Module 2 -Theory

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.

- The C programming language is one of the most influential languages in the history of computer science. It was developed in 1972 by Dennis Ritchie at Bell Labs as an improvement over the B language, which itself was derived from BCPL. C was initially created to develop the UNIX operating system, and its success led to the widespread adoption of C in system programming.

  C brought many new features that made it both powerful and flexible. It provided low-level memory access through pointers, a rich set of built-in operators, and structured programming features like loops and conditionals. This made C ideal for writing operating systems, compilers, and other performance-sensitive applications.

  In the 1980s, C became standardized by ANSI (American National Standards Institute), resulting in the ANSI C standard, also known as C89. Later, further improvements led to newer versions such as C99 and C11, which introduced additional features while maintaining the core structure of the language.

  C is important because it forms the foundation of many other programming languages, including C++, Java, and even modern scripting languages like Python and JavaScript, which borrow syntax and concepts from C. Its performance and control over hardware make it the language of choice for system software, embedded systems, and resource-constrained environments.

Even today, C remains highly relevant due to its efficiency, portability, and ability to interact directly with hardware. It is widely used in the development of operating systems, embedded devices, real-time systems, and game engines. C's long-lasting impact and widespread use in both academic and industrial settings highlight its enduring importance in the world of programming.

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.

Install GCC Compiler (Windows using MinGW)

- Download MinGW from the official website (https://osdn.net/projects/mingw/).
- Run the installer and select mingw32-gcc-g++, mingw32-base, and msys-base packages.
- After installation, add the path C:\MinGW\bin to the system **Environment Variables** under Path.
- Open Command Prompt and type gcc --version to check if GCC is installed successfully.

Set Up DevC++

- Download DevC++ from a trusted source (e.g., Bloodshed or SourceForge).
- Install the setup file and launch the IDE.
- DevC++ comes with an in-built compiler, so no separate installation is required.
- Go to Tools > Compiler Options to check or change compiler settings.
- Create a new project or source file and start writing C code.

Set Up VS Code for C Programming

- Download and install **Visual Studio Code** from https://code.visualstudio.com/.
- Install the **C/C++ extension** from Microsoft in the Extensions panel.
- Install **MinGW** as described earlier for the compiler.
- Open VS Code, create a new .c file, and set up tasks.json and launch.json files in the .vscode folder to compile and run code.
- Use Ctrl + Shift + B to build and F5 to run the program.

Set Up Code::Blocks

- Download Code::Blocks with the **MinGW setup** from https://www.codeblocks.org/downloads/.
- Install using the setup file and choose the version that includes the GCC compiler.
- After installation, go to Settings > Compiler to verify the compiler is set correctly.
- Create a new project, select Console Application, choose C language, and start coding.

---

3. 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 follows a specific structure that includes several key components such as header files, the main function, comments, data types, and variables. Here's an explanation with examples:

1. **Header Files**
   - Header files contain predefined functions and libraries.
   - Commonly used: `#include <stdio.h>` for input/output functions like `printf()` and `scanf()`.

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

2. **Main Function**
   - Every C program starts execution from the `main()` function.
   - It is mandatory and serves as the entry point of the program.

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

3. **Comments**
   - Comments are used to explain the code and are ignored by the compiler.
   - Single-line comment: `//`
   - Multi-line comment: `/* ... */`

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

4. **Data Types**
   - Define the type of data a variable can hold.
   - Common types:
     - `int` for integers
     - `float` for decimal numbers
     - `char` for characters
     - `double` for large floating-point numbers

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

5. **Variables**
    - Variables are named storage locations to hold data.
    - Must be declared with a data type before use.

```
int number; // declaration
number = 10; // assignment
```

Complete Example:

```
#include <stdio.h>  // Header file

int main() {  // Main function
   // Variable declarations
   int age = 25;
   float height = 5.9;
   char grade = 'A';

   // Output using printf
   printf("Age: %d\n", age);
   printf("Height: %.1f\n", height);
   printf("Grade: %c\n", grade);

   return 0;  // Exit program
}
```

---

4. Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

```
#include <stdio.h>  // Standard input-output header

int main() {
   // Variable declarations
   int age = 21;           // Integer variable
   char initial = 'D';     // Character variable
   float percentage = 85.6;  // Float variable

   // Constant declaration
   const float PI = 3.1415;  // Constant float value

   // Displaying variable values using printf
   printf("Student Details:\n");
   printf("Age: %d\n", age);              // Display integer
   printf("Initial: %c\n", initial);      // Display character
   printf("Percentage: %.2f%%\n", percentage); // Display float with 2 decimals
```

```
    printf("Value of PI (constant): %.4f\n", PI); // Display constant float

    return 0;  // End of program
}
```

---

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

Decision-Making Statements in C

Decision-making statements are used to control the flow of execution based on conditions. The main types are if, if-else, nested if-else, and switch.

1. **if Statement**
    - Executes a block of code if the condition is true.

```
int num = 10;
if (num > 0) {
    printf("Positive number\n");
}
```

2. **if-else Statement**
    - Executes one block if the condition is true, another if it is false.

```
int num = -5;
if (num >= 0) {
    printf("Non-negative\n");
} else {
    printf("Negative number\n");
}
```

3. **Nested if-else Statement**
    - An if or else block contains another if-else inside it.

```
int marks = 75;
if (marks >= 60) {
    if (marks >= 90) {
        printf("Excellent\n");
    } else {
        printf("Good\n");
    }
} else {
    printf("Needs Improvement\n");
}
```

4. **switch Statement**
    - Used to choose one block to execute from multiple options.

```
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 option\n");
}
```

---

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

1. while Loop

- **Checks the condition first**, then executes the block.
- Repeats as long as the condition is true.
- Might **not run at all** if the condition is false initially.

Syntax:

```
while (condition) {
   // Code to execute
}
```

Example:

```
int i = 1;
while (i <= 3) {
   printf("%d ", i);
   i++;
}
```

2. for Loop

- Compact form of loop.
- Best when **number of iterations is known**.
- Initialization, condition, and update are all in one line.

Syntax:

```
for (initialization; condition; increment) {
   // Code to execute
}
```

Example:

```
for (int i = 1; i <= 3; i++) {
   printf("%d ", i);
}
```

3. do-while Loop

- Executes the block **first**, then checks the condition.
- Ensures the loop runs **at least once**, even if condition is false.

Syntax:

```
do {
   // Code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
   printf("%d ", i);
   i++;
} while (i <= 3);
```

---

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

Use of break, continue, and goto in C

1. **break statement**
   Used to exit a loop or switch statement early.
   When break is executed, the control comes out of the loop or switch.

Example:

```
int i;
for (i = 1; i <= 5; i++) {
   if (i == 3) {
      break;
   }
   printf("%d ", i);
}
// Output: 1 2
```

2. **continue statement**
   Used to skip the current iteration of a loop and go to the next one.
   The loop continues, but skips the current step.

Example:

```
int i;
for (i = 1; i <= 5; i++) {
   if (i == 3) {
      continue;
   }
   printf("%d ", i);
}
// Output: 1 2 4 5
```

3. **goto statement**
   Used to jump to a specific label in the program.
   Not recommended for regular use as it can make the program hard to follow.

Example:

```
int num = 2;
if (num == 2) {
   goto label;
}
printf("This line is skipped.\n");

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

---

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

What is a Function?

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

- Reusing code
- Making programs modular and readable
- Breaking large programs into smaller tasks

- ### 3 Main Parts of a Function

Part	Description

Part	Description
Declaration	Tells the compiler about the function's name, return type, and parameters.
Definition	Contains the actual code (logic) of the function.
Calling	Tells the program to execute the function.

Syntax of a Function

```
return_type function_name(parameter_list);
```

Example:

```
// Declaration
void greet();

// Definition
void greet() {
   printf("Hello, welcome!\n");
}

// Calling the function
int main() {
   greet();  // Function call
   return 0;
}
```

Example with Parameters and Return Value

```
#include <stdio.h>

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

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

// Function Call
int main() {
   int result = add(10, 20);
   printf("Sum: %d\n", result);
   return 0;
}
```

---

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

Aspect	One-Dimensional Array	Multi-Dimensional Array
Definition	A collection of elements stored in a **single row/line**	A collection of elements stored in **rows and columns**
Structure	Linear (list-like)	Tabular (matrix-like)
Syntax	data_type array_name[size];	data_type array_name[rows][columns];
Example Declaration	int numbers[5];	int matrix[2][3];
Initialization	int numbers[5] = {1, 2, 3, 4, 5};	int matrix[2][3] = {{1,2,3}, {4,5,6}};
Access El ement	numbers[0], numbers[1]	matrix[0][1], matrix[1][2]
Usage Scenario	Storing student marks, ages, prices	Storing data in rows/columns like a table or grid
Example Code	int a[3] = {10, 20, 30}; printf("%d", a[1]);	int m[2][2] = {{1,2},{3,4}}; printf("%d", m[1][0]);

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

- ## What are Pointers in C?

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

In C, variables are stored in memory, and each variable has a specific **memory location (address)**. Pointers allow direct access to these memory addresses, enabling powerful and flexible programming techniques.

- ## Declaration and Initialization of Pointers
    - ***Syntax for Declaring a Pointer:***

data_type *pointer_name

Example:

int *ptr; // Declares a pointer to an int

- ***Initialization of Pointer:***

A pointer is usually initialized by assigning it the address of a variable using the **address-of operator &**.

```
int x = 10;
int *ptr = &x;  // ptr now holds the address of x
```

- **Accessing Value Using Pointer (Dereferencing)**

Use the **dereference operator** * to access the value stored at the address the pointer is pointing to.

```
printf("Value of x = %d\n", *ptr);  // prints 10
```

- **Complete Example:**

```
#include <stdio.h>

int main() {
   int x = 25;
   int *ptr = &x;

   printf("Address of x: %p\n", &x);
   printf("Address stored in ptr: %p\n", ptr);
   printf("Value of x using *ptr: %d\n", *ptr);

   return 0;
}
```

---

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

In C, strings are arrays of characters ending with a **null character** '\0'. The C Standard Library provides several functions to manipulate strings, which are declared in the **<string.h>** header file.

Here are 5 commonly used string handling functions:

1. strlen() – String Length

Purpose:
Returns the **length of a string**, excluding the null character ('\0').

Syntax:

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

Example:

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

int main() {
   char name[] = "Darshana";
   printf("Length = %lu\n", strlen(name));  // Output: 8
   return 0;
}
```

Use Case: Useful when you want to know how many characters a string contains.

---

2. strcpy() – String Copy

Purpose:
Copies one string into another.

Syntax:

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

Example:

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

int main() {
   char src[] = "Hello";
   char dest[20];
   strcpy(dest, src);
   printf("Copied String: %s\n", dest);  // Output: Hello
   return 0;
}
```

Use Case: When you need to duplicate or initialize a string.

---

3. strcat() – String Concatenation

Purpose:
Appends one string to the end of another.

Syntax:

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

Example:

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

int main() {
   char greeting[50] = "Hello, ";
   char name[] = "Darshana";
   strcat(greeting, name);
   printf("Full Greeting: %s\n", greeting);  // Output: Hello, Darshana
   return 0;
}
```

Use Case: Useful when combining multiple strings into one.

---

4. `strcmp()` – String Comparison

Purpose:
Compares two strings.

Syntax:

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

- Returns `0` if strings are equal
- Returns `< 0` if `str1 < str2`
- Returns `> 0` if `str1 > str2`

Example:

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

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

   if (strcmp(a, b) == 0)
      printf("Strings are equal\n");
   else
      printf("Strings are different\n");  // Output here

   return 0;
}
```

Use Case: Often used for checking user input or sorting strings.

5. strchr() – Find Character in String

Purpose:
Finds 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 word[] = "Programming";
   char *ptr = strchr(word, 'g');

   if (ptr)
      printf("Character found at position: %ld\n", ptr - word);
   else
      printf("Character not found\n");

   return 0;
}
```

---

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

- **Structures in C**

Structures in C are used to group different types of variables under one name. It allows storing a collection of variables of different data types together.

1. Declaring a structure:
You use the struct keyword to declare a structure.

Example:

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

2. Declaring structure variables:
You can create structure variables while or after defining the structure.

Example:
```
struct Student s1;
or
struct Student {
    int roll;
    char name[50];
    float marks;
} s1;
```

3. Initializing a structure:
You can assign values at the time of declaration.

Example:
```
struct Student s1 = {1, "Ravi", 85.5};
```

4. Accessing structure members:
Use the dot (.) operator to access members of the structure.

Example:
```
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.

What is File Handling?

File handling in C allows a program to **store data permanently** on storage devices (like hard disks) by **creating, reading, writing, and modifying** files. It enables data to be saved beyond the execution of the program.

Why is File Handling Important?

Importance	Explanation
Permanent Storage	Variables store data temporarily, but files store data permanently.

Importance	Explanation
Large Data Management	Handles large volumes of data more efficiently.
Data Sharing	Enables saving logs, reports, and user data for future use.
Input/Output Flexibility	Reads input from files and writes output to files instead of console.

File Operations in C

C provides several functions for file handling through the **<stdio.h>** header.

Operation	Function
Open a file	fopen()
Close a file	fclose()
Read from a file	fscanf(), fgets(), fgetc()
Write to a file	fprintf(), fputs(), fputc()

File Pointer

All file operations use a **file pointer** of type FILE * which points to the file being opened.

---

1. Opening a File – fopen()

Syntax:

```
FILE *fp;
fp = fopen("filename.txt", "mode");
```

Mode	Meaning
"r"	Read
"w"	Write (overwrite)
"a"	Append
"r+"	Read & Write
"+"	Write & Read
"a"	Append & Read

---

2. Writing to a File – fprintf(), fputs()

```
FILE *fp = fopen("data.txt", "w");
fprintf(fp, "Hello, World!\n");
fputs("Welcome to File Handling", fp);
fclose(fp);
```

---

3. Reading from a File – fscanf(), fgets(), fgetc()

```
FILE *fp = fopen("data.txt", "r");
char line[100];
fgets(line, 100, fp);
```

```
printf("Read Line: %s\n", line);
fclose(fp);
```

---

4. Closing a File – `fclose()`

Always close the file after operations to free system resources.

```
fclose(fp);
```

Full Example: Writing and Reading a File

```
#include <stdio.h>

int main() {
   FILE *fp;

   // Writing to file
   fp = fopen("student.txt", "w");
   fprintf(fp, "Name: Darshana\n");
   fprintf(fp, "Marks: 85.5\n");
   fclose(fp);

   // Reading from file
   char line[100];
   fp = fopen("student.txt", "r");
   while (fgets(line, sizeof(line), fp)) {
      printf("%s", line);
   }
   fclose(fp);

   return 0;
}
```
