **Modularization**: Divide the program into distinct modules or functions, each handling a specific task.
- **Control Structures**: Use sequence (order of execution), selection (if/else), and iteration (loops) to control the flow of the program.

Example Program: Summing an Array of Numbers

- **Design**:
  1. Initialize sum to 0.
  2. Iterate through the array and add each element to sum.

3. Return sum.

- **Structured Programming in C**:

  Example:

```
int sumArray(int arr[], int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += arr[i];
   }
   return sum;
}
```

Introduction to C Programming Language:

Variables:

- A variable is a named location in memory that stores data and can be modified during program execution.
- Variables are declared with a specific data type that determines the type of data they can hold and the space they occupy in memory.

Naming Conventions/Rules followed while declaring variables:

These rules must be followed for the compiler to recognize the variable names.

1. Valid Characters

- Variable names can consist of **letters** (both uppercase and lowercase), **digits**, and the **underscore** (_) symbol.
- **Examples**: count, total_sum, index1, value_2.

2. The First Character Must Be a Letter or an Underscore

- A variable name must start with a **letter** (A-Z or a-z) or an **underscore** (_). It **cannot** start with a digit.
- **Examples**:
  - Valid: count, _value
  - Invalid: 1count, 9_value

3. No Special Characters

- Variable names **cannot** contain special characters like @, #, $, %, ^, &, or *.
- **Invalid Examples**: value#1, total$sum

4. No Spaces

- Variable names **cannot** contain spaces. Use underscores (_) to separate words in multi-word variable names.
- **Invalid Example**: total count
- **Valid Example**: total_count

5. Case Sensitivity

- Variable names in C are **case-sensitive**. This means count and Count are considered different variables.

Variable Declaration:

A **variable declaration** informs the compiler about the name and type of a variable that will be used in the program. It reserves a memory location for the variable.

Syntax:

data_type variable_name;

- **data_type**: Specifies the type of data the variable will hold (e.g., int, float, char).
- **variable_name**: The name given to the variable, following naming rules.

example:

int total; // Declares an integer variable named 'total'

float average; // Declares a float variable named 'average'

char grade; // Declares a char variable named 'grade'

int x, y, z; // Declares three integer variables: x, y, and z

Variable Initialization:

Initialization refers to assigning an initial value to a variable when it is declared. A variable can be initialized when it is declared, or it can be assigned a value later in the code.

Syntax:

data_type variable_name = value;

Example:

int total = 100; // Declares and initializes an integer variable with a value of 100

float average = 76.5; // Declares and initializes a float variable with a value of 76.5

char grade = 'A'; // Declares and initializes a char variable with the value 'A'

int x = 10, y = 20, z = 30; // Declares and initializes three integer variables

- Variables can be initialized later in the program after declaration
  int total;
  total = 100; // Variable 'total' is assigned a value after declaration

Compile-time Initialization(static):

Variables are initialized when they are declared. The value assigned to the variable is fixed at compile time.

Example: int a = 5;

Run-time Initialization(Dynamic):

Variables are initialized during the execution of the program (i.e., at runtime), based on user input or calculation.

Example:

```
 int a;

scanf("%d", &a);  // 'a' is initialized with a value entered by the user
```

Comments:

Comments are non-executable statements that provide explanations, annotations, or notes about the code. They help programmers document their code, making it easier to understand, maintain, and debug.

Types of Comments:

C supports two types of comments:

1.Single-Line Comments

Single-line comments start with // and continue until the end of the line. They are typically used for brief explanations or annotations.

- **Syntax**:

  // This is a single-line comment

  Example:

  int total = 100; // Initializing total with 100

2. Multi-Line Comments

Multi-line comments, also known as block comments, start with /* and end with */. They can span multiple lines and are useful for more detailed explanations or temporarily commenting out blocks of code.

- **Syntax**:

  /*

```
   This is a multi-line comment.
   It can span multiple lines.
*/
```

Example:

```
/*
   This program calculates the sum of two numbers.
   It then prints the result to the console.
*/
int a = 5, b = 10;
int sum = a + b;
printf("Sum is: %d", sum);
```

Identifiers:

An **identifier** is a name used to identify variables, functions, arrays, or other user-defined items. Identifiers play a crucial role as they allow the programmer to reference specific parts of the program easily.

Constants:

Constants refer to fixed values that do not change during the execution of a program. These values are defined once and used multiple times in the program. Constants can be of different data types, including integers, floating-point numbers, characters, and strings.

Types of Constants

1. **Integer Constants**
   - Integer constants are whole numbers without a decimal point.
   - **Types of Integer Constants**:
     - **Decimal**: Base 10 (e.g., 10, 42, -7)
     - **Octal**: Base 8, prefixed with 0 (e.g., 012 represents decimal 10)
     - **Hexadecimal**: Base 16, prefixed with 0x or 0X (e.g., 0xA represents decimal 10)
   - **Rules**:
     - No commas or spaces.
     - Must fit within the range of the data type (int, long, etc.).

2. Floating-point Constants

- Represent real numbers, containing a fractional part.
- Can be expressed in **fractional form** (e.g., 3.14, -0.001) or **exponential form** (e.g., 1.5e3 for 1.5 * 10^3).
- **Precision**: By default, floating-point constants are treated as double in C, can use a suffix (f or F) to specify a float constant.

3. Character Constants

- A character constant is enclosed in single quotes and represents a single character (e.g., 'A', '9', '!').
- **Escape Sequences**: Special character constants that represent non-printable or special characters (e.g., '\n' for newline, '\t' for tab).
- Character constants are stored as integers using the **ASCII** values.

4. String Constants

- A string constant (or string literal) is a sequence of characters enclosed in double quotes (e.g., "Hello", "12345").
- **Memory Allocation**: Strings are stored as arrays of characters terminated by a null character (\0).

Defining Constants

1 #define Preprocessor Directive

- The #define directive is used to define symbolic constants in C.

  Example:

  ```
  #define PI 3.14159
  #define MAX_SIZE 100
  ```

Keywords:

Keywords are reserved words that have special meanings defined by the C language. They cannot be used for naming variables, functions, or other identifiers, as they are part of the language's syntax. These keywords are essential for writing valid C programs and control various operations within the language.

Characteristics of Keywords

- **Reserved**: Keywords are predefined and reserved for specific uses in the C language.
- **Lowercase**: All keywords in C must be written in **lowercase** (e.g., int, return). The C language is case-sensitive, so using uppercase versions (e.g., INT, RETURN) would not be valid keywords.

- **Cannot be redefined**: Since keywords are reserved, they cannot be redefined or used for any other purpose (e.g., variable names, function names).

List of Keywords

Some commonly used keywords in C are as follows:

- int : Defines integer data types.
- char : Defines character data types.
- float : Defines floating-point data types.
- double : Defines double-precision floating-point data types.
- if : Used for conditional statements.
- else : Provides alternative execution paths for if statements.
- switch : Used for multi-way branching.
- case : Used within switch statements.
- while, for, do : Used for looping.
- break : Terminates a loop or switch statement.
- continue : Skips the current iteration of a loop and moves to the next iteration.
- auto : Declares automatic storage (local variables).
- extern : Declares an external variable (global).
- static : Retains a variable's value between function calls.
- register : Requests storage in a register for faster access.
- signed : Modifies data types to hold signed values.
- unsigned : Modifies data types to hold only positive values.
- const : Declares variables whose values cannot be modified.
- volatile : Informs the compiler that a variable may change at any time.
- void : Declares functions that do not return a value.
- return : Terminates a function and optionally returns a value.
- sizeof : Returns the size of a variable or type.
- typedef : Defines new type names.
- enum : Declares enumerated types.
- struct : Defines a structure (collection of variables of different data types).
- union : Defines a union (similar to a structure, but stores different types in the same memory location).

- goto : Performs an unconditional jump to another part of the program.

Data Types:

1. **Basic Data Types**:
   - **int**: Stores integers. Typically requires 4 bytes of memory. E.g., int age = 25;
   - **float**: Stores single-precision floating-point numbers. Typically requires 4 bytes. E.g., float salary = 50000.50;
   - **double**: Stores double-precision floating-point numbers. Typically requires 8 bytes. E.g., double pi = 3.14159;
   - **char**: Stores a single character. Typically requires 1 byte. E.g., char grade = 'A';

1.1 Integer Types

- Used to represent whole numbers (both positive and negative).
- **Types**:
  - int: The most commonly used integer data type.
  - short int (or short): Shortened version of int, uses less memory.
  - long int (or long): Extended version of int, uses more memory for larger values.
  - unsigned int: Used for non-negative values only, which extends the range of positive numbers.

Type	Size (bytes)	Range
Int	2 or 4	-32,768 to 32,767 (2 bytes) or larger range with 4 bytes
short int	2	-32,768 to 32,767
long int	4	-2,147,483,648 to 2,147,483,647
unsigned int	2 or 4	0 to 65,535 (2 bytes)

1.2 Floating-Point Types

- Used to represent real numbers (numbers with a fractional part).
- **Types**:

- float: Single precision floating-point type, generally accurate up to 6 decimal places.
- double: Double precision floating-point type, generally accurate up to 15 decimal places.
- long double: Extended precision, used when even more precision is needed.

Type	Size (bytes)	Precision (decimal places)
float	4	~6
double	8	~15
long double	10 or 16	~19

1.3 Character Type

- Used to represent single characters.
- **Type**:
  - char: Used to store a single character (like 'A', '1', '!'), but technically, it stores an **ASCII** integer value.

Type	Size (bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255

2. **Derived Data Types**:
   - **array**: A collection of elements of the same type. E.g., int arr[5] = {1, 2, 3, 4, 5};
   - **pointer**: Stores the memory address of another variable. E.g., int *p = &age;
   - **structure**: A collection of different data types grouped together. E.g.
   - **union**: Similar to a structure, but stores only one member at a time.
3. **Enumerated Data Type:**
   - Used to define a collection of related named constants.
   - Ex: enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
   - Enumerated values start from 0 by default, so Sunday is 0, Monday is 1, and so on.

Size and Range of Data Types

The size of data types is machine-dependent and can vary from one system to another. Here is a general summary:

Data Type	Size (bytes)	Range
Char	1	-128 to 127
unsigned char	1	0 to 255
Int	2 or 4	-32,768 to 32,767 or larger range
unsigned int	2 or 4	0 to 65,535
short int	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long int	4	-2,147,483,648 to 2,147,483,647
Float	4	6 decimal places
double	8	15 decimal places
long double	10 or 16	19 decimal places

Syntax and Logical Errors in Compilation:

Syntax Errors (Syntactical Errors):

- Occur when the rules or syntax of the C programming language are violated.
- Common syntax errors include missing semicolons, mismatched parentheses or braces, undeclared variables, etc.
- Syntax errors are caught by the compiler and must be corrected before the program can be compiled.

Example of Syntax Error:

```
int main() {
   int num = 10
   printf("Number: %d", num);
   return 0;
}
```

Output: Error: Missing semicolon after int num = 10

Logical Errors (Semantical Errors):

- Occur when the program runs without syntax errors but produces incorrect or unexpected results.
- Logical errors are due to flaws in the algorithm or logic used in the program.
- These errors are not detected by the compiler and must be identified through testing and debugging.

Example of Logical Error:

```
int main() {
   int a = 5, b = 0;
   int result = a / b;  // Logical error: division by zero
   printf("Result: %d", result);
   return 0;
}
```

Output: Error: Division by zero, which leads to undefined behavior

Object and Executable Code:

Object Code

- Object code is the intermediate code generated by the compiler after compiling the source code.
- It is machine-specific and not directly executable.
- Object code is usually stored in files with extensions like .obj or .o.

Executable Code

- Executable code is the final product of the compilation and linking process.
- It is a binary file that can be executed directly by the machine's operating system.
- In Windows, executable files typically have the .exe extension.

Example: When you compile a C program using a compiler, the following steps occur:

1. **Compilation**: The source code (.c file) is converted into object code (.o file).
2. **Linking**: The object code is linked with other object files and libraries to produce the executable code (.exe file on Windows).

Operators:

Operators are symbols that perform operations on variables and values.

Types of Operators:

- **Arithmetic Operators**: +, -, *, /, % (modulus)
- **Relational Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: && (AND), || (OR), ! (NOT)
- **Bitwise Operators**: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift)
- **Assignment Operators**: =, +=, -=, *=, /=, %=
- **Increment/Decrement Operators**: ++, --

1. Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values.

- **Addition (+)**: Adds two operands.
  - **Example**: int sum = 5 + 3; // sum will be 8
- **Subtraction (-)**: Subtracts the second operand from the first.
  - **Example**: int difference = 10 - 4; // difference will be 6
- **Multiplication (*)**: Multiplies two operands.
  - **Example**: int product = 6 * 7; // product will be 42
- **Division (/)**: Divides the first operand by the second.
  - **Example**: int quotient = 20 / 5; // quotient will be 4
- **Modulus (%)**: Returns the remainder of a division.
  - **Example**: int remainder = 10 % 3; // remainder will be 1

Example:

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    int sum = a + b;        // Addition
    int diff = a - b;       // Subtraction
    int product = a * b;    // Multiplication
    int quotient = a / b;   // Division
    int remainder = a % b;  // Modulus
    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", diff);
```

```
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
    return 0;
}
```

Output:

Sum: 13

Difference: 7

Product: 30

Quotient: 3

Remainder: 1

2. Relational Operators

Relational operators compare two values or expressions and return a boolean value (true or false).

- **Equal to (==)**: Checks if two operands are equal.
    - **Example**: if (a == b) {...} // True if a is equal to b
- **Not equal to (!=)**: Checks if two operands are not equal.
    - **Example**: if (a != b) {...} // True if a is not equal to b
- **Greater than (>)**: Checks if the first operand is greater than the second.
    - **Example**: if (a > b) {...} // True if a is greater than b
- **Less than (<)**: Checks if the first operand is less than the second.
    - **Example**: if (a < b) {...} // True if a is less than b
- **Greater than or equal to (>=)**: Checks if the first operand is greater than or equal to the second.
    - **Example**: if (a >= b) {...} // True if a is greater than or equal to b
- **Less than or equal to (<=)**: Checks if the first operand is less than or equal to the second.
    - **Example**: if (a <= b) {...} // True if a is less than or equal to b

Example:

```
#include <stdio.h>
int main() {
```

```
    int a = 5, b = 10;
    printf("a == b: %d\n", a == b);  // Equal to
    printf("a != b: %d\n", a != b);  // Not equal to
    printf("a > b: %d\n", a > b);    // Greater than
    printf("a < b: %d\n", a < b);    // Less than
    printf("a >= b: %d\n", a >= b);  // Greater than or equal to
    printf("a <= b: %d\n", a <= b);  // Less than or equal to
    return 0;
}
```

Output:

```
a == b: 0
a != b: 1
a > b: 0
a < b: 1
a >= b: 0
a <= b: 1
```

3. Logical Operators

Logical operators are used to combine or negate boolean expressions.

- **Logical AND (&&)**: Returns true if both operands are true.
    - **Example**: if (a > 0 && b < 10) {...} // True if a is positive and b is less than 10
- **Logical OR (||)**: Returns true if at least one of the operands is true.
    - **Example**: if (a > 0 || b < 10) {...} // True if a is positive or b is less than 10
- **Logical NOT (!)**: Reverses the boolean value of the operand.
    - **Example**: if (!isValid) {...} // True if isValid is false

Example:

```
#include <stdio.h>
int main() {
    int a = 1, b = 0;
    printf("a && b: %d\n", a && b);  // Logical AND
```

```
    printf("a || b: %d\n", a || b);  // Logical OR
    printf("!a: %d\n", !a);          // Logical NOT
    printf("!b: %d\n", !b);          // Logical NOT
    return 0;
}
```

Output:

a && b: 0

a || b: 1

!a: 0

!b: 1

4. Bitwise Operators

Bitwise operators perform operations on binary representations of integers.

- **Bitwise AND (&)**: Performs a bitwise AND operation.
    - **Example**: int result = a & b; // result is the bitwise AND of a and b
- **Bitwise OR (|)**: Performs a bitwise OR operation.
    - **Example**: int result = a | b; // result is the bitwise OR of a and b
- **Bitwise XOR (^)**: Performs a bitwise exclusive OR operation.
    - **Example**: int result = a ^ b; // result is the bitwise XOR of a and b
- **Bitwise NOT (~)**: Inverts all the bits of the operand.
    - **Example**: int result = ~a; // result is the bitwise NOT of a
- **Left Shift (<<)**: Shifts the bits of the first operand left by the number of positions specified by the second operand.
    - **Example**: int result = a << 2; // result is a shifted left by 2 bits
- **Right Shift (>>)**: Shifts the bits of the first operand right by the number of positions specified by the second operand.
    - **Example**: int result = a >> 2; // result is a shifted right by 2 bits
- **Rotate Right Shift**: Rotate Right Shift is a bit manipulation operation where the bits of a binary number are shifted to the right, and the bits that fall off are reintroduced at the leftmost position.
  Example: rotating 10110011 right by 1 bit results in 11011001
- **Rotate Left Shift:** Rotate Left Shift is a bit manipulation operation where the bits of a binary number are shifted to the left, and the bits that fall off the end are reintroduced at the rightmost position.

Example: rotating 10110011 left by 1 bit results in 01100111

Example:

```
#include <stdio.h>
int main() {
    int a = 5;  // Binary: 0101
    int b = 3;  // Binary: 0011
    int andResult = a & b;   // Bitwise AND
    int orResult = a | b;    // Bitwise OR
    int xorResult = a ^ b;   // Bitwise XOR
    int notResult = ~a;      // Bitwise NOT
    int leftShift = a << 2;  // Left shift
    int rightShift = a >> 1; // Right shift
    printf("AND: %d\n", andResult);
    printf("OR: %d\n", orResult);
    printf("XOR: %d\n", xorResult);
    printf("NOT: %d\n", notResult);
    printf("Left Shift: %d\n", leftShift);
    printf("Right Shift: %d\n", rightShift);
    return 0;
}
```

Output:

AND: 1

OR: 7

XOR: 6

NOT: -6

Left Shift: 20

Right Shift: 2

5. Assignment Operators

Assignment operators assign values to variables. They can also combine other operations with assignment.

- **Assignment (=)**: Assigns the value of the right operand to the left operand.
    - **Example**: a = 10; // Assigns 10 to a
- **Add and Assign (+=)**: Adds the right operand to the left operand and assigns the result to the left operand.
    - **Example**: a += 5; // Equivalent to a = a + 5
- **Subtract and Assign (-=)**: Subtracts the right operand from the left operand and assigns the result to the left operand.
    - **Example**: a -= 3; // Equivalent to a = a - 3
- **Multiply and Assign (*=)**: Multiplies the left operand by the right operand and assigns the result to the left operand.
    - **Example**: a *= 2; // Equivalent to a = a * 2
- **Divide and Assign (/=)**: Divides the left operand by the right operand and assigns the result to the left operand.
    - **Example**: a /= 4; // Equivalent to a = a / 4
- **Modulus and Assign (%=)**: Takes the modulus of the left operand with the right operand and assigns the result to the left operand.
    - **Example**: a %= 2; // Equivalent to a = a % 2

Example:

```
#include <stdio.h>

int main() {
    int a = 10;
    a += 5;  // Equivalent to a = a + 5
    printf("a += 5: %d\n", a);
    a -= 3;  // Equivalent to a = a - 3
    printf("a -= 3: %d\n", a);
    a *= 2;  // Equivalent to a = a * 2
    printf("a *= 2: %d\n", a);
    a /= 4;  // Equivalent to a = a / 4
    printf("a /= 4: %d\n", a);
    a %= 3;  // Equivalent to a = a % 3
```

```
    printf("a %%= 3: %d\n", a);
    return 0;
}
```

Output:

a += 5: 15

a -= 3: 12

a *= 2: 24

a /= 4: 6

a %= 3: 0

6. Increment/Decrement Operators

These operators increase or decrease the value of a variable by one.

- **Increment (++)**: Increases the value of the operand by one.
    - **Example**: a++; // Increments a by 1
- **Decrement (--)**: Decreases the value of the operand by one.
    - **Example**: a--; // Decrements a by 1

These operators can be used in either **prefix** or **postfix** form:

- **Prefix (++a, --a)**: The value is incremented or decremented before it is used in an expression.
- **Postfix (a++, a--)**: The value is incremented or decremented after it is used in an expression.

Example:

```
#include <stdio.h>
int main() {
    int a = 5;
    printf("Initial value of a: %d\n", a);
    printf("Post-increment (a++): %d\n", a++);  // Use then increment
    printf("Value after post-increment: %d\n", a);
    printf("Pre-increment (++a): %d\n", ++a);   // Increment then use
    printf("Post-decrement (a--): %d\n", a--);  // Use then decrement
    printf("Value after post-decrement: %d\n", a);
```

```
    printf("Pre-decrement (--a): %d\n", --a);   // Decrement then use
    return 0;
}
```

Output:

Initial value of a: 5

Post-increment (a++): 5

Value after post-increment: 6

Pre-increment (++a): 7

Post-decrement (a--): 7

Value after post-decrement: 6

Pre-decrement (--a): 5

Expressions:

- An expression is a combination of variables, operators, and values that produces a result.
- **Example**: 3 + 5 * 2 is an expression where multiplication is performed first due to operator precedence, resulting in 3 + 10, which equals 13.

Precedence and Associativity

- **Precedence**: Determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated first.
- **Associativity**: Determines the direction of evaluation for operators with the same precedence. Most operators have left-to-right associativity.

Precedence Level	**Operator(s)**	**Description**	**Associativity**
1	() [] -> . ++ --	Function call, array subscript, structure member access, and post-increment/decrement	Left to Right
2	+ - ! ~ ++ -- * & sizeof (type)	Unary plus, minus, logical NOT, bitwise NOT, pre-increment/decrement, dereference, address-of, sizeof, type cast	Right to Left
3	* / %	Multiplication, division, modulus	Left to Right

Precedence Level	Operator(s)	Description	Associativity
4	+ -	Addition, subtraction	Left to Right
5	<< >>	Bitwise left shift, bitwise right shift	Left to Right
6	< <= > >=	Relational operators	Left to Right
7	== !=	Equality operators	Left to Right
8	&	Bitwise AND	Left to Right
9	^	Bitwise XOR	Left to Right
10	`	`	Bitwise OR
11	&&	Logical AND	Left to Right
12	`		`
13	?:	Ternary conditional operator	Right to Left
14	= += -= *= /= %= <<= >>= &= ^= `	=`	Assignment operators
15	,	Comma (used to separate expressions)	Left to Right

Explanation of Associativity

- **Left to Right Associativity:** Operators with left-to-right associativity are evaluated from left to right in an expression.
    - Example: In the expression a - b + c, both - and + have the same precedence and are evaluated from left to right, so a - b is evaluated first, then the result is added to c.
- **Right to Left Associativity:** Operators with right-to-left associativity are evaluated from right to left in an expression.
    - Example: In the expression a = b = c, the assignment operator = has right-to-left associativity, so b = c is evaluated first, and then the result is assigned to a.
- **Example 1:** int x = 10 + 5 * 2;
    - Here, multiplication (*) has a higher precedence than addition (+), so 5 * 2 is evaluated first, and then the result is added to 10.
    - Result: x will be 20.
- **Example 2:** int y = (10 + 5) * 2;

- Parentheses () have the highest precedence, so 10 + 5 is evaluated first, then the result is multiplied by 2.
- Result: y will be 30.

Expression Evaluation:

- The process of calculating the value of an expression.
- Follows the rules of precedence and associativity to determine the order of operations.

Example:

```
int a = 10, b = 5, c = 2;
int result = a - b * c + 4 / 2;  // Evaluates to 10 - 10 + 2 = 2
printf("Result: %d", result);
```

Storage Classes:

- **Storage classes** define the scope, lifetime, and visibility of variables within a program.
- **Scope** refers to the region of the code where a variable is accessible. In C, the scope of a variable is determined by where the variable is declared.
  - **Global Scope:**

    Variables declared outside of all functions (usually at the top of the file) are known as global variables. They have global scope.

    Characteristics:

    - **Visibility:** Global variables are accessible from any function within the same file and, if declared with the extern keyword, can be accessed from other files.
    - **Lifetime:** They exist for the entire duration of the program's execution, from start to finish.
    - **Initialization:** If not explicitly initialized, global variables are automatically initialized to zero (or null for pointers).
  - **Local Scope:**

    Variables declared inside a function or a block (enclosed by `{}`) are known as local variables. They have local scope.

    Characteristics:

- **Visibility:** Local variables are only accessible within the function or block in which they are declared.
- **Lifetime:** They exist only for the duration of the function or block execution. Once the function or block execution is complete, the local variable is destroyed.
- **Initialization:** Local variables are not automatically initialized; they contain garbage values if not explicitly initialized.

- **Lifetime** refers to the duration during which a variable exists in memory. In C, the lifetime of a variable depends on its storage class.
- **Visibility** (or linkage) refers to whether a variable or function can be accessed from different parts of a program. In C, visibility is influenced by the variable's storage class and its declaration.

Types of Storage Classes:

1. **auto**:
   - The default storage class for local variables.
   - The variable is created when the block in which it is defined is entered and destroyed when the block is exited.
   - **Example:**

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

2. **extern**:
   - Used to declare a global variable or function that is defined in another file.
   - It extends the visibility of the C variables and functions across multiple files.
   - **Example:**

```
// In file1.c
int num = 10;  // Global variable definition

// In file2.c
extern int num;  // Declaration of 'num' defined in file1.c
```

3. **static**:

- Retains the value of a variable across multiple function calls or throughout the program's lifetime.
- A static variable within a function is created only once and retains its value between function calls.
- **Example:**

```
void count() {
    static int counter = 0;  // 'counter' is initialized only once
    counter++;
    printf("Counter: %d\n", counter);
}
```

4. **register**:
   - Suggests to the compiler that the variable should be stored in a CPU register for faster access.
   - The compiler may ignore this suggestion if no registers are available.
   - **Example:**

```
void func() {
    register int num = 5;  // Suggests 'num' to be stored in a register
    printf("%d\n", num);
}
```

Type Conversion:

- Type conversion refers to converting a variable from one data type to another.
- **Implicit Conversion**: The compiler automatically converts a variable from one data type to another.
- **Explicit Conversion (Type Casting)**: The programmer manually converts a variable from one type to another using casting.

Example of Implicit Conversion:

```
int a = 10;
float b = a;  // 'a' is implicitly converted to float
```

Example of Explicit Conversion:

```
float num = 9.67;
int integerPart = (int)num;  // Explicit conversion from float to int
printf("Integer part: %d", integerPart);  // Output: 9
```

The main Method and Command Line Arguments:

The main Method

- The main function is the entry point of a C program where execution begins.
- It can take command line arguments: int main(int argc, char *argv[])

Command Line Arguments

- argc: Argument count, which indicates the number of command line arguments.
- argv[]: Argument vector, an array of strings representing the command line arguments.

Example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {

printf("Number of arguments: %d\n", argc);

for(int i = 0; i < argc; i++){

printf("Argument %d: %s\n", i, argv[i]);

 }

        return 0;

}
```

If run as ./program arg1 arg2, the output will show the arguments passed

Conditional Branching:

Conditional branching allows the program to make decisions based on certain conditions. The main constructs for conditional branching in C are if, if-else, switch-case, the ternary operator, and goto.

1. if Statement (two-way selection):

The if statement executes a block of code if a specified condition is true.

Syntax:

```
if (condition) {
   // Code to execute if condition is true
}
```

Null Else Statement:

A null else statement refers to a scenario where an else clause exists without any corresponding code block to execute. This situation usually arises when the programmer intends to handle only the if case and does not require any action if the condition fails.

Example:

```
int x = 5;
if (x > 3) {
   printf("x is greater than 3");
} else {
   ;  // Null else statement
}
```

In the above example, the else part has a semicolon ;, indicating that nothing will happen if x is not greater than 3. This kind of null else statement can be used for clarity or to avoid the dangling else problem, but it's generally unnecessary.

Example:

```
int a = 5;
if (a > 3) {
   printf("a is greater than 3");
}
```

2. if-else Statement

The if-else statement provides an alternative block of code to execute if the condition is false.

Syntax:

```
if (condition) {
   // Code to execute if condition is true
} else {
   // Code to execute if condition is false
}
```

Example:

```
int a = 5;
if (a > 3) {
   printf("a is greater than 3");
```

```
} else {
   printf("a is not greater than, 3");
}
```

Nested if Statement:

A **nested if statement** refers to placing one if statement inside another. This allows checking multiple conditions in a hierarchical manner, where the execution of inner if statements depend on the success of the outer ones.

Syntax:

```
if (condition1) {
   if (condition2) {
      // Code to execute if both condition1 and condition2 are true
   } else {
      // Code to execute if condition1 is true but condition2 is false
   }
} else {
   // Code to execute if condition1 is false
}
```

Example:

```
int x = 10;
int y = 20;
if (x > 5) {
   if (y > 15) {
      printf("x is greater than 5 and y is greater than 15");
   } else {
      printf("x is greater than 5 but y is not greater than 15");
   }
} else {
   printf("x is not greater than 5");
}
```

In this example, the inner if checks if y > 15, but only if the outer if confirms that x > 5. This structure allows for complex decision-making processes.

Dangling Else Problem

The **dangling else problem** arises in nested if-else statements when it's unclear which if statement an else clause is associated with. C follows a simple rule: the else matches the nearest preceding unmatched if.

Problematic Example:

```
int x = 5;
int y = 10;
if (x > 3)
   if (y > 5)
      printf("y is greater than 5");
else
   printf("x is not greater than 3");
```

In the above example, the else is meant to be associated with the first if (x > 3) statement, but due to C's matching rule, it actually associates with the second if (y > 5) statement. This causes unexpected behaviour.

Solution:

To resolve this, the if-else statements should be properly nested using braces {} to clearly define the association:

Corrected Example:

```
int x = 5;
int y = 10;
if (x > 3)
{
   if (y > 5)
 {
      printf("y is greater than 5");
   }
}
else
{
   printf("x is not greater than 3");
}
```

Now, the else clearly corresponds to the outer if (x > 3) statement, avoiding the dangling else problem.

3.Else if (if else if ladder):

else if is used to check multiple conditions in sequence. It follows an if statement, and if the first condition is false, it checks the condition in the else if block. If all conditions are false, the else block will be executed.

Syntax:

```
if (condition1)
 {
 // Code to be executed if condition1 is true
}
else if (condition2)
 {
   // Code to be executed if condition2 is true and condition1 is false
}
 else if (condition3)
 {
 // Code to be executed if condition3 is true and both condition1 and condition2 are false
}
else
 {
   // Code to be executed if all conditions are false
}
```

4. switch-case Statement (multi-way selection)

- The switch-case statement allows multi-way branching based on the value of an expression.
- The selection alternatives are called as **case** labels. For every possible value in the switch expression, a separate case label is defined.
- **Default** label is a special form of the case label. It is executed when none of the other case values/labels matches the value in the switch expression.
- The **break** statement causes the program to jump out of the switch statement.

Syntax:

```
switch (expression) {
    case constant1:
        // Code to execute if expression == constant1
        break;
    case constant2:
        // Code to execute if expression == constant2
        break;
    default:
        // Code to execute if none of the cases match
}
```

Example:

```
int day = 3;
switch (day) {
    case 1:
        printf("Monday");
        break;
    case 2:
        printf("Tuesday");
        break;
    case 3:
        printf("Wednesday");
        break;
    default:
        printf("Invalid day");
        break;
}
```

2b) Write a C program, which takes two integer operands and one operator from the user, performs the operation and then prints the result. (Consider the operators +,-,*, /, % and use Switch Statement)

Algorithm:

1.Start

2. Input two integers (operands) and an operator from the user.

3. Use a switch statement to perform the operation based on the operator:

- Case +: Perform addition of the two integers.
- Case -: Perform subtraction of the two integers.
- Case *: Perform multiplication of the two integers.
- Case /: Perform division of the two integers (check for division by zero).
- Case %: Perform modulus operation (check for division by zero).
- Default Case: Handle invalid operator input.

4. Output the result of the operation.

5.End

//C program

```
#include <stdio.h>
int main() {
    int num1, num2;
    char operator;
    float result;
    printf("Enter first integer: ");
    scanf("%d", &num1);
    printf("Enter second integer: ");
    scanf("%d", &num2);
    printf("Enter an operator (+, -, *, /, %%): ");
    scanf(" %c", &operator);  // Note the space before %c to consume any whitespace
    switch (operator)
    {
        case '+':
            result = num1 + num2;
            printf("Result: %d + %d = %.2f\n", num1, num2, result);
            break;
```

```
        case '-':
            result = num1 - num2;
            printf("Result: %d - %d = %.2f\n", num1, num2, result);
            break;
        case '*':
            result = num1 * num2;
            printf("Result: %d * %d = %.2f\n", num1, num2, result);
            break;
        case '/':
            if (num2 != 0) {
                result = (float)num1 / num2;
                printf("Result: %d / %d = %.2f\n", num1, num2, result);
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break;
        case '%':
            if (num2 != 0) {
                result = num1 % num2;
                printf("Result: %d %% %d = %.2f\n", num1, num2, result);
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break;
        default:
            printf("Error: Invalid operator.\n");
            break;
    }
    return 0;
}
```

Output:

Enter first integer: 7

Enter second integer: 4

Enter an operator (+, -, *, /, %): *

Result: 7 * 4 = 28.00

5. Ternary Operator (?:)

The ternary operator is a compact form of an if-else statement.

Syntax:

(condition) ? expression_if_true : expression_if_false;

Example:

```
int a = 5;
int b = (a > 3) ? 10 : 20;
printf("%d", b);  // Output: 10
```

6. goto Statement

The goto statement allows jumping to a labeled part of the program. It's generally discouraged because it can make code harder to follow.

Syntax:

```
goto label;
...
label:
   // Code to execute after jumping to the label
```

Example:

```
int a = 10;
if (a > 0) {
   goto positive;
}
negative:
   printf("a is negative");
   return;
```

positive:

```
printf("a is positive");
```

1a) Write a program for finding the max and min from the three numbers.

1. Start
2. Input three numbers a, b, and c.
3. Compare the three numbers to find the maximum:
    - If a > b and a > c, then a is the maximum.
    - Else if b > a and b > c, then b is the maximum.
    - Else, c is the maximum.
4. Compare the three numbers to find the minimum:
    - If a < b and a < c, then a is the minimum.
    - Else if b < a and b < c, then b is the minimum.
    - Else, c is the minimum.
5. Output the maximum and minimum values.
6. End

//C Program for Finding Maximum and Minimum:

```
#include <stdio.h>
int main() {
   int a, b, c, max, min;
   printf("Enter three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   // Finding the maximum
   if (a > b && a > c) {
      max = a;
   } else if (b > a && b > c) {
      max = b;
   } else {
      max = c;
   }
```

```
    // Finding the minimum
    if (a < b && a < c) {
        min = a;
    } else if (b < a && b < c) {
        min = b;
    } else {
        min = c;
    }
    printf("Maximum: %d\n", max);
    printf("Minimum: %d\n", min);
    return 0;
}
```

Output:

Enter three numbers: 10 25 15

Maximum: 25

Minimum: 10

1c) Write a program that declares Class awarded for a given percentage of marks, where mark <40%= Failed, 40% to <60% = Second class, 60% to <70%=First class, >= 70% = Distinction. Read percentage from standard input.

Algorithm:

1.Start

2.Input the percentage of marks.

3.Check the percentage:

- If the percentage is less than 40%, the result is "Failed".
- If the percentage is between 40% and 59%, the result is "Second Class".
- If the percentage is between 60% and 69%, the result is "First Class".
- If the percentage is 70% or more, the result is "Distinction".

4.Output the corresponding class based on the percentage.

5.End

//C program to find the class awarded for a student

```
#include <stdio.h>
int main() {
   float percentage;
   printf("Enter the percentage of marks: ");
   scanf("%f", &percentage);
   if (percentage < 40)
{
      printf("Result: Failed\n");
   } else if (percentage >= 40 && percentage < 60) {
      printf("Result: Second Class\n");
   } else if (percentage >= 60 && percentage < 70) {
      printf("Result: First Class\n");
   } else if (percentage >= 70) {
      printf("Result: Distinction\n");
   }
   return 0;
}
```

Output:

Enter the percentage of marks: 85

Result: Distinction

Standard function:

Function	Header File	Description	Example
printf	<stdio.h>	Prints formatted output to the console.	printf("Hello, %s!", "World"); // Output: Hello, World!
scanf	<stdio.h>	Reads formatted input from the user.	int x; scanf("%d", &x);
strcpy	<string.h>	Copies a string from source to destination.	char dest[20]; strcpy(dest, "Hello");
strlen	<string.h>	Returns the length of a string.	int len = strlen("Hello"); // len = 5

Function	Header File	Description	Example
strcmp	<string.h>	Compares two strings lexicographically.	strcmp("abc", "abd"); // Returns negative value
strcat	<string.h>	Concatenates two strings.	char s1[20] = "Hello"; strcat(s1, " World");
strchr	<string.h>	Finds the first occurrence of a character in a string.	char *ptr = strchr("Hello", 'e'); // ptr points to "ello"
memcpy	<string.h>	Copies a block of memory.	memcpy(dest, src, sizeof(src));
sqrt	<math.h>	Returns the square root of a number.	double res = sqrt(16.0); // res = 4.0
pow	<math.h>	Raises a number to a power.	double res = pow(2.0, 3.0); // res = 8.0
abs	<stdlib.h>	Returns the absolute value of an integer.	int result = abs(-10); // result = 10
malloc	<stdlib.h>	Allocates memory dynamically.	int *ptr = (int*)malloc(10 * sizeof(int));
calloc	<stdlib.h>	Allocates and initializes memory dynamically.	int *ptr = (int*)calloc(10, sizeof(int));
free	<stdlib.h>	Frees dynamically allocated memory.	free(ptr);
atoi	<stdlib.h>	Converts a string to an integer.	int num = atoi("123"); // num = 123
rand	<stdlib.h>	Generates a random number.	int r = rand();
isalpha	<ctype.h>	Checks if a character is alphabetic.	if (isalpha('A')) { /* It's an alphabetic character */ }
isdigit	<ctype.h>	Checks if a character is a digit.	if (isdigit('9')) { /* It's a digit */ }
toupper	<ctype.h>	Converts a character to uppercase.	char ch = toupper('a'); // ch = 'A'
tolower	<ctype.h>	Converts a character to lowercase.	char ch = tolower('A'); // ch = 'a'

Assignment:

1. Write a program that shows the binary equivalent of a given positive number between 0 to 255.

Loops:

Loops allow repeating a block of code multiple times, as long as a specified condition is true. The main loop constructs in C are for, while, and do-while.

1. for Loop

The for loop is typically used when the number of iterations is known beforehand. The for loop is another pre-test loop but is often used when the number of iterations is known in advance. It provides a compact syntax by combining initialization, condition-checking, and increment/decrement in a single line.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // Code to execute in each iteration
}
```

Example:

```
for (int i = 0; i < 5; i++) {
    printf("%d ", i);
}
```

2. while Loop

The while loop is used when the number of iterations is not known beforehand. The while loop is a pre-test loop, meaning that the condition is evaluated before the execution of the loop's body. If the condition is true, the loop's body is executed. This process repeats until the condition becomes false.

Syntax:

```
while (condition) {
    // Code to execute as long as condition is true
}
```

Example:

```
int i = 0;
while (i < 5) {
    printf("%d ", i);
    i++;
}
```

3. do-while Loop

The do-while loop is a post-test loop, meaning that the loop's body is executed at least once before the condition is checked. This ensures that the code inside the loop is executed at least once, regardless of whether the condition is true or false.

Syntax:

```
do {
   // Code to execute
} while (condition);
```

Example:

```
int i = 0;
do {
   printf("%d ", i);
   i++;
} while (i < 5);
```

1d) Write a program that prints a multiplication table for a given number and the number of rows in the table. For example, for a number 5 and rows = 3, the output should be:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

Algorithm:

1.Start

2. Input the number for which the multiplication table is required.

3. Input the number of rows.

4.Initialize a loop from 1 to the given number of rows:

- Multiply the number with the current loop index.
- Print the result in the format number x index = result.

5. End

//C Program for Printing a Multiplication Table

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

```
int main() {
   int number, rows;
   printf("Enter the number: ");
   scanf("%d", &number);
   printf("Enter the number of rows: ");
   scanf("%d", &rows);
   for (int i = 1; i <= rows; i++)
{
      printf("%d x %d = %d\n", number, i, number * i);
   }
   return 0;
}
```

Output:

Enter the number: 5

Enter the number of rows: 3

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

2a) A building has 10 floors with a floor height of 3 meters each. A ball is dropped from the top of the building. Find the time taken by the ball to reach each floor. (Use the formula s = ut+(1/2)at^2 where u and a are the initial velocity in m/sec (= 0) and acceleration in m/sec^2 (= 9.8 m/s^2)).

Algorithm:

1.Start.

2. Declare the necessary variables: height, time, and constants g, floorHeight, and totalFloors.

3. For each floor from 1 to totalFloors:

- Calculate the height from which the ball falls using:

  height=floorHeight×(totalFloors−floor)

- Calculate the time to reach that floor using:

  t=sqrt((2 * height)/g)

- Print the floor number and the time.

4. End the loop after calculating for all floors.

5.Stop.

//C program

```
#include <stdio.h>
#include <math.h>
int main()
 {
   float height, time;
    const float g = 9.8;  // acceleration due to gravity
    const int floorHeight = 3;
    int totalFloors = 10;
    for (int floor = 1; floor <= totalFloors; floor++)
 {
 height = floorHeight * (totalFloors - floor);  // height from which ball falls for each floor
  time = sqrt((2 * height) / g);  // calculating time to reach each floor
   printf("Time to reach floor %d: %.2f seconds\n", floor, time);
    }
   return 0;
}
```

2c) Write a program that finds if a given number is a prime number

1. Input the Number
    - Read the integer num.
2. Check
    - If num is less than or equal to 1, it is not a prime number.
3. Check for Divisibility
    - If num is equal to 2, it is a prime number (since 2 is the smallest prime number).
    - For numbers greater than 2:
        - Loop from 2 to the square root of num.

- For each iteration, check if num is divisible by the current loop variable i.
- If num is divisible by any i, it is not a prime number.
- If the loop completes without finding any divisors, the number is a prime number.

4. Output the Result
    - Print whether the number is prime or not.

//C program to check whether the given number is prime or not

```
#include <stdio.h>
#include <math.h>
int main()
{
   int num, i;
   int isPrime = 1;
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num <= 1)
{
      isPrime = 0;
   }
else if (num == 2)
{
      isPrime = 1; // 2 is the only even prime number
   }
Else
 {
      for (i = 2; i <= sqrt(num); i++) {
         if (num % i == 0) {
            isPrime = 0; // Number is divisible by i, so it's not prime
            break;
```

```
            }
        }
    }
    if (isPrime) {
        printf("%d is a prime number.\n", num);
    } else {
        printf("%d is not a prime number.\n", num);
    }
    return 0;
}
```

Output:

Enter a number: 11

11 is a prime number.

2c) Write a C program to find the sum of individual digits of a positive integer and test given number is palindrome.

Algorithm:

1.Start

2. Input a positive integer from the user.

3. Initialize a variable sum to 0.

4. Loop until the number becomes 0:

- Extract the last digit of the number using digit = num % 10.
- Add the digit to sum.
- Remove the last digit from the number using num = num / 10.

5. Output the value of sum (the sum of the digits).

6.End

//C Program to Find the Sum of Individual Digits

```
#include <stdio.h>
int main() {
    int num, digit, sum = 0;
```

```
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Please enter a positive integer.\n");
        return 0;
    }
    while (num > 0) {
        digit = num % 10;   // Extract the last digit
        sum += digit;       // Add the digit to sum
        num /= 10;          // Remove the last digit
    }
    printf("The sum of the digits is: %d\n", sum);
    return 0;
}
```

Output:

Enter a positive integer: 123

The sum of the digits is: 6

Algorithm for Palindrome:

1.Start

2. Read integer num.

3. Store originalNum = num

4. Initialize reverse = 0

5. While num > 0

- Extract: digit = num % 10
- Update: reverse = reverse * 10 + digit
- Update: num /= 10

6. Compare originalNum with reverse

- If equal, Output: "The number is a palindrome."
- Otherwise, Output: "The number is not a palindrome."

7.End

//C program to find the positive integer is palindrome or not.

```
#include <stdio.h>
int main() {
    int num, originalNum, reverse = 0, digit;
    printf("Enter a number: ");
    scanf("%d", &num);
    originalNum = num;
    // Reverse the digits of the number
    while (num > 0) {
     digit = num % 10;         // Extract the last digit
     reverse = reverse * 10 + digit;  // Build the reversed number
     num /= 10;                // Remove the last digit
    }
    // Check if the original number is equal to the reversed number
    if (originalNum == reverse) {
        printf("%d is a palindrome.\n", originalNum);
    } else {
        printf("%d is not a palindrome.\n", originalNum);
    }
    return 0;
}
```

2d) A Fibonacci sequence is defined as follows: the first and second terms in the sequence are 0 and 1. Subsequent terms are found by adding the preceding two terms in the sequence. Write a C program to generate the first n terms of the sequence.

Algorithm:

1.Start

2. Input the number of terms n from the user.

3. Initialize the first two terms of the Fibonacci sequence as t1 = 0 and t2 = 1.

4. Print the first two terms.

5. Loop from 3 to n (since the first two terms are already printed):

- Calculate the next term as nextTerm = t1 + t2.
- Print the next term.
- Update t1 and t2 as t1 = t2 and t2 = nextTerm.

6.End

//C program for Fibonacci sequence

```
#include <stdio.h>
int main() {
   int i,n, t1 = 0, t2 = 1, nextTerm;
   printf("Enter the number of terms: ");
   scanf("%d", &n);
   if (n <= 0)
{
      printf("Please enter a positive integer.\n");
      return 0;
   }
   printf("Fibonacci Sequence: %d, %d", t1, t2);
   for (i = 3; i <= n; ++i) {
      nextTerm = t1 + t2;
      printf(", %d", nextTerm);
      t1 = t2;
      t2 = nextTerm;
   }
   printf("\n");
   return 0;
}
```

Output:

Enter the number of terms: 5

Fibonacci Sequence: 0, 1, 1, 2, 3

2f) Write a C program to generate all the prime numbers between 1 and n, where n is a value supplied by the user.

Algorithm:

1.Start

2. Input the value of n from the user.

3. Loop through numbers from 2 to n:

- For each number, check if it is prime:
    - A number is prime if it is not divisible by any number from 2 to the square root of the number.

4. Print the number if it is prime.

5. End

//C program

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>  // for sqrt function
int main() {
   int n;
   printf("Enter the value of n: ");
   scanf("%d", &n);
   printf("Prime numbers between 1 and %d are: ", n);
   // Loop through numbers from 2 to n
   for (int num = 2; num <= n; num++) {
      bool isPrime = true;  // Assume num is prime
      for (int i = 2; i <= sqrt(num); i++) {
         if (num % i == 0) {
            isPrime = false;
            break;
         }
      }
      if (isPrime) {
         printf("%d ", num);
      }
   }
```

```
    printf("\n");
    return 0;
}
```

Output:

Enter the value of n: 20

Prime numbers between 1 and 20 are: 2 3 5 7 11 13 17 19

2g) Write a C program to find the roots of a Quadratic equation.

Algorithm:

1. Start
2. Input the coefficients of the quadratic equation a, b, and c from the user.
3. Calculate the discriminant D using the formula D = b^2 - 4ac.
4. Check the value of the discriminant:
   - If D > 0: The equation has two distinct real roots.
   - If D == 0: The equation has exactly one real root (repeated).
   - If D < 0: The equation has no real roots (the roots are complex).
5. Calculate the roots using the formulas:
   - For D >= 0, the roots are:
     - root1 = (-b + sqrt(D)) / (2 * a)
     - root2 = (-b - sqrt(D)) / (2 * a)
   - For D < 0, display a message indicating that the roots are complex.
6. Print the roots or the message about complex roots.
7. End

//C Program to Find the Roots of a Quadratic Equation

```
#include <stdio.h>
#include <math.h>
int main() {
    float a, b, c, discriminant, root1, root2;
    printf("Enter coefficients a, b and c: ");
    scanf("%f %f %f", &a, &b, &c);
    discriminant = b * b - 4 * a * c;
```

```
    if (discriminant > 0) {
        // Two distinct real roots
        root1 = (-b + sqrt(discriminant)) / (2 * a);
        root2 = (-b - sqrt(discriminant)) / (2 * a);
        printf("Roots are real and different.\n");
        printf("Root 1 = %.2f\n", root1);
        printf("Root 2 = %.2f\n", root2);
    } else if (discriminant == 0) {
        // One real root (repeated)
        root1 = root2 = -b / (2 * a);
        printf("Roots are real and the same.\n");
        printf("Root 1 = Root 2 = %.2f\n", root1);
    } else {
        // Complex roots
        float realPart = -b / (2 * a);
        float imaginaryPart = sqrt(-discriminant) / (2 * a);
        printf("Roots are complex and different.\n");
        printf("Root 1 = %.2f + %.2fi\n", realPart, imaginaryPart);
        printf("Root 2 = %.2f - %.2fi\n", realPart, imaginaryPart);
    }
    return 0;
}
```

Output:

Enter coefficients a, b and c: 1 -3 2

Roots are real and different.

Root 1 = 2.00

Root 2 = 1.00

Q) C program to construct a pattern

Algorithm:

1.Input the Number:

- Read an integer n from the user which represents the number of rows in the pattern.

2.Generate the Pattern:

- For each row from 1 to n:
    - Print i stars, where i is the current row number.

3.Output the Pattern:

- Print each row of stars as per the current row number.

// C Program

```
#include <stdio.h>
int main()
 {
   int n, i, j;
   printf("Enter the number of rows: ");
   scanf("%d", &n);
   for (i = 1; i <= n; i++)
 {
      for (j = 1; j <= i; j++)
{
         printf("*");
      }
      printf("\n");
   }
   return 0;
}
```

Output:

```
*
**
***
****
```

Comma Expression:

Allows multiple expressions to be evaluated in a single statement, with each expression separated by a comma. The key feature of a comma expression is that it evaluates all expressions from left to right, but the value of the entire comma expression is the value of the last expression in the sequence.

Syntax:

expression1, expression2, expression3, ..., expressionN;

Here, all the expressions expression1 to expressionN are evaluated in sequence. The value of the entire comma expression is the value of expressionN.

Example

```
int x = (1, 2, 3);
```

The expressions 1, 2, and 3 are evaluated in order, but the value of x is 3 because it is the result of the last expression.

While comma expressions are powerful, they can also be confusing if not used carefully:

- **Readability**: Code readability can suffer when multiple comma expressions are used, making it harder for others to understand the flow of the program.
- **Side Effects**: Since comma expressions allow multiple assignments or operations, they can introduce side effects if not carefully managed, leading to unexpected results.

Controlling Loop Execution:

Control statements can alter the flow of loops.

1. break Statement: Immediately exits the loop, skipping any remaining iterations.

Example:

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break; // Exit loop when i is 5
   }
   printf("%d\n", i);
}
```

The loop terminates when i equals 5, so only numbers 1 to 4 are printed.

2. continue Statement: Skips the remaining code in the current iteration and proceeds with the next iteration.

- **Example**:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Skip printing when i is 3
    }
    printf("%d\n", i);
}
```

The loop skips the number 3, so only numbers 1, 2, 4, and 5 are printed.

Nested Loops:

Loops can be nested, meaning that one loop can exist inside another. This is often used for tasks that require multiple levels of repetition, such as iterating over a two-dimensional array.

Syntax:

```
for (initialization; condition; increment/decrement) {
    for (initialization; condition; increment/decrement) {
        // nested loop statements
    }
}
```

Example:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        printf("i = %d, j = %d\n", i, j);
    }
}
```

Infinite Loops:

An infinite loop runs indefinitely because its condition is never false. This can happen intentionally (for tasks that must run continuously) or unintentionally (due to a logical error).

- **Example**:

```
while (1) {
    printf("This loop runs forever.\n");
}
```

The condition 1 is always true, so the loop never exits. This is often used in systems that require continuous operation.

Input and Output (I/O):

Input and Output operations are essential for interacting with the user. The standard functions for basic I/O in C are scanf and printf.

1. scanf and printf

- **scanf** is used for reading formatted input from the user.
- **printf** is used for displaying formatted output.

Example:

```
int age;

printf("Enter your age: ");

scanf("%d", &age);

printf("Your age is %d", age);
```

2. Formatted I/O

Formatted I/O allows specifying how the data should be input or output, such as specifying the number of decimal places for a floating-point number.

Formatted Output (printf)

- **Purpose**: To print data to the standard output (usually the console) in a specified format.
- **Syntax**: printf("format string", arguments);
- **Format Specifiers**: You use format specifiers in the format string to indicate how you want the data to be formatted. For example:
    - %d for integers
    - %f for floating-point numbers
    - %s for strings
    - %x for hexadecimal numbers

Example:

```
float pi = 3.14159;

printf("Pi to two decimal places: %.2f", pi);  // Output: 3.14
```

Formatted Input (scanf)

- **Purpose**: To read data from the standard input (usually the keyboard) and interpret it according to a specified format.
- **Syntax**: scanf("format string", &arguments);
- **Format Specifiers**: Similar to printf, format specifiers indicate what type of data is expected. For example:
    - %d for integers
    - %f for floating-point numbers
    - %s for strings

Example:

```
int age;
float height;
scanf("%d %f", &age, &height);
```

- **stdin**: Standard input stream (usually the keyboard).
- **stdout**: Standard output stream (usually the console).
- **stderr**: Standard error stream (used for error messages).

Example:

```
fprintf(stdout, "This is standard output\n");
fprintf(stderr, "This is an error message\n");
```

<u>Command-Line Arguments:</u>

Command-line arguments allow passing arguments to the program when it is executed.

Syntax:

```
int main(int argc, char *argv[]) {
   // argc: Number of arguments
   // argv: Array of argument strings
}
```

Example:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   printf("Program name: %s\n", argv[0]);
   if (argc > 1) {
```

```
        printf("First argument: %s\n", argv[1]);
    }
    return 0;
}
```

Output:

Program name: ./program_name

First argument: arg1

Assignment:

1.Write a C program to calculate the following, where x is a fractional value.

1-x/2 +x^2/4-x^3/6

2. Write a C program to read in two numbers, x and n, and then compute the sum of this geometric progression: 1+x+x^2+x^3+…….+x^n. For example: if n is 3 and x is 5, then the program computes 1+5+25+125.