**Module 2 – Introduction to Programming**

**Overview of C Programming**

**1>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 influential programming languages in the history of computer science. Known for its efficiency, simplicity, and flexibility, C laid the foundation for many modern languages like C++, Java, and Python. Despite being developed over five decades ago, C remains widely used today due to its performance and close relationship with hardware.

The history of C dates back to the early 1970s at Bell Laboratories (Bell Labs), where Dennis Ritchie and Ken Thompson were working on developing a new operating system called UNIX.

* Before C, there were languages like Assembly, B, and BCPL (Basic Combined Programming Language).
* B, created by Ken Thompson, was a predecessor of C but lacked certain features like data types and structures.

C was officially developed in 1972 by Dennis Ritchie, improving upon B by adding data types, structures, and better control constructs.

C’s design influenced many later programming languages:

* **C++** – Added object-oriented programming to C.
* **Java, C#, Objective-C** – Derived their syntax and concepts from C.
* **Python, JavaScript, and Go** – Borrowed structural and procedural features from C.
* Even operating systems like **Windows, Linux, macOS**, and **embedded systems** rely on C-based code.

**2. Setting Up Environment**

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

1. Go to the official MinGW website: https://www.mingw-w64.org/
2. Download the latest MinGW-W64 version.
3. Run the installer and choose:
   * **Architecture**: x86\_64
   * **Threads**: posix
   * **Exception**: seh
   * **Installation Path**: C:\mingw64
4. After installation, add GCC to the **System PATH**:
   * Go to **Control Panel → System → Advanced System Settings → Environment Variables**.
   * Under **Path**, click **Edit** → **New**, then add:

**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.

#include <stdio.h> // 1. Header files

// 2. Comments: Explain what the program does

int main() { // 3. Main function (entry point)

| Header File | Purpose |
| --- | --- |
| <stdio.h> | For input and output functions like printf, scanf. |
| <stdlib.h> | For memory allocation, conversions, and system commands. |
| <math.h> | For mathematical functions like sqrt(), pow(). |

// 4. Variable declaration

int number = 10;

float price = 99.50;

// 5. Logic and output

printf("Number: %d\n", number);

printf("Price: %.2f\n", price);

return 0; // 6. Return statement

}

1. **Operators in C**

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

| **Operator** | **Description** | | | | **Example** | | | **Result** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| + | Addition | | | | 5 + 3 | | | 8 | |
| - | Subtraction | | | | 5 - 3 | | | 2 | |
| \* | Multiplication | | | | 5 \* 3 | | | 15 | |
| / | Division | | | | 10 / 2 | | | 5 | |
| % | Modulus (remainder) | | | | 10 % 3 | | | 1 | |
| == | 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 to | | | | 5 >= 5 | | | 1 | |
| <= | Less than or equal to | | | | 3 <= 5 | | | 1 | |
| Operator | Description | | | | Example | | | Equivalent To | |
| = | Assign value | | | | x = 5; | | | x = 5; | |
| += | Add and assign | | | | x += 5; | | | x = x + 5; | |
| -= | Subtract and assign | | | | x -= 5; | | | x = x - 5; | |
| \*= | Multiply and assign | | | | x \*= 5; | | | x = x \* 5; | |
| /= | Divide and assign | | | | x /= 5; | | | x = x / 5; | |
| %= | Modulus and assign | | | | x %= 5; | | | x = x % 5; | |
| Operator | Description | | | | Example | | | Binary Result | |
| & | AND | | | | 5 & 3 | | | 101 & 011 = 001 (1) | |
| ` | ` | | | | OR | | | `5 | |
| ^ | XOR | | | | 5 ^ 3 | | | 101 ^ 011 = 110 (6) | |
| ~ | NOT (One's complement) | | | | ~5 | | | Flips bits | |
| << | Left Shift | | | | 5 << 1 | | | 101 << 1 = 1010 (10) | |
| >> | Right Shift | | | | 5 >> 1 | | | 101 >> 1 = 10 (2) | |
| **Operator Type** | | **Key Operators** | | | | **Main Use** | | |
| Arithmetic | | + - \* / % | | | | Basic calculations | | |
| Relational | | == != > < >= <= | | | | Comparisons | | |
| Logical | | `&& | | | |  | | |
| Assignment | | = += -= \*= /= %= | | | | Assigning values | | |
| Increment/Decrement | | ++ -- | | | | Increase/Decrease by 1 | | |
| Bitwise | | `& | | | | ^ ~ << >>` | | |
| Conditional | | ?: | | | | Short conditional checks | | |
| **Operator** | **Description** | | **Example** | **Result** | | |
| ++ | Increment by 1 | | x++ | x = x + 1 | | |
| -- | Decrement by 1 | | x-- | x = x - 1 | | |

1. **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**

Decision-making statements in C allow a program to **choose different actions** based on certain conditions. The main statements are:

* if
* if-else
* nested if-else
* switch

## **1.** if **Statement**

The if statement is used to **execute a block of code only if a condition is true**.

**Syntax:**

if (condition) {

// Code to execute if condition is true

}

**Example:**

#include <stdio.h>

int main() {

int age = 20;

if (age >= 18) {

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

}

return 0;

}

**Output:**

You are eligible to vote.

## **2.** if-else **Statement**

The if-else statement runs one block of code **if the condition is true**, and another block **if the condition is false**.

**Syntax:**

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

**Example:**

#include <stdio.h>

int main() {

int marks = 40;

if (marks >= 50) {

printf("Pass\n");

} else {

printf("Fail\n");

}

return 0;

}

**Output:**

Fail

## **3. Nested** if-else **Statement**

A **nested** if-else is used when there are **multiple conditions to check**. One if or else contains another if-else.

**Syntax:**

if (condition1) {

// Code for condition1

} else if (condition2) {

// Code for condition2

} else {

// Code if none of the conditions are true

}

**Example:**

#include <stdio.h>

int main() {

int marks = 85;

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("Grade: Fail\n");

}

return 0;

}

**Output:**

Grade: B

## **4.** switch **Statement**

The switch statement is used to **execute one block of code out of many options**, based on the value of a variable or expression. It is cleaner than multiple if-else statements.

**Syntax:**

switch (expression) {

case value1:

// Code for case 1

break;

case value2:

// Code for case 2

break;

default:

// Code if no case matches

}

**Example:**

#include <stdio.h>

int main() {

int choice = 2;

switch (choice) {

case 1:

printf("Option 1 selected\n");

break;

case 2:

printf("Option 2 selected\n");

break;

case 3:

printf("Option 3 selected\n");

break;

default:

printf("Invalid choice\n");

}

return 0;

}

**Output:**

Option 2 selected

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.

| Feature | while | for | do-while |
| --- | --- | --- | --- |
| Condition Check | Before the loop | Before the loop | After the loop |
| Minimum Executions | 0 times | 0 times | At least once |
| Best for | Unknown number of iterations | Known number of iterations | Guaranteed single execution |
| Syntax Length | Simple but needs manual update of counter | Compact and structured | Slightly longer |
| Common Use Cases | Reading input until EOF | Counting loops, array traversals | Menu-driven programs |

#include <stdio.h>

int main() {

int i = 1;

while (i <= 5) {

printf("%d ", i);

i++;

}

return 0;

}

1. **Loop Control Statements**

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

**Statement**

* The break statement is used to **immediately exit** a loop (for, while, do-while) or a switch statement.
* Control moves to the **first statement after the loop or switch**.

#include <stdio.h>

int main() {

int i;

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

if (i == 5) {

printf("Breaking at i = %d\n", i);

break; // exits the loop when i is 5

}

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

}

return 0;

}

## continue **Statement**

### **Use**:

* The continue statement **skips the current iteration** of the loop and moves control to the next iteration.
* Commonly used to **skip specific cases** without breaking out of the loop

#include <stdio.h>

int main() {

int i;

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

if (i == 5) {

continue; // skips printing when i is 5

}

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

}

return 0;

}

## goto **Statement**

### **Use**:

* The goto statement transfers control to a labeled statement **within the same function**.
* It is rarely recommended because it can make code **hard to read and debug** but can be useful in **error handling** or exiting from deeply nested loops.

#include <stdio.h>

int main() {

int i = 1;

start: // label

if (i <= 5) {

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

i++;

goto start; // jumps back to the label

}

return 0;

}

1. **Functions in C**

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

## **Structure of a Function**

A function in C generally has **three main parts**:

1. **Declaration (Prototype)**
2. **Definition (Body)**
3. **Calling the Function**

### **1. Function Declaration (Prototype)**

* Declares the function **before it is used** in the program.
* Tells the compiler the **name, return type, and parameters** of the function.

**Syntax**:

### **2. Function Definition (Body)**

* Contains the **actual code** that performs the task.

**Syntax**:

### **3. Function Call**

* Executes the function by **passing arguments** to it.

**Syntax**:

#include <stdio.h>

// Function Declaration

int add(int, int);

int main() {

int num1, num2, sum;

printf("Enter two numbers: ");

scanf("%d %d", &num1, &num2);

// Function Call

sum = add(num1, num2);

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

return 0;

}

// Function Definition

int add(int a, int b) {

return a + b;

}

1. **Arrays in C**

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

An **array** in C is a **collection of elements of the same data type** stored in **contiguous memory locations**.  
Arrays make it easier to store and manage multiple values under a single name, using an **index** to access each element.

### **Key Features of Arrays**

* Stores **multiple values of the same data type**.
* Uses **indexing** to access elements (index starts from 0).
* Size of the array must be **defined at declaration** and **cannot change dynamically**.

### **General Syntax**

data\_type array\_name[size];

Example:

int numbers[5]; // Array of 5 integers

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

A **one-dimensional array** is like a **list** that stores data in a **single row**.

### **Declaration**

data\_type array\_name[size];

### **Example**

#include <stdio.h>

int main() {

int marks[5] = {80, 75, 90, 85, 88}; // Declaration and initialization

int i;

printf("Marks in 5 subjects:\n");

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

printf("Subject %d: %d\n", i+1, marks[i]);

}

return 0;

}

## **2. Multi-Dimensional Array**

A **multi-dimensional array** stores data in a **table-like (matrix) form**.  
The most common is the **two-dimensional (2D) array**, which is like rows and columns.

### **Declaration**

#include <stdio.h>

int main() {

int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2 rows, 3 columns

int i, j;

printf("Matrix elements:\n");

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

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

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

}

printf("\n");

}

return 0;

}

1. **Pointers in C**

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

### **Pointers in C**

A **pointer** in C is a **variable that stores the memory address** of another variable.  
Instead of holding data directly, a pointer holds the **location** where the data is stored.

### **1. Declaration of Pointers**

A pointer is declared using the \* symbol before the pointer name.

**Syntax**:

int \*ptr; // Pointer to an integer

char \*cptr; // Pointer to a character

float \*fptr; // Pointer to a float

### **2. Initialization of Pointers**

A pointer is initialized using the **address-of operator (**&**)**.

**Syntax**:

#include <stdio.h>

int main() {

int num = 10;

int \*ptr; // Pointer declaration

ptr = &num; // Pointer initialization

printf("Value of num: %d\n", num);

printf("Address of num: %p\n", &num);

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

printf("Value pointed by ptr: %d\n", \*ptr); // Dereferencing

return 0;

}

### **3. Accessing Values Using Pointers**

* \* (dereference operator) is used to **access the value** stored at the address the pointer points to.

**Example**:

int x = 25;

int \*ptr = &x;

printf("%d", \*ptr); // Outputs 25

### **4. Importance of Pointers in C**

| **Reason** | **Explanation** |
| --- | --- |
| **Direct Memory Access** | Pointers let you access and modify memory directly, which increases efficiency. |
| **Dynamic Memory Allocation** | Functions like malloc(), calloc(), and free() rely on pointers. |
| **Function Arguments (Call by Reference)** | Allows passing the address of variables to modify data directly in functions. |
| **Efficient Array Handling** | Arrays and pointers are closely related, enabling fast traversal and manipulation. |
| **Data Structures** | Pointers are essential for linked lists, trees, graphs, and other advanced structures. |

### **5. Example: Using Pointer with Function**

#include <stdio.h>

void updateValue(int \*p) {

\*p = \*p + 10; // Modify value at the given address

}

int main() {

int num = 20;

printf("Before: %d\n", num);

updateValue(&num); // Passing address

printf("After: %d\n", num);

return 0;

}

**Output**:

Before: 20

After: 30

**11. Strings in C**

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

### strlen()

**Purpose:** Finds the **length** of a string (number of characters before '\0', not counting

#include <stdio.h>

#include <string.h>

int main() {

char name[] = "OpenAI";

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

return 0;

}

### strcpy()

**Purpose:** Copies a string from source to destination (including '\0').  
**Syntax:**

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "Hello";

char dest[20];

strcpy(dest, src);

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

return 0;

}

### **.** strcat()

**Purpose:** Concatenates (appends) one string to the end of another.  
**Syntax:**

#include <stdio.h>

#include <string.h>

int main() {

char str1[30] = "Good ";

char str2[] = "Morning";

strcat(str1, str2);

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

return 0;

}

1. **Structures in C**

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

Concept of Structures

* A structure is like a container that holds variables of different data types.
* Example: A "student" can have a name (string), roll number (int), and marks (float).
* Syntax uses the keyword struct.

🔹 Declaring a Structur

struct Student {

int roll\_no;

char name[50];

float marks;

};

🔹 Defining Structure Variables

There are two ways:

// Method 1: Separate declaration

struct Student s1, s2;

// Method 2: Declare along with structure

struct Student {

int roll\_no;

char name[50];

float marks;

} s3, s4;

Initializing Structure Members

You can initialize while declaring:

struct Student s1 = {101, "Alice", 89.5};

Or assign later:

s1.roll\_no = 101;

strcpy(s1.name, "Alice"); // For strings, use strcpy() from <string.h>

s1.marks = 89.5;

🔹 Accessing Structure Members

Use the dot operator (.):

printf("Roll No: %d\n", s1.roll\_no);

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

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

**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.

mportance of File Handling in C

* In C, when a program ends, all the data stored in variables is lost (stored in RAM).
* If you want data to persist (e.g., save user records, logs, or reports), you must store it in files on disk.
* File handling provides a way to:
  + Store data permanently
  + Retrieve and update data whenever needed
  + Manage large amounts of data easily

Without file handling, programs would only work with temporary data.

File Handling Functions in C

The <stdio.h> library provides functions for file handling. The most important ones are:

1. fopen() → Open/create a file
2. fclose() → Close a file
3. fgetc() / fgets() → Read from a file
4. fputc() / fputs() → Write to a file
5. fprintf() / fscanf() → Read/Write formatted data
6. fread() / fwrite() → Binary file handling

Modes in fopen():

* "r" → read (file must exist)
* "w" → write (creates new / overwrites file)
* "a" → append (adds to existing file)
* "r+" → read & write
* "w+" → read & write (overwrite)
* "a+" → read & append