What is a Programming Language?

- **Definition:**
  A programming language is a formal language comprising a set of instructions that produce various kinds of output. It is used to implement algorithms, and it allows developers to communicate with a computer to perform specific tasks.
- **Purpose:**
  Programming languages are used to create software applications, websites, games, and control devices, among many other things. They bridge the gap between human logic and computer functionality.
- **Types of Programming Languages:**
  - **High-Level Languages:** Easier for humans to understand (e.g., C, Python, Java).
  - **Low-Level Languages:** Closer to machine code and harder to read (e.g., Assembly language).
  - **Interpreted vs. Compiled Languages:**
    - **Interpreted:** The code is executed line by line (e.g., Python).
    - **Compiled:** The code is translated into machine language before execution (e.g., C).
- **Why Learn Programming?**
  - Problem-solving skills.
  - Automating tasks.
  - Building applications.
  - Job opportunities in software development.

Overview of C Programming

- **Introduction to C:**
  - Developed by Dennis Ritchie in 1972.
  - Widely used for system/software development.
  - Known for its efficiency and control over system resources.
  - Foundation for many modern languages like C++, Java, and Python.
- **Key Features of C:**
  - **Procedural Language:** Emphasizes functions and control flow.
  - **Low-Level Access to Memory:** Direct manipulation of hardware via pointers.
  - **Portability:** Write once, compile anywhere.
  - **Rich Library Support:** Standard C Library provides various functions.
  - **Modular Structure:** Programs can be broken into functions for easier management.
- **Common Uses of C:**
  - Operating systems.
  - Embedded systems.
  - System programming.
  - Compilers and interpreters.
  - Network drivers.

Installing a C Compiler (e.g., GCC)

- **What is a Compiler?**
  - A compiler is a software that converts code written in a high-level programming language into machine code that a computer's processor can execute.

Writing Your First "Hello, World!" Program

1. **Write the Code:**

   ```
   #include <stdio.h>
   int main()
   {
       printf("Hello, World!\n");
       return 0;
   }
   ```

   - **Explanation:**
     - `#include <stdio.h>`: This preprocessor directive tells the compiler to include the standard input/output library.
     - `int main() { ... }`: The main function is the entry point of any C program.
     - `printf("Hello, World!\n");`: This function prints "Hello, World!" to the console.
     - `return 0;`: Indicates that the program finished successfully.
2. **Save the File:**
   - Save the file with a `.c` extension, for example, `hello_world.c`.
3. **Compile the Program:**
   - Open the terminal or command prompt.
   - Navigate to the directory where the file is saved.
   - Type the command: `gcc hello_world.c -o hello_world`
   - This will compile the code and create an executable file named `hello_world`.
4. **Run the Program:**
   - In the terminal or command prompt, type: `./hello_world`
   - You should see the output: `Hello, World!`

Basic understanding of printf and scanf for I/O:

`printf` and **`scanf`** are two fundamental functions in C used for input and output (I/O) operations.

printf:

- `printf` is used to print text or data to the console.
- returns an int indicating the number of characters printed (excluding the null byte used to end output to strings)
- If any error then returns 0

scanf:

- `scanf` is used to read input from the user.
- It converts string to char, int, long, float, double and sets the value of the pointer located at the argument.In case of string it simply copies the string to the output.
- The & operator is used to get the memory address of the variable, which allows the scanf() function to modify it.
- The function returns the number of input items successfully matched and assigned.

Comments and formatting code:

Comments in Code:

- Comments are used to provide explanations or notes without affecting the program's execution.
- C supports two types of comments:
  - **Single-line comments**: Use `//` to comment out a single line
  - **Multi-line comments**: Use `/* ... */` to comment out multiple lines.

Formatting of Code:

- Proper code formatting is crucial for readability and collaboration.
- Key elements of formatting in C:
  - Indentation:
    - Indent code blocks consistently using spaces or tabs. The standard practice is 4 spaces per indentation level.
  - Braces:
    - Opening Brace Placement:
      - The opening brace `{` can be placed on the same line as the control statement or on the next line, depending on your style preference or the project's style guide.
    - Closing Brace:
      - Always align the closing brace `}` with the line containing the corresponding opening statement (e.g., `if`, `for`, `while`, `function definition`).
  - Spacing:
    - Use spaces around operators (`=`, `+`, `-`, etc.), after commas, and between function arguments.
  - Blank Lines:
    - Use blank lines to separate logical sections of code, such as between functions or to separate variable declarations from code.
  - Consistent Naming Conventions:
    - Use meaningful and consistent naming conventions for variables, functions, and other identifiers.

Variables in C Language:

Variables are the storage areas in a code that the program can easily manipulate.

Syntax:

```
type variable_name;
```

Example:

```
int age;

float salary;

char grade;
```

Rules for Naming Variables:

- Must begin with a letter (a-z, A-Z) or an underscore (_).
- Subsequent characters can be letters, digits (0-9), or underscores.
- No special characters allowed (e.g., `@`, `#`, `!`).
- Case-sensitive (`Age` and `age` are different).
- Cannot use reserved keywords (e.g., `int`, `return`).

Types of Variables:

- **Local Variables:** Declared inside a function or block and accessible only within that function/block.
- **Global Variables:** Declared outside any function and accessible by any function in the program.
- **Static Variables:** Retain their value between function calls.
- **Register Variables:** Stored in CPU registers instead of RAM for faster access.

Data Types:

data types determine the type of data that can be stored in a variable and the operations that can be performed on that data.

Size of char: 1 byte

Size of int: 4 bytes

Size of short: 2 bytes

Size of long: 8 bytes

Size of long long: 8 bytes

Size of float: 4 bytes

Size of double: 8 bytes

Size of long double: 16 bytes

Size of void * (pointer): 8 bytes

Program to find size of each data type:

```
#include <stdio.h>

int main()

{

   printf("Size of char: %zu bytes\n", sizeof(char));

   printf("Size of int: %zu bytes\n", sizeof(int));

   printf("Size of short: %zu bytes\n", sizeof(short));

   printf("Size of long: %zu bytes\n", sizeof(long));

   printf("Size of long long: %zu bytes\n", sizeof(long long));

   printf("Size of float: %zu bytes\n", sizeof(float));

   printf("Size of double: %zu bytes\n", sizeof(double));

   printf("Size of long double: %zu bytes\n", sizeof(long double));

   printf("Size of void * (pointer): %zu bytes\n", sizeof(void *));

   return 0;

}
```

Type Modifiers: Modifiers are used to alter the size and range of basic data types.

- **short, long:** Modifies the size of integers.
- **signed, unsigned:** Specifies whether the number can be negative.

Example:

```
short int a;        // Short integer (typically 2 bytes)

long int b;         // Long integer (typically 4 bytes)

unsigned int c;     // Unsigned integer (only positive values)
```

Data Type	Size in bytes	Range (signed)	Range(Unsigned)
char	1	-128 to +127	0 to 255
short int	2	-2^{15} to $+2^{15}-1$	0 to $2^{16}-1$
int(16 bit)	2	-2^{15} to $+2^{15}-1$	0 to $2^{16}-1$
int(32bit)	4	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
long int(32 bit)	4	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
long int(64 bit)	8	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
long long int	8	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$

Format Specifiers:

A **format specifier** in C is a special sequence of characters used in functions like `printf` and `scanf` to specify how to interpret or display data of various types.

Specifier	Data Type	Description
`%d` or `%i`	int	Signed integer
`%u`	unsigned int	Unsigned integer
`%f`	float , double	Floating-point number
`%c`	char	Single character
`%s`	`char[]` (string)	String of characters
`%lf`	double	Double precision floating-point number
`%ld`	long int	Long integer
`%lld`	long long int	Long long integer
`%x or %X`	unsigned int	Unsigned integer (hexadecimal)
`%o`	unsigned int	Unsigned integer (octal)
`%%`		Prints a literal percent sign (%)
`%p`	pointer	Address of pointer

Operators:

1. **Arithmetic Operators:**

Arithmetic operators are used to perform basic mathematical operations.

- **Addition (+)**: Adds two operands.

```
int a = 5 + 3; // a = 8
```

- **Subtraction (-)**: Subtracts the second operand from the first.
```
int a = 5 - 3; // a = 2
```
- **Multiplication (*)**: Multiplies two operands.
```
int a = 5 * 3; // a = 15
```

- **Division (/)**: Divides the first operand by the second (result is the quotient).
  ```
  int a = 5 / 3; // a = 1
  ```
- **Modulus (%)**: Divides the first operand by the second (result is the remainder).
  ```
  int a = 5 % 3; // a = 2
  ```

2. **Relational Operators:**

Relational operators are used to compare two values. They return `1` (true) if the comparison is true, otherwise `0` (false).

- **Equal to (==)**: Checks if two values are equal.
  ```
  if (a == b)
  ```
- **Not equal to (!=)**: Checks if two values are not equal.
  ```
  if (a != b)
  ```
- **Less than (<)**: Checks if the first value is less than the second.
  ```
  if (a < b)
  ```
- **Greater than (>)**: Checks if the first value is greater than the second.
  ```
  if (a > b)
  ```
- **Less than or equal to (<=)**: Checks if the first value is less than or equal to the second.
  ```
  if (a <= b)
  ```
- **Greater than or equal to (>=)**: Checks if the first value is greater than or equal to the second.
  ```
  if (a >= b)
  ```

3. **Assignment Operator**

The assignment operator is used to assign the value on the right to the variable on the left.

- **Assignment (=)**: Assigns the value on the right to the variable on the left.

  ```
  int a = 5;
  ```

4. Increment and Decrement Operators

These operators are used to increase or decrease the value of a variable by 1.

- **Increment (++)**: Increases the value of the operand by 1.

  ```
  int a = 5;
  a++; // a = 6
  ```

- **Decrement (--)**: Decreases the value of the operand by 1.

  ```
  int a = 5;
  a--; // a = 4
  ```

5. Logical Operators

Logical operators are used to combine multiple conditions.

- **Logical AND (`&&`)**: Returns true if both conditions are true.

A	B	A &&B
False	False	False
False	True	False
True	False	False
True	True	True

```
if (a > 0 && b > 0)
```

- **Logical OR (`||`)**: Returns true if at least one of the conditions is true.

A	B	A || B
False	False	False
False	True	True
True	False	True
True	True	True

```
if (a > 0 || b > 0)
```

- **Logical NOT (`!`)**: Reverses the logical state of its operand.

A	!A
False	True
True	False

```
if (!a) // true if a is 0
```

6. Bitwise Operators

Bitwise operators are used to perform operations on the bits of a number.

- **Bitwise AND (`&`)**: Performs a bitwise AND operation.

A	B	A &B
False	False	False
False	True	False
True	False	False
True	True	True

```
int a = 5 & 3; // a = 1 (0101 & 0011 = 0001)
```

- **Bitwise OR (|)**: Performs a bitwise OR operation.

A	B	A | B
False	False	False
False	True	True
True	False	True
True	True	True

```
int a = 5 | 3; // a = 7 (0101 | 0011 = 0111)
```

- **Bitwise XOR (^)**: Performs a bitwise XOR operation.

A	B	A ^ B
False	False	False
False	True	True
True	False	True
True	True	False

```
int a = 5 ^ 3; // a = 6 (0101 ^ 0011 = 0110)
```

- **Bitwise NOT (~)**: Performs a bitwise NOT operation.

```
int a = ~5; // a = -6 (inverts all bits)
```

- **Left Shift (<<)**: Shifts the bits of the operand to the left by a specified number of positions.
  - **x<<k = x * (2^k)**

```
int a = 5 << 1; // a = 10 (0101 << 1 = 1010)
```

- **Right Shift (>>):** Shifts the bits of the operand to the right by a specified number of positions.
  - **x>>k = x / (2^k)**

```
int a = 5 >> 1; // a = 2 (0101 >> 1 = 0010)
```

7. Ternary Operator

The ternary operator is a type of alternate for if-else statements.

- **Ternary (? :)**: Evaluates a condition and returns one of two values based on the condition.

```
int a = (b > 0) ? 1 : -1; // a = 1 if b > 0, else a = -1
```

Number System:

The number system is a way to represent numbers in different forms or bases. There are several types of number systems, each with a different base.

Decimal Number System (Base 10):

- **Digits Used:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Base:** 10
- **Description:** The most commonly used number system in everyday life. Each digit in a decimal number represents a power of 10.
- **Example:** 235 in decimal can be represented as:

$$(235)_{10} = (5 * 10^0) + (3*10^1) + (2*10^2)=5+30+200=(235)_{10}$$

Binary Number System (Base 2):

- **Digits Used:** 0, 1
- **Base:** 2
- **Description:** The binary system is used internally by almost all modern computers and computer-based devices because it is simple and can easily represent two states (on and off).
- **Example:** 1011 in binary can be represented as:

$$(1011)_2 = (1 * 2^0) + (1*2^1) + (0*2^2)+(1*2^3) = 1+2+0+8 = (11)_{10}$$

Octal Number System (Base 8):

- **Digits Used:** 0, 1, 2, 3, 4, 5, 6, 7
- **Base:** 8
- **Description:** The octal system is less commonly used but can be found in some computing environments. It was more common in older computer systems.
- **Example:** 125 in octal can be represented as:

$$(125)_8 = (5 * 8^0) + (2*8^1) + (1*8^2)=5+16+64=(85)_{10}$$

HexaDecimal Number System (Base 16):

- **Digits Used:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,A,B,C,D,E,F
- **Base:** 16
- **Description:** The hexadecimal system is used extensively in computing because it is more compact than binary and easy to convert from binary. Each hexadecimal digit represents four binary digits (bits).
- **Example:** 235 in decimal can be represented as:

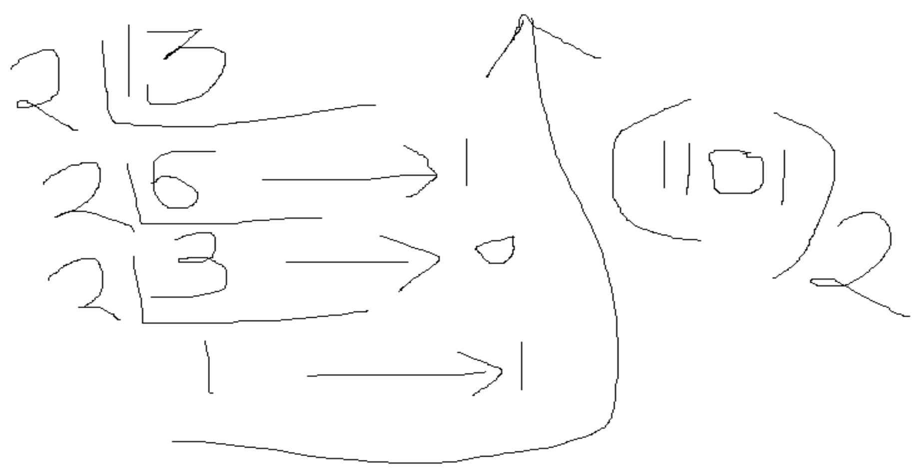
$$(2F)_{16} = (F * 16^0) + (2*16^1) = F+32 = 15+32 = (47)_{10}$$

Formula to Convert **Base-x** to **Decimal**:

$$\mathbf{(abcd)_x = d*x^0 + c*x^1 + b*x^2 + a*x^3}$$

Example:

Convert Decimal number 13_{10} to Binary Conversion



Power of 2 formual- if **(N&(N-1)==0)** then the number is **power of 2** where **N** is an integer number

Control Statements:

1. if Statement:

The simplest conditional control structure that executes a block of code if a condition is true.

Syntax:

```
if (condition)
{
    // code to be executed if condition is true
}
```

Example:

```
int a = 10;
if (a > 5)
{
    printf("a is greater than 5\n");
}
```

2. if-else Statement:

Executes one block of code if the condition is true, otherwise executes another block.

Syntax:

```
if (condition)
{
    // code to be executed if condition is true
}
else
{
    // code to be executed if condition is false
}
```

Example:

```
int a = 4;
if (a > 5)
{
    printf("a is greater than 5\n");
}
else
{
    printf("a is less than or equal to 5\n");
}
```

3. if-else-if Ladder:

Used when there are multiple conditions. It checks each condition in sequence.

Syntax:

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

Example:

```
int a = 20;
if (a == 10) {
    printf("a is 10\n");
}
else if (a == 20) {
    printf("a is 20\n");
}
else if (a == 30) {
    printf("a is 30\n");
}
else {
    printf("a is something else\n");
}
```

4. Nested if:

An `if` statement inside another `if` statement.

Syntax:

```
if (condition1)
{
    if (condition2)
    {
      // code to be executed if both condition1 and condition2 are true
     }
}
```

Example:

```
int a = 10, b = 20;
if (a == 10) {
    if (b == 20) {
        printf("a is 10 and b is 20\n");
    }
}
```

5. switch Statement:

The `switch` statement is used to select one of many blocks of code to be executed based on the value of a variable.

Syntax:

```
switch (expression)
{
    case constant1:
        // code to be executed if expression equals constant1
        break;

    case constant2:
        // code to be executed if expression equals constant2
        break;

    ...
    default:
        // code to be executed if expression doesn't match any case
}
```

Important Points:

- The `if-else` structure can handle complex conditions, while `switch` is limited to evaluating a single variable against constant values.
- `switch` is generally used for equality comparison.
- The `break` statement is used to exit the switch case after a match is found.
- The `default` case is optional and is executed if no matching case is found.
- `break` and `default` keyword are optional, not compulsory to use.
- if `break` keyword is not present,then all the cases will be executed from matching case until `break` found or switch case ends.

Example:

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

Difference Between switch case and if-else

Feature	**If-else**	**switch**
Type of Condition	Can handle complex conditions (relational, logical)	Only equality comparison is allowed
Efficiency	Becomes slower with many conditions	Faster for fixed number of conditions
Datatypes	Supports all data types	Works with integer, char, and enum