UNIT -1

INTRODUCTION

- Dennis Ritchie is known as the father of the C programming language.
- C is called the "mother of all programming languages"
    - (C is called the "mother of programming languages" because it is simple, powerful, and widely used. It teaches the basics of how computers work and is the foundation for learning other languages.)Definition: C is a general-purpose, mid-level, procedural, and structured programming language.

It lets you talk to the computer's hardware directly while also helping you write clear and structured programs.

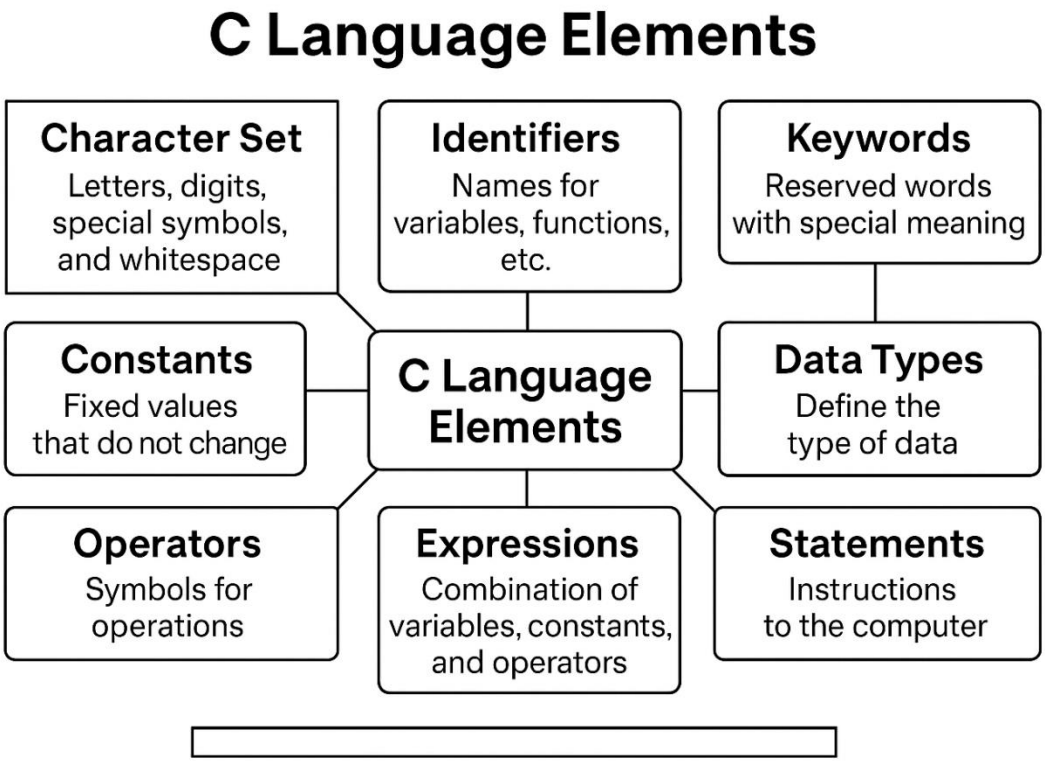
C LANGUAGE ELEMENTS:

When we write a program in C, we use certain building blocks. These blocks are called C language elements. They are the basic components that make up a C program.

The main elements are:

1. **Character Set** – The smallest unit of a program, consisting of letters (A–Z, a–z), digits (0–9), special symbols (+, -, *, /, etc.), and whitespace (space, tab, newline).
2. **Identifiers** – Names given to variables, functions, arrays, etc. Must begin with a letter or underscore and cannot use special symbols or keywords. Example: age, total_marks.
3. **Keywords** – Predefined reserved words with special meaning in C, such as int, if, return, while.
4. **Constants** – Fixed values that never change during program execution. Examples: 10, 3.14, 'A', "Hello".
5. **Variables** – Named storage locations whose values can change during program execution. Example: int num = 10;.

6. **Data Types** – Define the type of data a variable can store, such as int (integers), float (decimals), char (characters), double (large decimals).
7. **Operators** – Symbols used to perform operations. Types include arithmetic (+, -), relational (>, <=), logical (&&, ||), and assignment (=).
8. **Expressions** – Combinations of variables, constants, and operators that produce a result. Example: sum = a + b;.
9. **Statements** – Instructions given to the computer, such as expression statements, control statements (if, for, while), and compound statements (block of code inside { }).
10. **Functions** – A group of statements designed to perform a specific task. Every program must have main(), the starting function.



HISTORY OF C PROGRAMMING

The C programming language was developed by Dennis Ritchie at Bell Labs in 1972

Year	Event
1960s	BCPL (Basic Combined Programming Language) developed by Martin Richards, influencing later languages.
1970	B language created by Ken Thompson at Bell Labs, simplifying BCPL for system programming.
1972	Dennis Ritchie developed C at Bell Labs, adding data types and structures to B.
1973	Unix was rewritten in C, marking the rise of C in system programming.
1978	Kernighan & Ritchie published The C Programming Language, standardizing early C (K&R C).
1989	ANSI C (C89) was standardized, ensuring portability and consistency across platforms.
1999	C99 introduced features like inline functions and variable-length arrays.
2011	C11 brought enhancements such as multithreading support and improved Unicode handling.
2018	C17, a minor revision to C11, included bug fixes and slight improvements.
2023-	C23 (expected 2024) set to replace C17, introducing

24	further updates to modernize the language

VARIABLE

A variable in C programming is a named memory location used to store data that can be changed during program execution. It acts like a container holding values such as numbers or characters, which the program can manipulate.

The variable name allows the program to access and modify the stored value without needing to know its actual memory address.

Important Points

- Variables must be declared with a datatype before use.
  - Variable names must start with a letter (a-z, A-Z) or an underscore (_), not a digit.
  - Variable names can only contain letters, digits (0-9), and underscores (_).
  - Variable names cannot contain spaces or special characters (like @, #, $, etc.).
  - Variable names are case sensitive (e.g., count and Count are different).
  - Variable names cannot be C reserved keywords (like int, while, return, etc.).

- Variables allow programs to store and manipulate data dynamically.

- **Variable Declaration** : Variable declaration is the act of informing the compiler about the variable's name and datatype. It tells the compiler what kind of data the variable will hold. No memory is allocated during declaration alone.

- **Variable Definition** : Variable definition allocates storage (memory) for the variable and optionally initializes it with a value. It creates the variable and reserves memory for it in the program. Memory is allocated when a variable is defined.

IDENTIFIERS

Identifiers are the names used to identify variables, functions, arrays, structures, or any other user-defined items. It is a name that uniquely identifies a program element and can be used to refer to it later in the program.

Rules for naming identifiers :

- Identifier can contain following characters:
    - Uppercase (A-Z) and lowercase (a-z) alphabets.
    - Numeric digits (0-9).
    - Underscore (_).
- The first character of an identifier must be a letter or an underscore.

- Identifiers are case-sensitive.
- Identifiers cannot be keywords in C (such as int, return, if, while etc.).

Example 1:

int age; // age is a variable and also an identifier

void greet() { } // greet is a function name, an identifier but not a variable

- "age" is an identifier (it is the name given).
- The variable "age" refers to a memory location that stores an integer value.
- Variables are a type of identifier that point to memory locations holding values.
- All variables are identifiers, but not all identifiers are variables (for example, function names are identifiers too).
- "greet" is an identifier but not a variable.

Example 2:

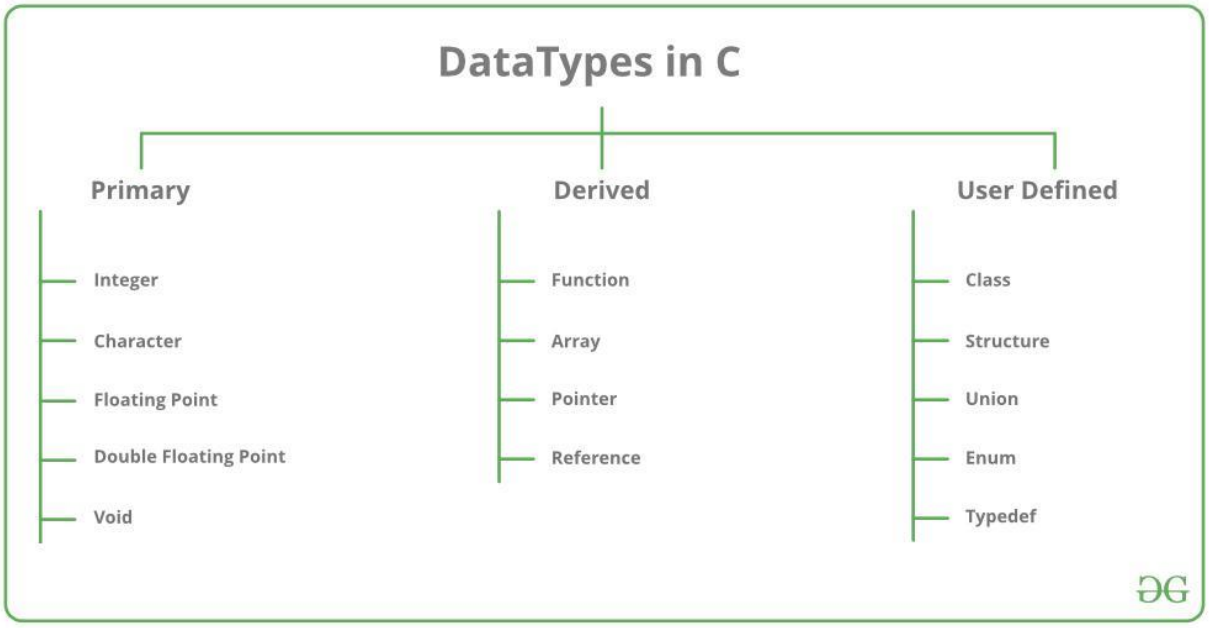
```
#include <stdio.h>

// Example: MAX_VALUE (typically a constant, not a variable)

#define MAX_VALUE 100        // 1: Not a variable, valid identifier (constant macro)

// Example: age

int age = 25;              // 2: Valid variable, valid identifier

// Example: _count

int _count = 10;           // 3: Valid variable, valid identifier
```

```
// Example: 2ndPlace (invalid)
 int 2ndPlace = 2;       // 4: Invalid variable, invalid identifier (starts with digit)
// Example: total-value (invalid)
int total-value = 100;   // 5: Invalid variable, invalid identifier (contains hyphen '-')
// Example: my age (invalid)
int my age = 50;         // 6: Invalid variable, invalid identifier (contains space)
// Example: int (invalid)
int int = 5;             // 7: Invalid variable, invalid identifier (reserved keyword)
// Example: myVar123
int myVar123 = 123;         // 8: Valid variable, valid identifier
// Example: var_name
int var_name = 7;           // 9: Valid variable, valid identifier
// Example: $amount (invalid)
int $amount = 50;         // 10: Invalid variable, invalid identifier (special char $)
// Example: #temp (invalid)
int #temp = 4;           // 11: Invalid variable, invalid identifier (special char #)
// Example: sum used as function name, not a variable
int sum(int a, int b) {     // 12: Not a variable, valid identifier (function name)
    return a + b;
}
```

```
// Example: main is the special function, not a variable

int main() {              // 13: Not a variable, valid identifier (special function)

    printf("Age: %d\n", age);

    printf("_count: %d\n", _count);

    printf("myVar123: %d\n", myVar123);

    printf("var_name: %d\n", var_name);

    printf("Sum of 5 and 10: %d\n", sum(5, 10));

    return 0;

}
```

DATA TYPES

A data type in C is a specification that determines the type and amount of data a variable can store, as well as how the system interprets and operates on that data.



ASCII:

ASCII (American Standard Code for Information Interchange) is a character encoding standard.

It assigns a **unique integer value (0–127)** to characters (letters, digits, symbols, control keys). In C, the type char stores a character as its ASCII integer value internally.

'0' **(digit zero) → 48**

'a' **(lowercase a) → 97**

'A' **(uppercase A) → 65**

Example:

```
#include <stdio.h>

int main() {

   int num = 65;   // ASCII value of 'A'

   printf("Integer value: %d\n", num);   // prints as integer

   printf("Character value: %c\n", num); // prints as character

   return 0;
```

}

Output:

Integer value: 65

Character value: A

Range :

- The **minimum** and **maximum** values that can be stored in a variable of that type.
- Ranges may vary based on platform (compiler, architecture) – the above is commonly on Turbo C (16 or 32 bit).
- It depends on the size (in bytes) of the type and whether it is signed or unsigned.

Unsigned Range : 0 to 2^n-1

Signed Range : -2^{n-1} to 2^{n-1}-1

Example :

char = 1 byte

1 byte = 8 bits

So, n = 8 bits

(a) Unsigned char

Formula:

Range=0 to 2^n-1

Substitute n=8

=0 to (2^8-1)= 0 to 255= 0

Final Range: 0 → 255

(b) Signed char

Formula:

Range= -2^{n-1} to $2^{n-1}-1$)

Substitute n=8

=-2^7 to (2^7-1)=-128 to 127

Final Range: -128 → 127

S.No	Data Type	Type of Data Type	Range (Typical 32-bit system)
1	int	Basic	–32,768 to 32,767 (short int) –2,147,483,648 to 2,147,483,647 (int)
2	float	Basic	~ 1.2E-38 to 3.4E+38 (6 digits precision)
3	double	Basic	~ 2.3E-308 to 1.7E+308 (15 digits precision)
4	char	Basic	–128 to 127 (signed char) 0 to 255 (unsigned char)
5	void	Basic	No value (used for empty return type / pointers)
6	short int	Basic	–32,768 to 32,767
7	long int	Basic	–2,147,483,648 to 2,147,483,647
8	long long int	Basic	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
9	unsigned int	Basic	0 to 4,294,967,295
10	unsigned long	Basic	0 to 4,294,967,295
11	unsigned long long	Basic	0 to 18,446,744,073,709,551,615
12	wchar_t	Basic	0 to 65,535 (or more, platform-dependent)
13	array	Derived	Collection of elements of same type
14	pointer	Derived	Stores memory address
15	structure	User-defined	Collection of different datatypes
16	union	User-defined	Stores one of many datatypes (shared memory)
17	enum	User-defined	Set of named integer constants
18	typedef	User-defined	Defines new name for an existing type

DATA TYPES

Type conversion refers to the process of converting one data type to another. It can be done automatically by the compiler or manually by the programmer. The type conversion is only performed to those data types where conversion is possible.

Implicit Type Conversion (Type Coercion): The compiler automatically converts one data type to another without programmer intervention, usually when mixing different types in operations (e.g., int to float).

Explicit Type Conversion (Type Casting): The programmer manually converts a value from one data type to another using casting syntax (e.g., (int)3.14) to control the conversion process.

Example:

```
#include <stdio.h>

int main() {
   float f1 = 5.5;
        int a;
   // Automatic type conversion from float to int
   a = f1;
   // Manual type conversion from int to float
   float f2 = (float)a;
   printf("a: %d\n", a);
        printf("f1: %f\n", f1);
        printf("f2: %f", f2);
```

```
    return 0;
}
```

Output:

a: 5

f1: 5.500000

f2: 5.000000

EXECUTABLE STATEMENTS

Executable statements are program instructions that direct the computer to perform specific operations, such as computation, data manipulation, or input/output, during program execution.

Types of Executable Statements

Input/Output Statements (scanf, printf)

Assignment Statements (x = 5;)

Arithmetic Statements (x = a + b;)

General Form of a C Program

A C program follows a well-defined structure that makes it easier to write, read, and maintain. Every program, from the simplest "Hello World" to complex applications, is built on this general form.

A typical C program is divided into six essential sections:

1. Documentation Section
2. Preprocessor Section (Link Section)

3. Definition Section

4. Global Declaration Section

5. Main() Function

6. Subprograms (User-defined Functions)

1. Documentation Section

- This part uses comments to describe the program, its purpose, author, and creation date.

- Comments do not impact program execution but improve readability and documentation.

Comments:

A comment in C is a line or block of text in a program that is ignored by the compiler. They are written only for human understanding (programmers, students, or readers).

Single-line Comment:

A single-line comment in C starts with //. Everything after // on the same line is ignored by the compiler. It is used for short notes or explanations.

Multi-line Comment:

A multi-line comment in C starts with /* and ends with */. Everything between these symbols is ignored by the compiler. It is used for long explanations or for commenting out multiple lines of code.

Example:

```
#include <stdio.h>

int main() {

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

```
    // Declaring variables
    int a = 10, b = 20;
    /*
       This is a multi-line comment.
       The following code calculates
       the sum of two numbers.
    */
    int sum = a + b;
    // Printing the result
    printf("Sum = %d", sum);
    return 0; // Program ends here
}
```

Output:

Sum = 30

2. Preprocessor Section

- Contains preprocessor directives like #include <stdio.h> which import header files.
- Essential for using built-in functions (like printf, scanf, etc.).

3. Definition Section

- Defines constants using #define or const.

- These values remain unchanged throughout the program and improve code reliability.

Example:

```
#include <stdio.h>

#define PI 3.14159  // defining a constant using #define

int main() {

   float radius = 5.0;

   float area = PI * radius * radius; // using the constant PI

   printf("Area of Circle = %.2f", area);

   return 0;

}
```

Output:

Area of Circle = 78.54

4. Global Declaration Section

- Variables and function prototypes declared here are accessible throughout the program.

5. Main() Function

- Mandatory entry point for every C program.
- Only one main() function exists in a program; it's where code execution begins.

6. Subprograms (User-defined Functions)

- Functions make code modular, reusable, and organized.
- They are written outside the main() function but called from within it.

Arithmetic Expression

Arithmetic Expression is a combination of operands and Arithmetic operators. These combinations of operands and operators should be mathematically meaningful, otherwise, they can not be considered as an Arithmetic expression in C.

Based on the number of operands they require, operators are classified into Unary, Binary, and Ternary operators.

Unary Operators: A unary operator works with only one operand. It performs operations such as increment, decrement, negation, and logical NOT.

Binary Operators: A binary operator works with two operands**.**

Ternary Operator:

- The ternary operator is the only operator in C that takes three operands.
- It is also called the conditional operator (?:).
- It is used as a shorthand for if-else statements.

Example 1:

```
#include <stdio.h>

int main() {

   int a = 5, b = 10, result;

    // Unary operator example
```

```
    result = -a;   // changes sign
    printf("Unary (-a): %d\n", result);
    // Binary operator example
    result = a + b; // addition
    printf("Binary (a+b): %d\n", result);
    // Ternary operator example
    result = (a > b) ? a : b;  // condition ? true_value : false_value
    printf("Ternary (a > b ? a : b): %d\n", result);
    return 0;
}
```

Output:

```
Unary (-a): -5
Binary (a+b): 15
Ternary (a > b ? a : b): 10
```

Example 2:

```
int a = 5;
printf("%d\n", ++a); // Pre → a becomes 6, then prints 6
printf("%d\n", a++); // Post → prints 6 first, then a becomes 7
```

Output:

```
6
6
```

Symbol	Unary/ Binary	Description
+	Unary	denotes that the number is a positive integer.
-	Unary	denotes that the number is a negative integer.
++	Unary	Increments the value of the variable by 1
--	Unary	Decreases the value of the variable by 1
+	Binary	performs the mathematical addition of the two given operands.
-	Binary	performs the mathematical subtraction of the two given operands.
*	Binary	performs the mathematical multiplication of the two given operands.
\	Binary	performs the mathematical division of the two given operands and returns the quotient.
%	Binary	performs the mathematical division of the two given operands and returns the remainder as the result.

The evaluation of an arithmetic expression is based on three different things; precedence, the associativity of the arithmetic operators, and the data types of the operands over which the arithmetic operation is being performed.

OPERATORS:

An operator in C is a symbol or special character that tells the compiler to perform specific mathematical, logical, or relational operations on data (operands). Operators manipulate data values and produce results.

Types of Operators in C

1. **Arithmetic Operators**

   Perform basic mathematical operations.

Examples: + (addition), -
(subtraction), * (multiplication), / (division), % (modulus).

Example:

Write a program using % to check whether a number is **even or odd**.

Program:

```
#include <stdio.h>
int main() {
    int num = 13;   // number defined directly
    if (num % 2 == 0)
        printf("%d is Even\n", num);
    else
        printf("%d is Odd\n", num);
    return 0;
}
```

Output:

13 is Odd

2. **Relational Operators**

Compare two values and return true or false.

Examples: == (equal to), != (not equal to), > (greater than), < (less than), >=, <=.

Example:

Write a C program that compares two integers and prints whether the first is less than or equal to the second.

Program:

```
#include <stdio.h>

int main() {
    int a = 10;   // first integer
    int b = 20;   // second integer
    if (a <= b)
        printf("%d is less than or equal to %d\n", a, b);
    else
        printf("%d is greater than %d\n", a, b);
    return 0;
}
```

Output:

10 is less than or equal to 20

3. **Logical Operators**

   Perform logical operations on conditions.

   Examples: && (logical AND), || (logical OR), ! (logical NOT).

   Example:

   Write a C program to check if a number is either divisible by 3 or divisible by 5. If yes, print "Divisible by 3 or 5". Otherwise, print "Not divisible by 3 or 5".

   Program:

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num); // user input is required
    if (num % 3 == 0 || num % 5 == 0)
        printf("Divisible by 3 or 5\n");
    else
        printf("Not divisible by 3 or 5\n");
    return 0;
}
```

Output:

Enter a number: 15

Divisible by 3 or 5

4. **Assignment Operators**

Assign values to variables.

Examples: =, +=, -=, *=, /=, %=.

Example Program:

```
#include <stdio.h>

int main() {

   int a = 5;   // assignment (=)

   a += 3;      // same as a = a + 3

   printf("Value of a = %d\n", a);

   return 0;

}
```

Output:

Value of a = 8

5. **Bitwise Operators**

Perform operations on individual bits of integers.

Examples: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift).

Example : Check Even or Odd using Bitwise Operator

Program:

```
#include <stdio.h>
int main() {
   int num = 13;   // Hardcoded number
   if (num & 1)    // checks last bit
      printf("%d is Odd\n", num);
   else
      printf("%d is Even\n", num);
   return 0;
}
```

Output:

13 is Odd

6. **Conditional (Ternary) Operator**

   A shorthand for if-else statement.

   Syntax: condition ? expr1 : expr2.

   Example:

int num = 7

(num % 2 == 0) ? printf("%d is Even\n", num) : printf("%d is Odd\n", num);

Output:

7 is Odd

7. **Miscellaneous Operators**

- sizeof: Returns size of data type or variable.
- & (address of): Returns the address of a variable.
- *** (dereference)**: Accesses the value at an address (pointer).

, (comma): Separates expressions.

Example:

```
#include <stdio.h>
int main() {
   int a = 10, b = 20;
   printf("Size of int = %lu bytes\n", sizeof(a));
   printf("Address of a = %p\n", (void*)&a);
   int *ptr = &a;
   printf("Value of a using pointer = %d\n", *ptr);
   int result = (a = a + 5, b = b + 5, a + b);
   printf("Result using comma operator = %d\n", result);
```

```
    return 0;
}
```

Output:

```
Size of int = 4 bytes
Address of a = 0x7ffee65f4abc
Value of a using pointer = 10
Result using comma operator = 40
```

Precedence Level	**Operators**	**Associativity**	**Description**
1 (Highest)	(), [], ., ->	Left to Right	Function call, array subscript, member access
2	++, --, + (unary), - (unary), !, ~, &, *, (type), sizeof	Right to Left	Unary operators: increment, dereference, sizeof, type casting
3	*, /, %	Left to Right	Multiplication, division, modulus
4	+, -	Left to Right	Addition, subtraction
5	<<, >>	Left to Right	Bitwise shift left/right

Precedence Level	Operators	Associativity	Description
6	<, <=, >, >=	Left to Right	Relational operators
7	==, !=	Left to Right	Equality and inequality
8	&	Left to Right	Bitwise AND
9	^	Left to Right	Bitwise XOR
10	`	`	Left to Right
11	&&	Left to Right	Logical AND
12	?:	Right to Left	Ternary conditional
13	=, +=, -=, *=, /=, %= and other assignment operators	Right to Left	Assignment operators
14 (Lowest)	,	Left to Right	Comma operator

Formatting Numbers in Program Output

This is called **output formatting**.

The formatting is done using **format specifiers**.

You can **specify the number of digits** after the decimal point for floating numbers:

%.nf

where n = number of digits.

Most Common format specifiers:

%d → signed decimal integer (int)

%i → signed decimal integer (same as %d)

%f → floating-point (float, double → prints with 6 decimals by default)

%c → single character

%s → string (array of characters)

%u → unsigned decimal integer

%ld → long integer

%lld → long long integer

S.No	Data Type	Format Specifier(s)
1	char	%c
2	signed char	%c
3	unsigned char	%c (print), %hhu (scan)

S.No	Data Type	Format Specifier(s)
4	int	%d or %i
5	unsigned int	%u
6	short int	%hd
7	unsigned short int	%hu
8	long int	%ld
9	unsigned long int	%lu
10	long long int	%lld or %lli
11	unsigned long long int	%llu
12	float	%f
13	double	%lf
14	long double	%Lf

S.No	Data Type	Format Specifier(s)
15	void* (pointer)	%p
16	string (char array)	%s
17	octal representation	%o
18	hexadecimal	%x or %X
19	percentage sign (%)	%%
20	n (number of characters printed so far)	%n

Example:

```
#include <stdio.h>
int main() {
    int age = 20;              // integer
    unsigned int year = 2025;  // unsigned integer
    float price = 99.50;       // float
    float num = 12.34567; //float
```

```
    char grade = 'A';          // character
    char name[] = "John";      // string
    printf("Integer (%%d): %d\n", age);
    printf("Unsigned (%%u): %u\n", year);
    printf("Float (%%f): %f\n", price);
    printf("Two Decimal Places: %.2f\n", num);  // 2 digits after decimal
    printf("Character (%%c): %c\n", grade);
    printf("String (%%s): %s\n", name);
    return 0;
}
```

Output:

Integer (%d): 20

Unsigned (%u): 2025

Float (%f): 99.500000

Two Decimal Places: 12.35

Character (%c): A

String (%s): John

Input and Output In C Programming:

The printf and scanf statements are fundamental functions in C programming used for input and output operations.

The **printf** function is used to display output on the screen. It allows you to print text, variables, and formatted data so that users can see the program's results.

Example:

```
int age = 20;

printf("My age is %d\n", age);
```

Output: My age is 20

The **scanf** function is used to read input from the user through the keyboard. It allows the user to input values that the program can process.

Using scanf, programs can interact dynamically by taking input data, making programs more useful.

Example:

```
int age;

printf("Enter your age: ");

scanf("%d", &age);

printf("You entered: %d\n", age);
```

- ⇨ The program waits for the user to enter a number and then prints it.

SELECTION STRUCTURES (DECISION MAKING):

Selection structures allow a program to make decisions and execute different actions depending on whether a condition is true or false. They are also called **decision-making statements**.

- We have a number of situations where we may have to change the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.
- The if statement is a two way decision statement and is used in conjunction with an
- expression. It takes the following form
  if(test expression)
- If the test expression is true then the statement block after if is executed otherwise it is not executed

if statement: The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It can be achieved through different forms of if statement.

a) simple if

b) if – else

c) nested if-else

d) else-if ladder

a) simple if statement: C uses the keyword "if" to execute a set of statements or one

statements when the logical condition is true.

Syntax:

```
if(condition or expression)
{   Statements-block;
}
```

program for if :

```
#include<stdio.h>
main()
{
    int a,b;
    printf("Enter two numbers");
    scanf("%d%d",&a,&b):
    if a>b
       printf(" a is greater");
    if b>a
       printf("b is greater");
}
```

b) if –else statement: It is an extension of simple if. It takes care of both the true as well as false conditions. It has two blocks. One is for if and it is executed when the condition is true, the other block is for else and it is executed when the condition is false. No multiple else statements are allowed with one if.

Syntax:

```
if(test condition)
{
    True block statements
}
else
```

```
{
    False block statements
}
```

program for if-else :

```
#include<stdio.h>
main()
{
        int a,b;
        printf("Enter two numbers");
        scanf("%d%d",&a,&b):
        if a>b
                printf(" a is greater")
        else
                printf("b is greater");
}
```

c) Nested if-else statement: When a series of decisions are involved, we have to use more than one if-else in nested form.

Syntax:

```
if( condition -1)
{
        if(condition -2)
```

```
        {
                ……
                if(condition-3)
                {
                        Statement -1
                }
                else
                {
                        Statement -2
                }
        }
        else
        {
                Statement-3
        }
}
else
{
        Statement-4
}
```

Program for nested if:

```
if(a>b)
```

```
{
        if(a>c)
        {
                printf(“ ais big”);
        else
        {
                printf(“c is big”);
        }
 }
 else
{
        if(c>b)
        {
                printf(“c is big”);
        }
        else
        {
                printf(“ b is big’);
        }
}
```

d) else-if ladder: The else-if ladder is used when multipath decisions are involved.

Syntax:

```
if(condition-1)
        Statement-1
else if(condition-2)
        Statement-2
 else if(condition-3)
        Statement-3
        ...
        ….
        ….
 else
        Statement-x
```

Program on else-if ladder:

```
if(avg>=70)
        printf(" distinction");
else if(avg>=60)
        printf(" first class");
else if(avg>=50)
        printf("second class");
 else if(avg>=40)
        printf("third class");
 else
```

```
printf("fail");
```

THE SWITCH STATEMENT:

If for suppose we have more than one valid choices to choose from then we can use

switch statement in place of if statements.

Syntax:

```
switch(expression) {
   case value1:
      // statements
      break;

   case value2:
      // statements
      break;

   ...

   default:
      // statements (if no case matches)
}
```

Example:

```
#include <stdio.h>
int main() {
    int a, b;
    char op;
    printf("Enter first number: ");
    scanf("%d", &a);
    printf("Enter second number: ");
    scanf("%d", &b);
    printf("Enter operator (+, -, *, /): ");
    scanf(" %c", &op);   // notice the space before %c to avoid newline issue
    switch(op) {
        case '+':
            printf("Result = %d\n", a + b);
            break;
        case '-':
            printf("Result = %d\n", a - b);
            break;
        case '*':
            printf("Result = %d\n", a * b);
            break;
        case '/':
            if (b != 0)
```

```
            printf("Result = %d\n", a / b);
        else
            printf("Error: Division by zero!\n");
        break;
    default:
        printf("Invalid operator!\n");
  }
  return 0;
}
```

Output:

```
Enter first number: 10
Enter second number: 5
Enter operator (+, -, *, /): *
Result = 50
```

Repetition in Programs

Instead of writing the same code many times, we repeat it using loops.

Loop control statements

Many tasks are needed to be done with the help of a computer and they are repetitive in nature. Such type of actions can be easily done by using loop control statements.

Example: calculation of salary of employs of an organization for every month

A loop is defined as a block of statements which are repeatedly executed for certain number of times to do a specific task.

Steps in loops:-

1. Loop variable: It is a variable used in loop to evaluate.

2. Initialization: It is the step in which starting value or final values are assigned to the loop

variable.

3. Test-condition: It is to check the condition to terminate the loop. It is any relational

expression with the help of logical operators.

4. Update statement: It is the numerical value added or subtracted to the loop variable in

each round of the loop.

C language supports three types of loop control statements.

a) while

b) do-while

c) for

Conditional Loop: A while loop is known as conditional loop as it repeats as long as a condition is true. Most commonly implemented using a for loop**.**

while: This is the simplest looping structure in C. the while is an entry-controlled loop

statement.

Syntax: initial statement

```
while(test condition)
{
        Statement(s) Update
}
```

Example:

```
main()
{
        int i,sum;
         i = 1; sum=0;
        while (i<=10)
        {
                sum = sum + i ;
                i + +;
        }
        printf("sum=%d",sum);
 }
```

Here the value, sum of first ten numbers is stored into the variable sum, i is called as loop variable.

Example: Computing a Sum or Product in a Loop

```
#include <stdio.h>
int main() {
    int i = 1, sum = 0;
    while (i <= 5) {
        sum += i;   // add i to sum
        i++;
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

Output:

Sum = 15

Example: Product of first 5 numbers

```
#include <stdio.h>
int main() {
    int i = 1, product = 1;
    while (i <= 5) {
        product *= i;   // multiply i
        i++;
    }
    printf("Product = %d\n", product);
```

```
    return 0;
}
```

Output:

Product = 120

do-while: On some occasions it might be necessary to execute the body of the loop before the test condition is performed. Such situations can be handled by the do-while statement. Do-while is exit controlled loop statement.

Syntax:

```
initial statement
do
{
        Statement(s)
        Update statement
} while(test – condition);
```

Example:

```
 main()
{
      int i, sum;
```

```
        i =1;
    sum=0;
    do
    {
        sum = sum + i;
        i + +;
    } while( i<=10);
        printf("Sum=%d", sum);
    }
```

Output:

Sum=55

Counting Loops: A counting loop runs a fixed number of times, controlled by a counter variable**.**

for Loop :

It is also an entry control loop that provides a more concise structure

Syntax:

```
for(initialization; test control; increment)
{
    //body of loop
}
```

program of for loop :

```
#include <stdio.h>

int main() {

   int i, sum = 0;

   // for loop to calculate sum of first 10 natural numbers

   for (i = 1; i <= 10; i++) {

      sum = sum + i;   // add each number to sum

   }

   printf("Sum of first 10 natural numbers = %d\n", sum);

   return 0;

}
```

Output:

Sum of first 10 natural numbers = 55

Loop Design:

- Loops are one of the most powerful control structures in C, allowing programmers to repeat a sequence of instructions without unnecessary code duplication.
- poorly designed loops can lead to infinite execution, wrong results, or inefficient programs.
- Thus, understanding loop design principles is essential.

Loop design refers to the **process of planning and structuring a loop** so that it:

1. Executes the required number of times.
2. Produces correct results.

3. Terminates properly.
4. Is easy to read and maintain.

Basic Elements of Loop Design

1. Initialization

- Set the starting value of the loop control variable.
- Ensures the loop begins in a valid state.
- Example: int i = 1;

2. Condition (Test Expression)

- Decides whether the loop should continue or stop.
- Evaluated **before each iteration** (for for and while).
- Example: i <= 10

3. Update (Progression Step)

- Changes the control variable so that the loop moves toward termination.
- Prevents infinite looping.
- Example: i++

Common Mistakes in Loop Design

- **Uninitialized variables**
- **Wrong condition** (too strong/too weak)
- **Missing update** → infinite loop
- **Off-by-one error**
- **Changing control variable inside loop body**
- **Unintended infinite loops due to faulty logic**

Nested Loops:

- A **nested loop** is a loop inside another loop.
- The **outer loop** controls how many times the **inner loop** executes.

- Commonly used in **tables, patterns, and multi-dimensional data** (like matrices).

General Form:

```
for (initialization; condition; update) {      // Outer loop
   for (initialization; condition; update) {  // Inner loop
      // Statements
   }
}
```

Example:

```
#include <stdio.h>
int main() {
   for (int i = 1; i <= 5; i++) {        // Outer loop → rows
      for (int j = 1; j <= i; j++) {     // Inner loop → stars per row
         printf("* ");
      }
      printf("\n");
   }
   return 0;
}
```

Output:

```
*
* *
```

```
* * *

* * * *

* * * * *
```