Module 2 – Introduction to Programming

Overview of C Programming

Theory Exercise

C programming is a general-purpose, procedure-oriented programming language. It is both machine-independent and structured. C is a high-level programming language developed by Dennis Ritchie in the early 1970s. It is now one of the most popular and influential programming languages worldwide.

Lab Exercise :-Three real world application where C Programming is extensive used operating system like Linux, macOS, and Android rely on for core functionalities.

2. Setting Up Environment

3. Basic Structure of a C Program

• THEORY EXERCISE:

**1. Comments (Documentation)**

Comments describe what the program does. They’re ignored by the compiler and help make your code readable.

* **Single-line comment**:

// This program prints Hello, World!

* **Multi-line comment**:

/\*

Author: You

Purpose: Demonstrate basic C structure

\*/

**2. Header Files**

Headers provide access to standard functions and libraries.

* **Example**:

#include <stdio.h> // Standard input/output functions

* You can also include custom headers:

#include "myheader.h"

**3. Main Function**

Every C program starts execution from the main() function.

* **Syntax**:

int main() {

// Code goes here

return 0;

}

* int indicates the function returns an integer.
* return 0; signals successful execution.

**4. Data Types**

Data types define the kind of data a variable can hold.

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| int | Integer numbers | int age = 25; |
| float | Decimal numbers | float pi = 3.14; |
| char | Single characters | char grade = 'A'; |
| double | Double-precision decimals | double temp = 98.6; |

**5. Variables**

Variables store data that can be used and manipulated.

* **Declaration**:

int score;

* **Initialization**:

score = 100;

* **Combined**:

int score = 100;

4. Operators in C

• THEORY EXERCISE:

**Arithmetic Operators**

Used to perform basic mathematical operations.

| **Operator** | **Description** | **Example (a = 10, b = 3)** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division | a / b | 3 |
| % | Modulus (remainder) | a % b | 1 |

**Note**: % works only with integers.

**Relational Operators**

Used to compare two values and return a Boolean result (1 for true, 0 for false).

| **Operator** | **Meaning** | **Example (a = 10, b = 3)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | 0 |
| != | Not equal to | a != b | 1 |
| > | Greater than | a > b | 1 |
| < | Less than | a < b | 0 |
| >= | Greater than or equal to | a >= b | 1 |
| <= | Less than or equal to | a <= b | 0 |

**Logical Operators**

Used to combine multiple conditions.

| **Operator** | **Meaning** | **Example (a = 10, b = 3)** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | (a > 5 && b < 5) | 1 |
| ` |  | ` | Logical OR | `(a < 5 |  | b < 5)` | 1 |
| ! | Logical NOT | !(a > b) | 0 |  |  |  |  |

**Tip**: These are commonly used in if, while, and for statements.

**Assignment Operators**

Used to assign values to variables.

| **Operator** | **Meaning** | **Example (a = 10)** | **Result** |
| --- | --- | --- | --- |
| = | Assign | a = 5 | 5 |
| += | Add and assign | a += 2 | 12 |
| -= | Subtract and assign | a -= 3 | 7 |
| \*= | Multiply and assign | a \*= 2 | 20 |
| /= | Divide and assign | a /= 5 | 2 |
| %= | Modulus and assign | a %= 3 | 1 |

**Increment / Decrement Operators**

Used to increase or decrease a variable’s value by 1.

| **Operator** | **Type** | **Example (a = 10)** | **Result** |
| --- | --- | --- | --- |
| ++a | Pre-increment | ++a | 11 |
| a++ | Post-increment | a++ | 10 (then 11) |
| --a | Pre-decrement | --a | 9 |
| a-- | Post-decrement | a-- | 10 (then 9) |

**Pre** changes the value before use; **Post** changes it after.

**Bitwise Operators**

Operate at the binary level.

| **Operator** | **Meaning** | **Example (a = 5, b = 3)** | **Result** |
| --- | --- | --- | --- |
| & | AND | a & b | 1 |
| ` | ` | OR | `a | b` | 7 |
| ^ | XOR | a ^ b | 6 |  |  |
| ~ | NOT | ~a | -6 |  |  |
| << | Left shift | a << 1 | 10 |  |  |
| >> | Right shift | a >> 1 | 2 |  |  |

**Use case**: Efficient low-level programming and hardware control.

5. Control Flow Statements in C

• THEORY EXERCISE:

**1. if Statement**

Executes a block of code **only if** a condition is true.

c

int age = 20;

if (age >= 18) {

printf("You are eligible to vote.\n");

}

**Explanation**: If age is 18 or more, the message is printed.

**2. if-else Statement**

Provides an **alternative path** if the condition is false.

int age = 16;

if (age >= 18) {

printf("You are eligible to vote.\n");

} else {

printf("You are not eligible to vote.\n");

}

**Explanation**: If the condition fails, the else block runs.

**3. Nested if-else Statement**

Allows **multiple levels of conditions** inside one another.

int marks = 85;

if (marks >= 50) {

if (marks >= 75) {

printf("Distinction\n");

} else {

printf("Pass\n");

}

} else {

printf("Fail\n");

}

**Explanation**: First checks if the student passed, then checks for distinction.

**4. switch Statement**

Used when you have **multiple discrete values** to compare.

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");

}

**Explanation**: Matches day with a case and executes the corresponding block.

6. Looping in C

• THEORY EXERCISE:

**Overview Comparison**

| **Feature** | **while Loop** | **for Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| **Condition Check** | Before loop starts | Before loop starts | After first iteration |
| **Minimum Execution** | 0 times | 0 times | At least once |
| **Syntax** | while (condition) { ... } | for (init; condition; update) { ... } | do { ... } while (condition); |
| **Use Case** | Unknown iterations | Known number of iterations | Must run at least once |
| **Loop Control** | Manual init & update | Built-in init, condition, update | Manual init & update |

**1. while Loop**

**Best for**: When the number of iterations is **not known in advance**.

int i = 0;

while (i < 5) {

printf("%d\n", i);

i++;

**2. for Loop**

**Best for**: When the number of iterations is **known or count-controlled**.

for (int i = 0; i < 5; i++) {

printf("%d\n", i);

}

**3. do-while Loop**

**Best for**: When the loop must **run at least once**, regardless of the condition.

int i = 0;

do {

printf("%d\n", i);

i++;

} while (i < 5);

7. Loop Control Statements

• THEORY EXERCISE:

**Statement**

Used to **exit a loop or switch-case block** immediately.

**Use Cases:**

* Terminate a loop when a condition is met
* Exit a switch case after executing a block

**Example: Breaking a for loop**

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

if (i == 5) {

break; // Exit loop when i is 5

}

printf("%d ", i);

}

// Output: 1 2 3 4

**Example: break in a switch case**

int choice = 2;

switch (choice) {

case 1: printf("Option 1\n"); break;

case 2: printf("Option 2\n"); break;

default: printf("Invalid\n");

}

**continue Statement**

Used to **skip the current iteration** of a loop and move to the next one.

**Use Cases:**

* Skip unwanted values (e.g., even numbers)
* Filter input or conditions inside loops

**Example: Skipping even numbers**

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

if (i % 2 == 0) {

continue; // Skip even numbers

}

printf("%d ", i);

}

// Output: 1 3 5

**Example: continue in a while loop**

int i = 0;

while (i < 5) {

i++;

if (i == 3) continue;

printf("%d ", i);

}

// Output: 1 2 4 5

8. Functions in C

• THEORY EXERCISE:

**1. Function Declaration (Prototype)**

This tells the compiler about the function’s name, return type, and parameters **before it’s used**.

c

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

* int: return type
* add: function name

(int a, int b): parameters

**Note**: Declarations are optional if the function is defined before it’s called.

**2. Function Definition**

This is where you write the actual code that the function will execute.

c

int add(int a, int b) {

return a + b;

}

* The body is enclosed in {} and contains the logic.
* The return statement sends a value back to the caller.

**3. Function Call**

This is how you **invoke** the function to perform its task.

c

int result = add(5, 3);

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

* You pass arguments that match the parameters.
* The return value can be stored or used directly.

9. Arrays in C

• THEORY EXERCISE:

An **array** is a collection of elements (like int, float, char) stored in a fixed-size sequence. Instead of declaring multiple variables, you can use one array to store them all.

int scores[5]; // Declares an array of 5 integers

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

A **1D array** is a simple list of elements accessed using a single index.

**Declaration & Initialization:**

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

**Accessing Elements:**

printf("%d", marks[2]); // Outputs 78

**Use Case:**

* Storing student scores
* Tracking daily temperatures
* Managing inventory counts

**Multi-Dimensional Array**

A **multi-dimensional array** is essentially an array of arrays. The most common is the **two-dimensional array (2D)**, which resembles a table or matrix.

**Declaration & Initialization:**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

**Accessing Elements:**

printf("%d", matrix[1][2]); // Outputs 6

**Use Case:**

* Representing a chessboard or game grid
* Storing tabular data (rows × columns)
* Image pixel data (height × width)

10. Pointers in C

• THEORY EXERCISE:

A **pointer** is a variable that stores the **memory address** of another variable rather than the actual value.

int x = 10;

int \*ptr = &x; // ptr holds the address of x

* \*ptr is a pointer to an integer.
* &x is the address of variable x.

**Declaration of Pointers**

To declare a pointer, use the \* symbol before the variable name:

Data \_type \* pointer\_ name;

**Examples:**

int \*p ; // Pointer to an integer

float \*fptr ; // Pointer to a float

char \*cptr ; // Pointer to a character

11. Strings in C

• THEORY EXERCISE:

**1.**strlen()**: String Length**

* **Purpose**: Calculates the length of a string (excluding the null terminator \0).
* **Use Case**: Useful when you need to determine the size of a string for operations like memory allocation or iteration.

**2.**strcpy()**: String Copy**

* **Purpose**: Copies the content of one string into another.
* **Use Case**: Useful when you need to duplicate a string into a new variable.

**3..**strcat()**: String Concatenation**

* **Purpose**: Appends one string to the end of another.
* **Use Case**: Useful for combining strings, such as creating a full file path or a message.

**4.**strcmp()**: String Comparison**

* **Purpose**: Compares two strings lexicographically.
  + Returns 0 if strings are equal.
  + Returns a negative value if the first string is less than the second.
  + Returns a positive value if the first string is greater than the second.
* **Use Case**: Useful for sorting strings or checking equality.

**5.**strchr()**: Find Character in String**

* **Purpose**: Searches for the first occurrence of a character in a string.
* **Use Case**: Useful for locating specific characters, such as delimiters or markers.

12. Structures in C

• THEORY EXERCISE:

In C programming, **structures** are user-defined data types that allow grouping variables of different types under a single name. They are particularly useful for organizing complex data and creating records, such as representing a student with attributes like name, age, and grade.

13. File Handling in C

• THEORY EXERCISE:

File handling in C is a crucial aspect of programming as it allows programs to store data permanently, retrieve it when needed, and manage large datasets efficiently. Unlike variables, which store data temporarily in memory, files enable persistent storage on disk, making them essential for applications like databases, logging, and configuration management.

**Importance of File Handling in C:**

1. **Data Persistence**: Files allow data to be stored permanently, even after the program terminates.
2. **Large Data Management**: Files can handle large amounts of data that cannot fit into memory.
3. **Data Sharing**: Files enable data to be shared between different programs or systems.
4. **Backup and Recovery**: Files provide a way to back up important data and recover it when needed.
5. **Structured Storage**: Files can store data in structured formats (e.g., text, binary, CSV) for easy processing.

**File Operations in C:**

C provides a set of standard library functions for file handling, which are declared in the <stdio.h> header file. Below are the key operations:

**1. Opening a File**

* Files are opened using the f open() function, which returns a pointer to the file.
* Syntax: FILE \*f open (const char \*filename, const char \*mode);

**Example**:

Copy the code FILE \*file = f open("example.txt", "w");

if (file == NULL) {

print f("Error opening file!\n");

}

**2. Closing a File**

* Files must be closed using the f close() function to free resources and ensure data integrity.
* Syntax: int f close(FILE \*stream);

**Example**:

Copy the code f close (file);

**3. Reading from a File**

* Use functions like fread() to read data.
  + F read(): Reads binary data.

**Example**:

Copy the code FILE \*file = f open("example.txt", "r");

char buffer[100];

if (file != NULL) {

while ( f gets (buffer, size of(buffer), file) != NULL) {

print f("%s", buffer);

}

F close(file);

}

**4. Writing to a File**

* Use functions like f write() to write data.
  + F write(): Writes binary data.

**Example**:

Copy the codeFILE \*file = fopen("example.txt", "w");

if (file != NULL) {

fputs("Hello, World!\n", file);

fclose(file);

}

