# Module 2 – Introduction to Programming

## 1.Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

C is a foundational programming language, developed by Dennis Ritchie in the early 1970s at Bell Labs, known for its efficiency, flexibility, and low-level access. It has been incredibly influential, shaping the development of numerous other languages like C++, Java, and Python.

### Evolution of C Programming

Since its inception, C has undergone several standardizations to ensure consistency and portability across different platforms. Some key developments include:

- **K&R C (1978):** The first widely recognized version, introduced in the book "*The C Programming Language*" by Brian Kernighan and Dennis Ritchie.
- **ANSI C (1989):** The American National Standards Institute (ANSI) standardized C to resolve differences among various implementations.
- **C89/C90:** These terms refer to the ANSI standard and its subsequent adoption by the International Organization for Standardization (ISO).
- **C99 (1999):** Added features like inline functions, new data types (e.g., long long int), and improved support for floating-point operations.
- **C11 (2011):** Focused on enhancing multi-threading support and security features.
- **C18 (2018):** Mostly a bug-fix version of C11, ensuring stability and consistency.

### Importance of C Programming

**Foundation for Other Languages:** Languages like C++, Java, and even Python have been heavily influenced by C. Understanding C helps programmers grasp core programming concepts such as memory management and data structures.

**System-Level Programming:** C is the preferred language for developing operating systems, device drivers, and embedded software due to its ability to directly manipulate hardware resources.

**Portability and Efficiency:** C programs can be compiled and run on various hardware platforms with minimal changes, and its efficiency makes it ideal for performance-critical applications.

**Educational Value:** C is often taught as an introductory programming language because it provides a solid foundation in procedural programming and exposes students to how computers work at a low level.

## 2.Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.

**Option 1: Dev-C++ (Simple and Lightweight)**

1. **Download Dev-C++**:
   - From: https://sourceforge.net/projects/orwelldevcpp/
2. **Install Dev-C++**:
   - Run the setup and follow the instructions.
   - It includes a built-in GCC compiler.
3. **Write and run code**:
   - Open Dev-C++, create a new C project.
   - Write your code, press **F9** to compile and run.

**Option 2: VS Code (Flexible and Modern)**

1. **Install VS Code**:
   - From: https://code.visualstudio.com/
2. **Install the C/C++ Extension**:
   - Open VS Code → Extensions (Ctrl+Shift+X).
   - Search for and install **"C/C++"** by Microsoft.
3. **Install GCC (if not already done)**:
   - Use MinGW as explained above.
4. **Set up tasks and launch configurations**:
   - Create a .vscode folder in your project.
   - Add tasks.json and launch.json to build and run your code.
   - Or use extensions like **Code Runner** to run C code easily.
5. **Write and run code**:
   - Create a .c file and run using terminal or Code Runner.

**Option 3: Code::Blocks (All-in-One Solution)**

1. **Download Code::Blocks with GCC included**:
   - From: https://www.codeblocks.org/downloads/
   - Choose the version labeled: **"codeblocks-20.03mingw-setup.exe"**
2. **Install Code::Blocks**:
   - Run the installer.
   - During first launch, it will auto-detect the GCC compiler.
3. **Write and run code**:
   - Create a new C project.
   - Write your code.
   - Press **F9** to compile and run.

## 3.Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

the **basic structure of a C program**, including its key components like **headers**, **main function**, **comments**, **data types**, and **variables**, along with clear examples.

**1. Header Files**

- Syntax: #include <header_name>
- Purpose: Includes libraries for built-in functions (like printf()).

**2. Comments**

- Comments are ignored by the compiler. Used to explain code.
- Two types:
    - **Single-line:** // This is a comment
    - **Multi-line:**

```
/* This is a
   multi-line comment */
```

### 3. Main Function

- Entry point where program execution starts.
- Every C program **must** have a main() function.

**Syntax:**

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

### 4. Data Types

Used to declare what type of data a variable will hold.

| Data Type | Description | Example |
|---|---|---|
| int | Integer values | int x = 10; |
| float | Decimal numbers | float y = 5.5; |
| char | Single characters | char c = 'A'; |
| double | Large decimal values | double d=3.14159; |

## 5. Variables

- Named storage used to hold data.
- Must be declared before use.
- Follows this format:
  data_type variable_name = value;

**Examples:**

```
int age = 20;
float weight = 55.5;
char grade = 'A';
```

## Full Example Program

```
#include <stdio.h>

int main() {
   // Declare variables
   int a = 10;
   float b = 3.14;
   char c = 'Z';

   // Print values
   printf("Integer: %d\n", a);
   printf("Float: %.2f\n", b);
   printf("Character: %c\n", c);

   return 0;
}
```

Output:

```
Integer: 10
Float: 3.14
Character: Z
```

## 4.Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

## 1. Arithmetic Operators

Used to perform basic mathematical operations.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| * | Multiplication | 5 * 3 | 15 |
| / | Division | 10 / 2 | 5 |
| % | Modulus | 10 % 3 | 1 (remainder) |

## 2. Relational Operators

Used to compare two values. The result is either **true (1)** or **false (0)**.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | Equal to | 5 == 5 | 1 |
| != | Not equal to | 5 != 3 | 1 |
| > | Greater than | 5 > 3 | 1 |
| < | Less than | 5 < 3 | 0 |
| >= | Greater than or equal | 5 >= 5 | 1 |
| <= | Less than or equal | 5 <= 2 | 0 |

## 3. Logical Operators

Used to combine multiple conditions.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| && | Logical AND | (1 && 0) | 0 |
| \|\| | Logical OR | (1 \|\| 0) | 1 |
| ! | Logical NOT | !(1) | 0 |

## 4. Assignment Operators

Used to assign values to variables.

| Operator | Meaning | Example |
|---|---|---|
| `=` | Assign value | `a = 10` |
| `+=` | Add and assign | `a += 5` → `a = a + 5` |
| `-=` | Subtract and assign | `a -= 3` → `a = a - 3` |
| `*=` | Multiply and assign | `a *= 2` |
| `/=` | Divide and assign | `a /= 2` |
| `%=` | Modulus and assign | `a %= 3` |

## 5. Increment/Decrement Operators

Used to increase or decrease the value of a variable by 1.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| `++` | Increment | `a++` or `++a` | Adds 1 |
| `--` | Decrement | `a--` or `--a` | Subtracts 1 |

## 6. Bitwise Operators

Operate at the **bit level** of data (used for embedded and low-level programming).

| Operator | Meaning | Example | Result |
|---|---|---|---|
| `&` | AND | `5 & 3` | 1 |
| ` | ` | ` | OR | `5 |
| `^` | XOR | `5 ^ 3` | 6 |
| `~` | NOT (1's complement) | `~5` | -6 |

| Operator | Meaning | Example | Result |
|---|---|---|---|
| << | Left shift | `5 << 1` | 10 |
| >> | Right shift | `5 >> 1` | 2 |

### 7. Conditional Operator (Ternary Operator)

A shorthand for `if-else` statements.

## 4.Explain decision-making statements in C (if, else, nested if-else, switch).Provide examples of each.

### ● If Statements

● if statement is the basic decision making statement
● Used to decide whether a certain statement or block of statements will be executed or notIf Statements

### Syntax :

```
if( condition )
{
statement_1 ; // true block
statements
}
statement x ;
```

### If else Statements

if else statement allows selecting any one of the two available options depending upon the output of the test condition

### Syntax :

```
if (condition )
{
statements;
}
else
{
statements ; // false statement
}
```

## Nested If Statements

Nested if statement is simply an if statement embedded with an another if statement

● Syntax :

// executes when condition1 is

true

// executes when condition2 is

true

if (condition1)

{

statements ;

if ( condition2)

{

statements ;

}

}

## Switch Statements

Switch case statements are a substitute for long if statements that compare
a variable to several integer values

● Syntax :
// executed when n =

1

// executed when n =

2

// executed when n doesn’t match any

case

```
switch ( n)

{

case 1 :

break;

case 2 :

break ;

default :

}
```

## 6. Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

**1.While Loops -** It repeatedly executes a target statement as long as the condition is true.

**Syntax :**

```
while (condition )
{
Body of loop
}
```

### ● All while loops are executed in following sequence :

Step 1 : It checks the test

Condition  if true then

go to Step 2  other wise

to Step 3

Step 2 : Executes body of loop and go to Step 1.

Step 3 : Other statements of program.

**2.For Loops** - It is a repetition control structure that allows you to efficiently write

loop that neaeds to execute a specific number of times.

## Syntax :

for ( initialization ;test condition ; increment )

{

Body of loop

}

● **All for loops are executed in following sequence :**

Step 1 : It executes initialization

statements Step 2 : It checks the condition;

if true then go to Step

3 other wise Step

4

Step 3: Executes loop and go to Step 2

Step 4 : Out of the Loop

**3.Do-while Loops** - Do-while loop is similar to while loop , except the fact that it execute once even if condition is false.

● **Syntax :**

do

{

body of loop

} while (condition ) ;

- **All do while loops are executed in following sequence :**

Step 1 : Executes the body of loop and go to Step 2.

Step 2 : It checks the test

condition if true then

go to Step 1 otherwise go

to Step 3.

Step 3: Other statements of the program .

## 7.Explain the use of break, continue, and goto statements in C. Provide examples of each.

Break Statement:

- The break statement is used inside loop or switch statement.
- When compiler finds the break statement inside a loop, compiler will

abort the loop and continue to execute statements followed by loop.

Syntax: break ;

Example :

```
#include <stdio.h>
int main() {
   for (int i = 1; i <= 10; i++) {
```

```
        if (i == 5) {
            break;  // Exit the loop when i is 5
        }
        printf("%d\n", i);
    }
    return 0;
}
```

## Continue statement:

• The continue statement is also used inside loop.

• When compiler finds the continue statement inside a loop, compiler will skip all the following statements in the loop and resume the next loop iteration.

Syntax: continue ;

Example :

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // Skip this iteration when i is 3
        }
        printf("%d\n", i);
    }
    return 0;
}
```

## The GOTO statement:

• By using this goto statements we can transfer the control from current location to anywhere in the program.

• To do all this we have to specify a label with goto and the control will transfer to the location where the label is specified.

## Syntax :

```
goto label_name;


// some code


label_name:
   // code to execute after jump
```

## Example :

```
#include <stdio.h>
int main() {
   int num = 3;


   if (num == 3) {
      goto skip;  // Jump to the label 'skip'
   }


   printf("This line will be skipped.\n");


skip:
   printf("Jumped to the label using goto.\n");
```

```
    return 0;
}
```

## 8.What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

What are Functions in C?

A **function** is a reusable block of code that performs a specific task.
It helps in:

Breaking a large program into smaller parts

Avoiding code repetition

Making programs easier to understand and debug

**1. Function Declaration (Prototype)**

Tells the compiler:

The function name

Return type

Parameters (if any)

Syntax:

return_type function_name(parameter_list);

Example:

int add(int a, int b);  // Declares a function that takes two integers and returns an int

**2. Function Definition**

This is the **actual code** or logic of the function.

Syntax:

```
return_type function_name(parameter_list) {
   // function body
   return value;
}
```

Example:

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

**3. Calling a Function**

You **call** (use) the function in the main() function or any other function.

Syntax:

```
function_name(arguments);
```

Example:

```
int result = add(5, 3);  // Calls the add() function
```

Example

```
#include <stdio.h>

// Function Declaration
int add(int a, int b);
```

```
int main() {
   int sum;

   // Function Call
   sum = add(10, 20);

   printf("Sum is: %d\n", sum);

   return 0;
}

// Function Definition
int add(int a, int b) {
   return a + b;
}
```

**Output:**

Sum is: 30

## 9.Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

An **array** is a collection of **similar data types** stored in **contiguous memory locations**.
Instead of creating multiple variables, you can store multiple values in a **single array variable**.

**Why Use Arrays?**

To store a list of items like marks, names, numbers.

Efficiently manage and process collections of data.

One-Dimensional Array (1D Array)

A 1D array stores a list of elements in a single row (like a simple list).

Declaration:

```
data_type array_name[size];
```

Example:

```
int marks[5] = {85, 90, 75, 88, 92};
```

Accessing elements:

```
printf("%d", marks[2]);  // Output: 75 (indexing starts from 0)
```

Loop Example:

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

Multi-Dimensional Array (2D Array)

A multi-dimensional array (usually 2D) is like a table or matrix with rows and columns.

Declaration:

```
data_type array_name[rows][columns];
```

Example:

```
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

Accessing elements:

```
printf("%d", matrix[1][2]);  // Output: 6 (2nd row, 3rd column)
```

Loop Example:

```
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 3; j++) {
      printf("%d ", matrix[i][j]);
   }
   printf("\n");
}
```

## 10.Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

A pointer is a variable that stores the memory address of another variable.

Instead of holding a value directly (like int x = 10;), a pointer holds the location in memory where the value is stored.

Example:

```
int x = 10;
int *ptr = &x;
```

x is a normal integer variable.

&x gives the **address of x**.

ptr is a pointer to an integer, storing the address of x.

Pointer Declaration and Initialization

Declaration Syntax:

data_type *pointer_name;

Initialization Syntax:

pointer_name = &variable_name;

Full Example:

```
#include <stdio.h>

int main() {
   int a = 5;
   int *p;        // pointer declaration
   p = &a;        // pointer initialization

   printf("Value of a: %d\n", a);       // 5
   printf("Address of a: %p\n", &a);    // e.g. 0x7ffc...
   printf("Value using pointer: %d\n", *p); // 5

   return 0;
}
```

Key Symbols

| Symbol | Meaning |
| --- | --- |
| * | Declares a pointer or dereferences it |
| & | Gets the address of a variable |

Why Are Pointers Important in C?

**Efficient Memory Access**

Allows direct access to memory, making C a low-level and powerful language.

**Function Arguments (Call by Reference)**

You can modify actual values of variables inside functions using pointers.

**Dynamic Memory Allocation**

With functions like malloc(), calloc() from <stdlib.h>, pointers are essential.

**Working with Arrays and Strings**

Arrays are closely related to pointers; they allow efficient iteration and manipulation.

**Building Complex Data Structures**

Pointers are required for **linked lists, trees, graphs**, and more.

## 11. Explain string handling functions like strlen(), strcpy(), strcat(),strcmp(), and strchr(). Provide examples of when these functions are useful.

string handling functions in C from the <string.h> library. These functions help you manipulate and process strings efficiently.

### 1. strlen() – String Length

Purpose:

Returns the **length** of a string (number of characters, **excluding \0** null terminator).

Syntax:

```
int strlen(const char *str);
```

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

   char name[] = "Hello";
```

```
    printf("Length = %lu\n", strlen(name));  // Output: 5
    return 0;
}
```

**Use Case**: To check input length or validate passwords, usernames, etc.

2. strcpy() – String Copy

Purpose:

Copies the content of one string into another.

Syntax:

```
char *strcpy(char *dest, const char *src);
```

Example:

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

int main() {
    char src[] = "C Language";
    char dest[50];

    strcpy(dest, src);
    printf("Copied String: %s\n", dest);
    return 0;
}
```

**Use Case**: Duplicating a string or saving user input to another variable.

3. strcat() – String Concatenation

Purpose:

**Appends** one string at the **end of another**.

Syntax:

```
char *strcat(char *dest, const char *src);
```

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

   char str1[50] = "Hello ";

   char str2[] = "World!";


   strcat(str1, str2);

   printf("Concatenated: %s\n", str1);  // Output: Hello World!

   return 0;

}
```

**Use Case**: Building messages like "Hello " + username.

4. strcmp() – String Comparison

Purpose:

Compares two strings **character by character**.

Syntax:

```
int strcmp(const char *str1, const char *str2);
```

Returns:

0 → if both strings are **equal**

<0 → if str1 < str2

>0 → if str1 > str2

Example:

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

int main() {
   char a[] = "apple";
   char b[] = "apple";
   char c[] = "banana";

   printf("%d\n", strcmp(a, b));  // Output: 0
   printf("%d\n", strcmp(a, c));  // Output: negative number
   return 0;
}
```

**Use Case**: Checking user input (e.g., password match, search word).

5. strchr() – Find Character in String

Purpose:

Searches for the **first occurrence of a character** in a string.

Syntax:

```
char *strchr(const char *str, int ch);
```

Example:

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

int main() {
```

```
    char text[] = "hello world";
    char *pos = strchr(text, 'w');

    if (pos != NULL) {
        printf("Character found at position: %ld\n", pos - text);
    } else {
        printf("Character not found.\n");
    }

    return 0;
}
```

**Use Case**: Finding a delimiter (e.g., @ in email), or character location.

## 12. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

A **structure** in C is a user-defined data type that allows grouping variables of different types under one name. It is used to represent a record, such as a student, employee, book, etc.

Structures help organize complex data, especially when you want to store and process multiple pieces of related information together.

### Structure Declaration

```
struct Student {
    int roll;
    char name[50];
    float marks;
};
```

Here:

Student is the name of the structure.

It contains three members: roll, name, and marks.

## Declaring Structure Variables

You can declare structure variables in two ways:

```
// Method 1: Separate declaration

struct Student s1;


// Method 2: With the structure

struct Student {

   int roll;

   char name[50];

   float marks;

} s1, s2;
```

## Initializing a Structure

You can initialize a structure at the time of declaration or later.

```
// At declaration

struct Student s1 = {1, "Rahul", 85.5};


// After declaration

struct Student s2;

s2.roll = 2;

strcpy(s2.name, "Priya");

s2.marks = 90.0;
```

Use strcpy() to assign strings to character arrays.

## Accessing Structure Members

Use the dot (.) operator with structure variables:

```
printf("Roll: %d\n", s1.roll);

printf("Name: %s\n", s1.name);

printf("Marks: %.2f\n", s1.marks);
```

## 13.Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

**File handling** in C allows a program to store output and retrieve input from files (like **.** txt, .csv, etc.), enabling **permanent storage** of data.

**Importance of File Handling**

**Permanent Storage**: Keeps data even after the program ends.

**Large Data Handling**: Manage large amounts of data more efficiently than with variables.

**Data Sharing**: Files can be shared between different programs.

**Logging**: Useful for keeping logs, backups, and reports.

**1. Opening a File:** fopen()

```
FILE *fp;

fp = fopen("data.txt", "w");  // Open for writing
```

**2. Writing to a File:** fprintf() **/** fputs()

```
FILE *fp = fopen("data.txt", "w");

fprintf(fp, "Hello, world!\n");

fputs("This is a file.\n", fp);

fclose(fp);
```

**3. Reading from a File:** fscanf() **/** fgets()

```
FILE *fp = fopen("data.txt", "r");
```

```
char str[100];

int num;


fscanf(fp, "%s", str);  // Read a word

fgets(str, 100, fp);    // Read a line

fclose(fp);
```

**4. Closing a File:** fclose()

```
fclose(fp);  // Always close the file to save resources
```