**Module 2 – Introduction to Programming**

1. **Overview of C Programming**

**C programming** is a powerful, general-purpose programming language developed in the early 1970s by **Dennis Ritchie** at **Bell Labs**. It is known for its **speed, efficiency, and control over system resources**, making it widely used in system/software development.

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

#### **Introduction**

The **C programming language** is one of the most influential and widely used programming languages in the history of computer science. Developed more than five decades ago, C has played a crucial role in shaping modern computing. Despite the rise of newer programming languages, C remains an essential language due to its efficiency, portability, and close relationship with hardware.

#### **History and Evolution of C Programming**

The roots of C trace back to the late **1960s** and early **1970s**, during the early days of computer systems. Here's a brief timeline of its evolution:

1. **BCPL and B (1966–1969):**
   * The story begins with **BCPL (Basic Combined Programming Language)**, developed by **Martin Richards**.
   * Later, **Ken Thompson** at Bell Labs created a simplified version called **B**, which was used in early Unix development.
2. **Birth of C (1972):**
   * **Dennis Ritchie**, also at **Bell Labs**, built upon B to create a new language called **C** in 1972.
   * C was developed to rewrite the **Unix operating system**, which was initially written in assembly language. This was a major milestone—**Unix became the first OS written in a high-level language**.
3. **Standardization and Spread (1980s):**
   * C quickly gained popularity in academia and industry.
   * In **1983**, the **American National Standards Institute (ANSI)** started working on standardizing C.
   * The first official standard, known as **ANSI C or C89**, was released in **1989**. It was later adopted as **ISO C (ISO/IEC 9899)**.
4. **Further Versions:**
   * **C99 (1999):** Added features like inline functions, variable-length arrays, and new data types.
   * **C11 (2011):** Improved multithreading support and safer programming features.
   * **C18 (2018):** Minor revisions and bug fixes.
5. **Setting Up Environmen**

**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**.

To write and run C programs, you need two things:

1. A **C compiler** (like **GCC**) to compile your code.
2. An **IDE** (like **DevC++**, **VS Code**, or **Code::Blocks**) to write and manage your code easily.

## 🔹 Step-by-Step Guide

### Option 1: **Install DevC++** (Beginner Friendly)

**DevC++** is an IDE that comes with a built-in compiler (GCC). Ideal for beginners.

#### Steps:

1. Go to: https://sourceforge.net/projects/orwelldevcpp/
2. Click **Download** and install the .exe file.
3. Follow the setup wizard (default settings are fine).
4. Launch DevC++.
5. Go to **File → New → Source File**, write your C code, and save with .c extension.
6. Press **F9** to compile and run the code.

### Option 2: **Install Code::Blocks with GCC**

**Code::Blocks** is another beginner-friendly IDE that can include GCC.

#### Steps:

1. Go to: https://www.codeblocks.org/downloads/
2. Download the **"codeblocks-XXmingw-setup.exe"** version (includes GCC compiler).
3. Run the installer and select default options.
4. Open Code::Blocks.
5. Go to **File → New → Project → Console Application → C**.
6. Write your code and press **F9** to build and run.

### Option 3: **Install VS Code with GCC (Advanced Setup)**

**Visual Studio Code (VS Code)** is a modern, powerful code editor. You need to manually install GCC and some extensions.

#### 🔽 Steps:

##### 1. **Install GCC Compiler (via MinGW):**

* Go to: <https://www.mingw-w64.org/>
* Download and install the version for Windows.
* During installation:
  + Architecture: x86\_64
  + Threads: posix
  + Exception: seh
* Add the bin folder (e.g., C:\Program Files\mingw-w64\...\bin) to your **System PATH**:
  + Right-click **This PC → Properties → Advanced System Settings → Environment Variables → PATH → Edit** and paste the path.

##### 2. **Verify GCC Installation:**

* Open **Command Prompt** and type:

css

gcc --version

If installed correctly, it will show the version.

##### 3. **Install VS Code:**

* Download from: <https://code.visualstudio.com/>
* Install and launch it.

##### 4. **Install C/C++ Extension:**

* Go to **Extensions** (Ctrl+Shift+X), search for **C/C++**, and install Microsoft’s C/C++ extension.

##### 5. **Create and Run a C File:**

* Create a new folder and open it in VS Code.
* Create a file program.c, write your code.
* Open the terminal in VS Code (Ctrl + ~) and compile:

bash

gcc program.c -o program

./program

## Summary

| **IDE** | **Compiler Needed** | **Difficulty** | **Best For** |
| --- | --- | --- | --- |
| **DevC++** | Built-in (GCC) | Easy | Beginners |
| **Code::Blocks** | Built-in (with MinGW) | Easy | Beginners to Intermediate |
| **VS Code** | Requires GCC setup | Medium | Intermediate to Advanced |
| 3. Basic Structure of a C Program |  |  |  |
|  |  |  |  |

**Explain the basic structure of a C program, including headers, main function,**

**comments, data types, and variables. Provide examples.**

## 1. **Header Files**

Header files contain declarations for functions and macros used in the program. The most common one is:

c

#include <stdio.h>

This includes the Standard Input/Output library, which allows functions like printf() and scanf().

## 2. **Main Function**

Every C program must have a main() function. It’s the entry point where the execution starts.

c

int main() {

// Code goes here

return 0;

}

## 3. **Comments**

Comments are used to describe the code. They are ignored by the compiler.

* **Single-line comment:**

c

// This is a single-line comment

* **Multi-line comment:**

c

/\* This is

a multi-line comment \*/

## 4. **Data Types**

C provides several data types, such as:

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| Int | Integer values | int age = 18; |
| Float | Decimal values (low precision) | float temp = 36.5; |
| Double | Decimal values (high precision) | double pi = 3.14159; |
| Char | Single character | char grade = 'A'; |

## 5. **Variables**

Variables are used to store data. You must declare the type before using them.

c

int age = 25;

float temperature = 37.5;

char grade = 'A';

**4. Operators in C**

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

## 1. **Arithmetic Operators**

These perform basic mathematical operations.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |

### Example:

c

int a = 10, b = 3;

printf("%d", a % b); // Output: 1

## 2. **Relational Operators**

These compare two values and return true (1) or false (0).

| **Operator** | **Description** | **Example** | |
| --- | --- | --- | --- |
| == | Equal to | a == b | |
| != | Not equal to | a != b | |
| > | Greater than | a > b | |
| < | Less than | a < b | |
| >= | Greater or equal | | a >= b | |
| <= | Less or equal | | a <= b | |

### Example:

c

if (a > b) {

printf("a is greater than b");

}

## 3. **Logical Operators**

Used to combine multiple conditions.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| && | Logical AND (both true) | (a > 5 && b < 10) |
| ` |  | ` |
| ! | Logical NOT (negation) | !(a > 5) |

## 4. **Assignment Operators**

Assign values to variables.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| = | Assign value | a = 5 |
| += | Add and assign | a += 2 → a = a + 2 |
| -= | Subtract and assign | a -= 2 |
| \*= | Multiply and assign | a \*= 2 |
| /= | Divide and assign | a /= 2 |
| %= | Modulus and assign | a %= 2 |
|  |  |  |

## 5. **Increment and Decrement Operators**

Used to increase or decrease a value by 1.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| ++ | Increment | a++ or ++a |
| -- | Decrement | a-- or --a |

* a++ → Post-increment (use a, then increment)
* ++a → Pre-increment (increment, then use a)

## 7. **Conditional (Ternary) Operator**

A shorthand for if-else statement.

**Syntax:**

c

(condition) ? expression1 : expression2;

### Example:

c

int max = (a > b) ? a : b;

1. **Control Flow Statements in C.**

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

## 1. if Statement

It checks a condition. If true, it executes the block.

**int num = 10;**

**if (num > 0) {**

**printf("Positive number\n");**

**}**

## 2. if-else Statement

It provides two paths: one if the condition is true, and another if it's false.

**int num = -5;**

**if (num >= 0) {**

**printf("Non-negative number\n");**

**} else {**

**printf("Negative number\n");**

**}**

## Nested if-else Statement

You can put one if or if-else inside another. Used for multiple conditions

**int num = 0;**

**if (num >= 0) {**

**if (num == 0) {**

**printf("Zero\n");**

**} else {**

**printf("Positive number\n");**

**}**

**} else {**

**printf("Negative number\n");**

**}**

## switch Statement

It is used to select one block of code from multiple options based on the value of a variable.

**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("Other day\n");**

**}**

| **Statement Type** | **Use Case Example** |
| --- | --- |
| if | Single condition |
| if-else | Two conditions (true/false) |
| Nested if-else | Multiple conditions/levels |
| switch | Multiple constant value checks |

1. **Looping in C**

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

* 1. while Loop

### Description:

* Checks the **condition first**.
* Executes the loop **only if** the condition is **true**.
* It's a **pre-test loop**.

int i = 1;

while (i <= 5) {

printf("%d ", i);

i++;

}

**2. For loop**

### Description:

* All loop control (start, end, update) is in one line.
* Also a **pre-test loop** (condition is checked first).
* Most compact loop.

### Example:

c

for (int i = 1; i <= 5; i++) {

printf("%d ", i);

}

### Best Use:

* When the number of iterations is **known**.
* Example: Printing the first 10 natural numbers.

## 3. do-while Loop

### 

### Description:

* Executes the loop **at least once**, then checks the condition.
* It's a **post-test loop**.

### Example:

c

int i = 1;

do {

printf("%d ", i);

i++;

} while (i <= 5);

### Best Use:

* When the loop **must run at least once**.
* Example: Menu-driven programs or user confirmation prompts.

**7. Loop Control Statements**

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

## break Statement in C

### Use:

* Immediately **exits** from the **loop** or a switch block.
* Control moves to the statement **after** the loop/switch.

### Example:

c

#include <stdio.h>

int main() {

for (int i = 1; i <= 5; i++) {

if (i == 3)

break; // Exit loop when i == 3

printf("%d ", i);

}

return 0;

}

### Output:

1 2

## continue Statement in C

### Use:

* **Skips** the **current iteration** of the loop.
* Loop continues with the **next iteration**.Example:

c

#include <stdio.h>

int main() {

for (int i = 1; i <= 5; i++) {

if (i == 3)

continue; // Skip printing when i == 3

printf("%d ", i);

}

return 0;

}

### Output:

1 2 4 5

## 🔹 goto Statement in C

### Use:

* Transfers control to a **labeled statement**.
* **Avoid** using unless absolutely necessary (makes code harder to read/debug).

### Example:

c

#include <stdio.h>

int main() {

int i = 1;

loop:

if (i <= 3) {

printf("%d ", i);

i++;

goto loop; // Jump back to the label 'loop'

}

return 0;

}

### Output:

* + 1. 2 3

**8.Functions in C**

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

In **C programming**, a **function** is a block of code that performs a **specific task**. Functions make code **modular, reusable, and easier to manage**.

*3 Main Parts of a Function*

### ****Function Declaration**** (also called function prototype)

* Tells the compiler about the **function name, return type, and parameters**.

### 2. ****Function Definition****

* Contains the **actual code** for the function.

### 3. ****Function Call****

* Invokes or **executes the function**.

c

#include <stdio.h>

// 1. Function declaration

int multiply(int, int);

int main() {

// 2. Function call

int result = multiply(4, 5);

printf("Product: %d\n", result);

return 0;

}

// 3. Function definition

int multiply(int a, int b) {

return a \* b;

}

**9. Arrays in C**

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

In **C programming**, an **array** is a **collection of elements** of the **same data type**, stored in **contiguous memory locations**. Arrays are used to **store multiple values** using a **single variable name**, with access via **indexing**.

## Concept of Arrays in C

* All elements must be of the **same type** (e.g., all int, all float).
* Indexing in C **starts from 0**.
* Arrays can be:
  + **One-dimensional** (1D)
  + **Multi-dimensional** (2D, 3D, etc.)

## **One-Dimensional Array (1D Array)**

### Declaration:

***int marks[5]; // Array of 5 integers***

**Initialization:**

***int marks[5] = {90, 85, 78, 88, 92};***

**Accessing Elements:**

***printf("%d", marks[2]); // Output: 78***

## 2. **Multi-Dimensional Arrays**

### Two-Dimensional Array (2D Array)

Used for **tables**, **matrices**, or **grids**.

### Declaration:

### int matrix[2][3]; // 2 rows, 3 columns

### Initialization:

### int matrix[2][3] = {

### {1, 2, 3},

### {4, 5, 6}

### };

### Accessing Elements:

### printf("%d", matrix[1][2]); // Output: 6

### Example:

### #include <stdio.h>

### int main() {

### int matrix[2][3] = {

### {1, 2, 3},

### {4, 5, 6}

### };

### for (int i = 0; i < 2; i++) {

### for (int j = 0; j < 3; j++) {

### printf("%d ", matrix[i][j]);

### }

### printf("\n");

### }

### return 0;

### }

**10. Pointers in C**

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

In **C programming**, a **pointer** is a **variable that stores the memory address** of another variable. Pointers are a powerful and essential feature of C that allow for efficient memory management, dynamic data structures, and faster program execution.

# What is a Pointer in C?

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

* Instead of holding a direct value like int a = 10;, a pointer holds the address like int \*p = &a;
* Pointers allow **direct access and manipulation** of memory

**Basic Terminologies**

| **Term** | **Description** |
| --- | --- |
| & (Address-of operator) | Returns the memory address of a variable. |
| \* (Dereference operator) | Accesses the value stored at a memory address. |
| int \*p; | Declares a pointer to an integer. |

# How to Use Pointers

## 1. Declaration

***int \*ptr; // pointer to an int***

***float \*fptr; // pointer to a float***

***char \*cptr; // pointer to a char***

2. Initialization

***int a = 10;***

***int \*p;***

***p = &a; // store address of 'a' in pointer 'p'***

3. Dereferencing (Accessing the Value)

printf("%d", \*p); // Outputs 10

**11. Strings in C**

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

In **C programming**, strings are handled as **character arrays** ending with a **null character '\0'**. The **string handling functions** provided in the <string.h> library help you work with strings efficiently.

## Common String Handling Functions in C

### 1. strlen() — String Length

**Purpose:**  
Returns the **number of characters** in a string **excluding** the null terminator.

**Example:**

c

#include <stdio.h>

#include <string.h>

int main() {

char name[] = "Meet";

printf("Length = %lu\n", strlen(name));

return 0;

}

**Output:**

ini

Length = 4

**Use Case:** Validating input length (e.g., password length).

### 2. strcpy() — Copy String

**Purpose:**  
Copies one string to another.

**Example:**

c

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "Hello";

char dest[20];

strcpy(dest, src);

printf("Destination: %s\n", dest);

return 0;

}

**Output:**

makefile

Destination: Hello

**Use Case:** Creating copies of strings for editing/backup.

3. strcat() — Concatenate Strings

**Purpose:**  
Appends one string to the end of another.

**Example:**

c

#include <stdio.h>

#include <string.h>

int main() {

char first[50] = "Good ";

char second[] = "Morning";

strcat(first, second);

printf("Combined: %s\n", first);

return 0;

}

**Output:**

makefile

Combined: Good Morning

**Use Case:** Building full messages, usernames, file paths.

### 4. strcmp() — Compare Strings

**Purpose:**  
Compares two strings **lexicographically** (like in a dictionary).

* Returns 0 if equal,
* < 0 if str1 < str2,
* > 0 if str1 > str2.

**Example:**

c

#include <stdio.h>

#include <string.h>

int main() {

char a[] = "apple";

char b[] = "banana";

int result = strcmp(a, b);

if (result == 0)

printf("Strings are equal\n");

else if (result < 0)

printf("a comes before b\n");

else

printf("a comes after b\n");

return 0;

}

**Output:**

a comes before b

**Use Case:** Sorting strings, comparing user input.

### 5. strchr() — Find Character in String

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

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char text[] = "Hello World";

char \*pos = strchr(text, 'o');

if (pos != NULL)

printf("'o' found at position: %ld\n", pos - text);

else

printf("Character not found\n");

return 0;

}

**Output:**

'o' found at position: 4

**12. Structures in C**

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

In **C programming**, a **structure** (struct) is a **user-defined data type** that allows grouping variables of **different types** under one name.

## What is a Structure?

A **structure** is used to represent a **record** — like a student, employee, book, etc., where each has multiple attributes of different data types.

### Real-Life Analogy:

A **student record** with fields:

* Name (string),
* Roll Number (int),
* Marks (float)

## Structure Declaration

c

struct Student {

int roll;

char name[50];

float marks;

};

* Student is the structure tag.
* Inside are the **members/fields** of the structure.

## Structure Variable Declaration

c

struct Student s1; // s1 is a variable of type struct Student

You can also combine declaration + variable:

c

struct Student {

int roll;

char name[50];

float marks;

} s1, s2; // declaring two variables

## Initialization of Structure

### Method 1: During declaration

c

struct Student s1 = {1, "Meet", 89.5};

### Method 2: After declaration

c

s1.roll = 1;

strcpy(s1.name, "Meet");

s1.marks = 89.5;

(Note: Use *strcpy()* from *<string.h>* for string assignment)

## Accessing Structure Members

Use the **dot operator .** for direct access:

c

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

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

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

## Complete Example

c

#include <stdio.h>

#include <string.h>

// Structure definition

struct Student {

int roll;

char name[50];

float marks;

};

int main() {

struct Student s1;

// Assign values

s1.roll = 101;

strcpy(s1.name, "Meet");

s1.marks = 87.5;

// Access and print values

printf("Student Info:\n");

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

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

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

return 0;

}

**13. File Handling in C**

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

In **C programming**, **file handling** is crucial for **storing data permanently** — beyond the life of a program's execution. Instead of using memory (RAM), which is **temporary**, files allow reading/writing data to **disk**, making it **persistent**.

| **Advantage** | **Explanation** |
| --- | --- |
| 🗂 **Data Storage** | Stores data permanently in files (e.g., text, logs, records) |
| 🔁 **Data Retrieval** | Reads previously saved data (e.g., user info, settings) |
| 💾 **Large Data Handling** | Easily manage data larger than memory can hold |
| 🔒 **Security** | Can restrict access using file permissions |
| Function | Purpose |
| fopen() | Opens a file |
| fclose() | Closes an opened file |
| fprintf() | Writes formatted data to a file |
| fscanf() | Reads formatted data from a file |
| fgets() | Reads string from file |
| fputs() | Writes string to file |
| fread() | Reads binary data |
| fwrite() | Writes binary data |

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

| **Mode** | **Meaning** |
| --- | --- |
| "r" | Open for reading |
| "w" | Open for writing (overwrites) |
| "a" | Open for appending |
| "r+" | Read + write |
| "w+" | Write + read (overwrite) |
| "a+" | Append + read |

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

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

fprintf(fp, "Hello World\n");

fputs("Another line\n", fp);

fclose(fp);

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

char str[100];

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

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

printf("%s", str);

fclose(fp);

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

**fclose(fp); // Always close after use**

**13. File Handling in C**

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

# What is File Handling in C?

In **C**, **file handling** allows you to **create**, **open**, **read**, **write**, and **manipulate files** stored on disk. Unlike variables (which are temporary), files allow you to **store data permanently**.

**Importance of File Handling in C**

| **Reason** | **Description** |
| --- | --- |
| **Persistent Storage** | Data saved in a file remains after the program ends. |
| **Large Data Handling** | Suitable for processing large volumes of data. |
| **Data Reuse** | Can read previously saved data for future use. |
| **Report Generation** | Allows exporting results or logs to external files. |
| **Useful in Projects** | Essential for applications like billing systems, games, databases, etc. |

**Basic Steps in File Handling**

1. **Open** the file using fopen()
2. **Read/Write** data using functions like fprintf(), fscanf(), fgets(), fputs()
3. **Close** the file using fclose()