Assignment – 2

Module 2 – Introduction to Programming

## Overview of C Programming

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

* C programming is one of the most powerful and widely used programming languages in computer science. It has had a lasting influence on many modern languages and continues to play a vital role in software development even today.
* In **1972**, Dennis Ritchie developed **C**, which combined the efficiency of low-level assembly with the structure and readability of high-level languages.
* C became the main language for rewriting the UNIX operating system, which greatly increased its popularity.
* By the **1980s**, C had spread widely, leading to the creation of the **ANSI C standard** (American National Standards Institute, 1989), which ensured that C programs could run on different machines without modification.
* **Importance of C Programming**
* Efficiency and Performance
* Portability
* Foundation of Other Languages
* Operating Systems and Compilers

Even after 50+ years, C remains highly relevant:

* **Embedded Systems**: Microcontrollers, IoT devices, and hardware programming often rely on C.
* **System Software**: Operating systems, databases, and device drivers are mostly written in C.
* **Performance-Critical Applications**: Fields like gaming, scientific computing, and high-frequency trading still use C for speed.
* **Learning Foundation**: Understanding C helps programmers learn memory management, data structures, and concepts that are essential for mastering other languages.

## Setting Up Environment

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

* **Step 1: Install a C Compiler (GCC)**
* GCC (GNU Compiler Collection) is one of the most common C compilers.
* Download **MinGW** (Minimalist GNU for Windows) from MinGW official site.
* Install it → during installation, select **mingw32-gcc-g++** package.
* Add the **bin folder path** (like C:\MinGW\bin) to the **Environment Variables → PATH**.
* Open **Command Prompt** and type: gcc --version
* **Step 2: Install an IDE**
* You can use **Dev C++**, **VS Code**, or **Code::Blocks** to write and run C programs easily.

## 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.

* **Basic Structure of a C Program**

A C program is made up of different parts that work together.

1. **Header Files**

* Start of the program.
* Contain definitions of functions (like printf, scanf).
* Written using #include.

1. **Main Function**

* Every C program starts execution from main ().
* Code inside {} runs when the program starts.
* Example:

int main () {

// code goes here

return 0;

}

1. **Comments**

* Notes for programmers, ignored by compiler.
* Two types:

Single-line → // comment

* + Multi-line → /\* comment \*/

1. **Data Types**

* Define the kind of data a variable can store.
* Examples:
  + int → integers (10, -5)
  + float → decimals (3.14, -2.5)
  + char → single characters (‘A’, ‘b’)

1. **Variables**

* Named storage for data.
* Must be declared with a data type.
* Example:

int age = 20; // integer variable

float pi = 3.14; // floating-point variable

char grade = 'A'; // character variable

## Operators in C

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

* **Operators in C**

1. Arithmetic Operators (+, -, *, /, %)*

* Used for mathematical operations.
* Example:

int a = 10, b = 3;

printf("%d\n", a + b); // 13

printf("%d\n", a - b); // 7

printf("%d\n", a \* b); // 30

printf("%d\n", a / b); // 3 (integer division)

printf("%d\n", a % b); // 1 (remainder)

1. Relational Operators (==, !=, >, <, >=, <=)

* Compare two values and return **true (1)** or **false (0)**.
* Example:

int x = 5, y = 10;

printf("%d\n", x == y); // 0 (false)

printf("%d\n", x != y); // 1 (true)

printf("%d\n", x < y); // 1 (true)

printf("%d\n", x >= y); // 0 (false)

1. Logical Operators (&&, ||,!)

* Combine or reverse conditions.
* Example:

int a = 1, b = 0;

printf("%d\n", a && b); // 0 (AND → both must be true)

printf("%d\n", a || b); // 1 (OR → at least one true)

printf("%d\n", !a); // 0 (NOT → reverses value)

1. Assignment Operators (=, +=, -=, *=, /=, %=)*

* Assign values to variables, sometimes with shortcuts.
* Example:

int n = 5;

n += 3; // same as n = n + 3 → n = 8

n -= 2; // n = 6

n \*= 4; // n = 24

n /= 6; // n = 4

1. Increment/Decrement Operators (++, --)

* Increase or decrease a value by 1.
* Example:

int a = 5;

printf("%d\n", a++); // 5 (post-increment: use then increase)

printf("%d\n", a); // 6

printf("%d\n", ++a); // 7 (pre-increment: increase then use)

1. Bitwise Operators (&, |, ^, ~, <<, >>)

* Work at the **bit level**.
* Example:

int a = 5, b = 3; // 5 = 0101, 3 = 0011 (binary)

printf("%d\n", a & b); // 1 (AND → 0001)

printf("%d\n", a | b); // 7 (OR → 0111)

printf("%d\n", a ^ b); // 6 (XOR → 0110)

printf("%d\n", ~a); // -6 (NOT → flips bits)

printf("%d\n", a << 1); // 10 (Left shift → 1010)

printf("%d\n", a >> 1); // 2 (Right shift → 0010)

1. Conditional (Ternary) Operator (?:)

* Shorthand for if-else.
* Syntax: condition ? value\_if\_true : value\_if\_false
* Example:

int age = 18;

char \*result = (age >= 18) ? "Adult" : "Minor";

printf("%s\n", result); // Adult

## Control Flow Statements in C

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

* **Decision-Making Statements in C**
* These statements let a program **choose different paths** based on conditions.

1. if Statement

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

#include <stdio.h>

int main() {

int age = 20;

if (age >= 18) {

printf("You are an adult.\n");

}

return 0;

}

1. if-else Statement

* Runs one block if condition is true, otherwise another block.
* Example:

#include <stdio.h>

int main() {

int num = 5;

if (num % 2 == 0) {

printf("Even number\n");

} else {

printf("Odd number\n");

}

return 0;

}

1. Nested if-else

* An **if-else inside another if-else** for multiple conditions.
* Example:

#include <stdio.h>

int main() {

int marks = 75;

if (marks >= 90) {

printf("Grade: A\n");

} else if (marks >= 75) {

printf("Grade: B\n");

} else if (marks >= 50) {

printf("Grade: C\n");

} else {

printf("Fail\n");

}

return 0;

}

1. switch Statement

* Used when we have **many choices** based on a variable’s value.
* Easier to read than multiple if-else.
* Example:

#include <stdio.h>

int main() {

int day = 3;

switch (day) {

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

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

case 3: printf("Wednesday\n"); break;

case 4: printf("Thursday\n"); break;

case 5: printf("Friday\n"); break;

case 6: printf("Saturday\n"); break;

case 7: printf("Sunday\n"); break;

default: printf("Invalid day\n");

}

return 0;

}

## Looping in C

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

* **Loops in C**
* Loops let us **repeat a block of code** multiple times until a condition is met.

1. **while loop**

* Condition is checked first → if true, loop runs.
* May run 0 or more times.
* **Example:**

#include <stdio.h>

int main() {

int i = 1;

while (i <= 5) {

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

i++;

}

return 0;

}

* Output: 1 2 3 4 5

1. **for loop**

* Used when we know exactly how many times to repeat.
* Compact → initialization, condition, and update in one line.
* **Example:**

#include <stdio.h>

int main() {

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

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

}

return 0;

}

* Output: 1 2 3 4 5

1. **do-while loop**

* Runs at least once, because condition is checked after executing the loop body.
* Example:

#include <stdio.h>

int main() {

int i = 1;

do {

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

i++;

} while (i <= 5);

return 0;

}

* Output: 1 2 3 4 5

## Loop Control Statements

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

1. **break Statement**

* Used to **exit** a loop or switch immediately, even if the condition is still true.
* **Example (exit loop when i == 3):**

#include <stdio.h>

int main() {

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

if (i == 3) {

break; // exits the loop when i = 3

}

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

}

return 0;

}

* Output: 1 2

1. **Continue Statement**

* Used to skip the current iteration of a loop and move to the next one.
* **Example (skip printing 3):**

#include <stdio.h>

int main() {

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

if (i == 3) {

continue; // skips when i = 3

}

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

}

return 0;

}

* **Output: 1 2 4 5**

1. **Goto Statement**

* Used to **jump** to a labelled part of the program.
* Not recommended (makes code hard to read), but sometimes used in error handling.
* **Example (jump to a label):**

#include <stdio.h>

int main() {

int i = 1;

start: // label

if (i <= 5) {

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

i++;

goto start; // jumps back to 'start'

}

return 0;

}

* Output: 1 2 3 4 5

## Functions in C

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

* **Functions in C**
* A **function** is a block of code that performs a specific task.
* Helps in **code reusability** (write once, use many times).
* Makes programs easier to **read, debug, and maintain**.

1. **Function Declaration (Prototype)**

* Tells the compiler about the function name, return type, and parameters (before using it).
* **Syntax:**

return\_type function\_name(parameter\_list);

* **Example:**

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

1. **Function Definition**

* Actual body of the function (what it does).
* Contains statements to perform the task.
* **Example:**

int add(int a, int b) { // Definition

return a + b;

}

1. **Function Call**

* Used to execute the function.
* Control goes to the function, executes it, and returns the result.
* **Example:**

#include <stdio.h>

// Function declaration

int add(int a, int b);

int main() {

int result = add(5, 3); // Function call

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

return 0;

// Function definition

int add(int a, int b) {

return a + b;

}

* Output: Sum = 8

## Arrays in C

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

* **Arrays in C**
* An **array** is a collection of elements of the **same data type** stored in **contiguous memory locations**.
* Each element can be accessed using an **index** (starting from 0).
* Helps when we need to store multiple values without creating many variables.
* Example:

#include <stdio.h>

int main() {

int numbers[5] = {10, 20, 30, 40, 50}; // Array of 5 integers

printf("First element = %d\n", numbers[0]);

printf("Third element = %d\n", numbers[2]);

return 0;

}

* Output: First element = 10

Third element = 30

1. **One-Dimensional Array**

* A simple list of elements.
* Syntax:

data\_type array\_name[size];

* Example:

#include <stdio.h>

int main() {

int marks[4] = {85, 90, 78, 92}; // 1D array

for (int i = 0; i < 4; i++) {

printf("marks[%d] = %d\n", i, marks[i]);

}

return 0;

}

* Output: marks[0] = 85 marks[1] = 90 marks[2] = 78 marks[3] = 92

1. **Multi-Dimensional Array**

* An array of arrays (like a **table or matrix**).
* Most common: **2D array** (rows & columns).
* Example:

#include <stdio.h>

int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

// Printing the matrix

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

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

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

}

printf("\n");

}

return 0;

}

* Output:

1 2 3

4 5 6

## Pointers in C

### 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 direct value (like 10), a pointer holds the **location** where that value is stored.

1. **Declaration**

data\_type \*pointer\_name;

* \* indicates it’s a pointer.
* datatype must match the variable it points to.
* Example:

int \*ptr; // pointer to int

float \*fptr; // pointer to float

1. **Initialization**

* You assign the address of a variable to a pointer using the address-of operator (&).
* Example:

#include <stdio.h>

int main() {

int x = 10;

int \*ptr = &x; // store address of x in ptr

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

printf("Address of x = %p\n", &x); // %p prints address

printf("Pointer ptr stores = %p\n", ptr);

printf("Value at address ptr points to = %d\n", \*ptr); // dereferencing

return 0;

}

* Output:

Value of x = 10

Address of x = 0x7ffee7b5c6ac

Pointer ptr stores = 0x7ffee7b5c6ac

Value at address ptr points to = 10

* Why are Pointers Important in C?
* Direct Memory Access
* Dynamic Memory Allocation
* Efficient Array Handling
* Function Arguments
* Data Structures

## Strings in C

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

* **String Handling Functions in C**
* All these functions are defined in the **<string.h> library**.

1. **strlen() – String Length**

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

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Hello";

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

return 0;}

* Output: Length = 5

1. **strcpy() – String Copy**

* Copies one string into another.
* Example:

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "C Programming";

char dest[20];

strcpy(dest, src);

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

return 0;

}

* Output: Copied String: C Programming

1. **strcat() – String Concatenation**

* Joins (appends) one string at the end of another.
* Example:

#include <stdio.h>

#include <string.h>

int main() {

char str1[20] = "Hello, ";

char str2[] = "World!";

strcat(str1, str2);

printf("Concatenated String: %s\n", str1);

return 0;

}

* Output: Concatenated String: Hello, World!

1. **strcmp() – String Compare**

* Compares two strings:
* Returns 0 if equal.
* Returns **positive** if first string > second string.
* Returns **negative** if first string < second string.
* Example:

#include <stdio.h>

#include <string.h>

int main() {

char s1[] = "apple";

char s2[] = "banana";

int result = strcmp(s1, s2);

if (result == 0) printf("Strings are equal\n");

else if (result < 0) printf("s1 is smaller\n");

else printf("s1 is greater\n");

return 0;

}

* Output: s1 is smaller

1. **strchr() – Find Character**

* Finds the first occurrence of a character in a string.
* **Example**:

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "programming";

char \*pos = strchr(str, 'g');

if (pos != NULL)

printf("Found 'g' at position: %ld\n", pos - str + 1);

else

printf("Character not found\n");

return 0;

}

* Output: Found 'g' at position: 4

## Structures in C

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

* **Structures in C**
* A **structure** in C is a **user-defined data type** that lets you group different types of variables under a single name.
* Useful when you want to represent a real-world entity with multiple properties.

1. **Declaring a Structure**

* **Syntax**:

struct structure\_name {

data\_type member1;

data\_type member2;

...

};

* **Example:**

struct Student {

int roll\_no;

char name[50];

float marks;

};

1. **Initializing a Structure**

* There are different ways:
  + **Direct Initialization**
    - struct Student s1 = {101, "John", 85.5};
  + **Assigning Values Later**
    - struct Student s2;

s2.roll\_no = 102;

strcpy(s2.name, "Alice"); // use strcpy for strings

s2.marks = 90.0;

1. **Accessing Structure Members**

* Use the **dot operator (.)** with the structure variable.
* Example:

#include <stdio.h>

#include <string.h>

struct Student {

int roll\_no;

char name[50];

float marks;

};

int main() {

// Initialize directly

struct Student s1 = {101, "John", 85.5};

// Initialize step by step

struct Student s2;

s2.roll\_no = 102;

strcpy(s2.name, "Alice");

s2.marks = 90.0;

// Accessing members

printf("Student 1: %d, %s, %.2f\n", s1.roll\_no, s1.name, s1.marks);

printf("Student 2: %d, %s, %.2f\n", s2.roll\_no, s2.name, s2.marks);

return 0;

}

* Output: Student 1: 101, John, 85.50

Student 2: 102, Alice, 90.00

## 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.

* **Importance of File Handling in C**
* Normally, variables and arrays store data temporarily in memory (RAM).
* Once the program ends, that data is lost.
* File handling allows us to store, read, and modify data permanently on disk.
* **Examples**: saving student records, logs, configuration files, databases.
* **File Operations in C**
  + All file handling functions are in the **<stdio.h>** library.
  + Files in C are handled using a **FILE pointer**.
  1. **Opening a File**
* We use fopen () with modes:
* "r" → read
* "w" → write (creates new or overwrites existing file)
* "a" → append (add at end)
* "r+", "w+", "a+" → read/write
* Example:

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

if (fp == NULL) {

printf("File not found!\n");}

* 1. **Writing to a File**
* fprintf() → formatted writing (like printf for files)
* fputs() → write a string
* fputc() → write a character
* Example:

#include <stdio.h>

int main() {

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

if (fp == NULL) {

printf("Error opening file!\n");

return 1;

}

fprintf(fp, "Hello, File Handling in C!\n");

fputs("This is another line.\n", fp);

fclose(fp);

return 0;

}

* 1. **Reading from a File**
* fscanf() → formatted reading
* fgets() → read a line
* fgetc() → read a character
* Example:

#include <stdio.h>

int main() {

char line[100];

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

if (fp == NULL) {

printf("Error opening file!\n");

return 1;

}

while (fgets(line, sizeof(line), fp)) {

printf("%s", line); // print file contents

}

fclose(fp);

return 0;}

* 1. **Closing a File**
* Always close files using fclose(fp);
* This ensures data is saved properly and frees system resources.