Module 2 – Introduction to Programming

Overview of C Programming

THEORY EXERCISE:• o Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

Ans. The C programming language, conceived in the early 1970s, remains one of the most influential and enduring programming languages. Its simplicity, versatility, and efficiency have shaped the development of modern software and programming paradigms. This essay traces the origins and evolution of C, explores its significant contributions, and explains why it continues to play a vital role in programming today.

**Origins of C**

C was created by Dennis Ritchie at Bell Labs in 1972 as a successor to the B language, itself a simplified version of BCPL (Basic Combined Programming Language). The primary motivation for developing C was to provide a language that could be used to implement the Unix operating system. Unlike its predecessors, C introduced more structured programming constructs, including functions, loops, and conditionals, making it a more practical choice for systems programming.

The development of Unix in C marked a turning point in programming history. Unix's portability, facilitated by C, allowed it to run on various hardware platforms. This flexibility was groundbreaking at a time when software was often tightly coupled with specific hardware. By the late 1970s, C had become synonymous with systems programming.

**Standardization and Growth**

C's widespread adoption necessitated a standardized definition of the language. In 1983, the American National Standards Institute (ANSI) formed a committee to create a standard version of C, leading to the release of ANSI C in 1989 (also known as C89). This version addressed inconsistencies in earlier implementations and provided a uniform specification.

Subsequent updates, such as C99 and C11, introduced features like inline functions, variable-length arrays, and multithreading support, keeping the language relevant in an evolving technological landscape. Despite these enhancements, C retained its core philosophy of simplicity and efficiency, avoiding excessive complexity.

**Importance of C**

C's importance lies in its foundational role in computing. Its key contributions include:

1. **Portability:** C programs can be compiled on various platforms with minimal modifications, making it ideal for cross-platform development.
2. **Efficiency:** C provides low-level access to memory and system resources, allowing developers to write performance-critical applications.
3. **Simplicity:** With a relatively small set of keywords and straightforward syntax, C is accessible to beginners while offering powerful capabilities for advanced developers.
4. **Influence:** Many modern programming languages, including C++, Java, and Python, were influenced by C's syntax and structure.

**C’s Role in Modern Computing**

Despite the emergence of newer languages, C remains indispensable for several reasons:

1. **Systems Programming:** C is the go-to language for developing operating systems, embedded systems, and device drivers. Its ability to interact closely with hardware makes it irreplaceable in this domain.
2. **Legacy Systems:** Many critical software systems written in C continue to operate today. Maintaining and updating these systems requires proficiency in C.
3. **Educational Value:** C is often the first language taught in computer science programs because it introduces fundamental programming concepts like memory management, pointers, and data structures.
4. **Performance:** Applications demanding high performance, such as game engines, scientific computing, and real-time systems, frequently rely on C for its efficiency.

**Challenges and Limitations**

While C is powerful, it has limitations. Its lack of built-in memory safety features can lead to vulnerabilities like buffer overflows. Additionally, managing pointers and manual memory allocation can be error-prone, especially for inexperienced developers. However, these challenges are often viewed as opportunities for learning and mastering the principles of programming.

**Conclusion**

The history and evolution of C programming reflect its unparalleled contributions to the field of computer science. From its inception at Bell Labs to its continued use in modern applications, C has proven to be a versatile and enduring language. Its influence on subsequent programming languages, coupled with its ongoing importance in systems programming and education, ensures that C will remain a cornerstone of computing for years to come. For developers, understanding C is not just a nod to its historical significance but a pathway to mastering the fundamentals of efficient and effective programming.

LAB EXERCISE: o Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.

Ans. Done in Lab

2. Setting Up Environment

THEORY EXERCISE:• o 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.

Ans. Here is a comprehensive guide to installing a C compiler and setting up an IDE.

### ****1. Install GCC (GNU Compiler Collection)****

#### For Windows:

1. **Download MinGW (Minimalist GNU for Windows):**
   * Go to the MinGW website.
   * Download the MinGW installer.
2. **Run the Installer:**
   * Open the installer and proceed through the setup steps.
   * Choose the components to install (ensure "gcc" and "g++" are selected).
3. **Set Environment Variables:**
   * Find the installation directory (e.g., C:\MinGW\bin).
   * Add this path to the system's PATH variable:
     + Open **Control Panel** → **System** → **Advanced System Settings** → **Environment Variables**.
     + Under **System Variables**, find Path, edit it, and add C:\MinGW\bin.
4. **Verify Installation:**
   * Open Command Prompt and type gcc --version.
   * If installed correctly, it will display the GCC version.

### ****2. Install and Set Up an IDE****

#### Option 1: ****DevC++****

1. **Download DevC++:**
   * Visit DevC++'s official website.
   * Download the installer.
2. **Install DevC++:**
   * Run the installer and follow the on-screen instructions.
   * During installation, ensure that the bundled GCC compiler is selected.
3. **Configure DevC++:**
   * Open DevC++.
   * Go to **Tools** → **Compiler Options** and ensure the path to GCC is correctly set.

#### Option 2: ****Visual Studio Code (VS Code)****

1. **Download and Install VS Code:**
   * Visit [Visual Studio Code's website](https://code.visualstudio.com/) and download the installer.
   * Install it on your system.
2. **Install the C/C++ Extension:**
   * Open VS Code.
   * Go to the **Extensions Marketplace** (Ctrl+Shift+X).
   * Search for "C/C++" by Microsoft and install it.
3. **Configure the Compiler:**
   * Create a new file named tasks.json in the .vscode folder of your project:

json

Copy code

**Summary**

* Install GCC for your operating system.
* Choose an IDE (DevC++, VS Code, or Code::Blocks) and set it up.
* Configure the IDE to use the installed GCC compiler.
* Write and run a simple C program to ensure everything is working correctly.

LAB EXERCISE: o Install a C compiler on your system and configure the IDE. Write your first program to print "Hello, World!" and run it

Ans. Done in Lab

1. Basic Structure of a C Program

THEORY EXERCISE: o Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

Ans. The structure of a basic C program includes several essential components: headers, the main function, comments, data types, and variables. Here’s a breakdown of these components with an example:

**1. Headers**

* **Purpose**: Include necessary libraries for functions or features.
* **Syntax**: Use #include to include a library.
* **Example**: #include <stdio.h> allows input and output functions like printf and scanf.

**2. The main Function**

* **Purpose**: Serves as the entry point for program execution.
* **Syntax**: The main function is typically declared as int main() and returns an integer value (usually 0 for successful execution).

**3. Comments**

* **Purpose**: Add descriptions or explanations to make the code understandable without affecting execution.
* **Types**:
  + **Single-line comment**: // Comment
  + **Multi-line comment**: /\* Comment \*/

**4. Data Types**

* **Purpose**: Specify the type of data a variable can hold.
* **Common types**:
  + int: Integer numbers.
  + float: Decimal numbers.
  + char: Single character.
  + double: Double-precision floating-point numbers.

**5. Variables**

* **Purpose**: Store data values for the program to use.
* **Syntax**: Declare variables by specifying the data type and variable name (e.g., int number;).

Example

#include <stdio.h> // Header to include standard input/output functions

// Main function: Entry point of the program

int main() {

// Comments to explain the code

// Declare variables

int number; // An integer variable

float decimal; // A float variable

char character; // A character variable

// Assign values to the variables

number = 10; // Assigning an integer value

decimal = 3.14; // Assigning a decimal value

character = 'A'; // Assigning a character value

// Print the values of the variables

printf("Integer: %d\n", number); // %d for int

printf("Float: %.2f\n", decimal); // %.2f for float with 2 decimal places

printf("Character: %c\n", character); // %c for char

return 0; // Return 0 to indicate successful execution

}

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

Ans. Done in Lab

1. Operators in C THEORY EXERCISE:

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

Ans. Here are detailed notes on each type of operator in C, with explanations and examples:

**1. Arithmetic Operators**

* **Purpose**: Perform basic mathematical operations.
* **Operators**:
  + + (Addition): Adds two operands.
  + - (Subtraction): Subtracts the second operand from the first.
  + \* (Multiplication): Multiplies two operands.
  + / (Division): Divides the first operand by the second (integer division for int).
  + % (Modulus): Finds the remainder of the division of two integers.

**2. Relational Operators**

* **Purpose**: Compare two values and return a boolean result (1 for true, 0 for false).
* **Operators**:
  + == (Equal to)
  + != (Not equal to)
  + < (Less than)
  + > (Greater than)
  + <= (Less than or equal to)
  + >= (Greater than or equal to)

**3. Logical Operators**

* **Purpose**: Perform logical operations on boolean values.
* **Operators**:
  + && (Logical AND): True if both operands are true.
  + || (Logical OR): True if at least one operand is true.
  + ! (Logical NOT): Reverses the truth value.

**4. Assignment Operators**

* **Purpose**: Assign values to variables.
* **Operators**:
  + = (Simple assignment): Assigns the value of the right operand to the left.
  + += (Add and assign)
  + -= (Subtract and assign)
  + \*= (Multiply and assign)
  + /= (Divide and assign)
  + %= (Modulus and assign)

**5. Increment and Decrement Operators**

* **Purpose**: Increase or decrease the value of a variable by 1.
* **Operators**:
  + ++ (Increment): Increases by 1.
  + -- (Decrement): Decreases by 1.
* **Types**:
  + **Prefix**: Increments or decrements before using the value (++x, --x).
  + **Postfix**: Increments or decrements after using the value (x++, x--).

**6. Bitwise Operators**

* **Purpose**: Perform operations at the bit level.
* **Operators**:
  + & (Bitwise AND)
  + | (Bitwise OR)
  + ^ (Bitwise XOR)
  + ~ (Bitwise NOT)
  + << (Left shift)
  + >> (Right shift)

**7. Conditional (Ternary) Operator**

* **Purpose**: Evaluate a condition and return one of two values.
* **Syntax**: condition ? value\_if\_true : value\_if\_false
* **Summary Table**

| **Operator Type** | **Example** | **Description** |
| --- | --- | --- |
| **Arithmetic** | a + b, a % b | Math operations |
| **Relational** | a > b, a != b | Compare values |
| **Logical** | a && b, !a | Logical operations |
| **Assignment** | a += 5, a = b | Assign and modify |
| **Increment/Decrement** | a++, --b | Increment/decrement by 1 |
| **Bitwise** | a & b, a << 1 | Bit-level manipulation |
| **Conditional** | a > b ? a : b | Select one of two values based on a condition |
|  |  |  |

LAB EXERCISE: o Write a C program that accepts two integers from the user and performs arithmetic, relational, and logical operations on them. Display the results.

Ans. Done in Lab

1. Control Flow Statements in C THEORY EXERCISE:• o Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

Ans. In C, **decision-making statements** are used to execute specific code blocks based on certain conditions. These include if, else, nested if-else, and switch. Let’s explore each in detail with examples:

**1. if Statement**

* **Purpose**: Executes a block of code if a condition is true.
* **Syntax**:

c

CopyEdit

if (condition) {

// Code to execute if condition is true

}

* **Example**:

c

CopyEdit

int number = 5;

if (number > 0) {

printf("The number is positive.\n");

}

* **Explanation**: If number > 0 evaluates to true, the message is printed.

**2. if-else Statement**

* **Purpose**: Provides an alternative block of code to execute when the condition is false.
* **Syntax**:

c

CopyEdit

if (condition) {

// Code to execute if condition is true

} else {

// Code to execute if condition is false

}

* **Example**:

c

CopyEdit

int number = -3;

if (number > 0) {

printf("The number is positive.\n");

} else {

printf("The number is not positive.\n");

}

* **Explanation**: If the condition is false, the else block executes.

**3. Nested if-else Statement**

* **Purpose**: Allows multiple levels of decision-making.
* **Syntax**:

c

CopyEdit

if (condition1) {

// Code if condition1 is true

} else if (condition2) {

// Code if condition2 is true

} else {

// Code if none of the conditions are true

}

* **Example**:

c

CopyEdit

int number = 0;

if (number > 0) {

printf("The number is positive.\n");

} else if (number == 0) {

printf("The number is zero.\n");

} else {

printf("The number is negative.\n");

}

* **Explanation**: The conditions are checked sequentially. The first true condition’s block is executed, and the rest are ignored.

**4. switch Statement**

* **Purpose**: Executes one block of code out of multiple options based on the value of a variable or expression.
* **Syntax**:

c

CopyEdit

switch (expression) {

case value1:

// Code for value1

break;

case value2:

// Code for value2

break;

// Additional cases...

default:

// Code if no cases match

}

* **Example**:

c

CopyEdit

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**:
  + If day == 3, "Wednesday" is printed.
  + The break statement prevents execution of subsequent cases.
  + If no cases match, the default block executes.

**Comparison of if-else and switch**

| **Feature** | **if-else** | **switch** |
| --- | --- | --- |
| **Conditions** | Can check complex conditions | Checks only equality |
| **Scalability** | Good for a few conditions | Better for many discrete values |
| **Performance** | May be slower for many checks | Generally faster for many cases |

**Summary Table of Decision-Making Statements**

| **Statement** | **Purpose** | **Example Condition** |
| --- | --- | --- |
| if | Executes if a condition is true | if (x > 0) |
| if-else | Executes one of two blocks based on a condition | if (x > 0) else |
| Nested if-else | Handles multiple conditions in a hierarchical way | if (x > 0) else if (x == 0) |
| switch | Selects from multiple blocks based on a variable | switch (x) with case values |

These decision-making statements are foundational in C programming, enabling complex logic and flow control.

LAB EXERCISE:

Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the month name based on the user’s input (1 for January, 2 for February, etc.).

Ans. Done in Lab

Looping in C THEORY EXERCISE:

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

### Ans. Comparison of while, for, and do-while Loops in C

| **Feature** | **while Loop** | **for Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| **Definition** | Repeats a block of code as long as the condition is true. | Executes a block of code a specified number of times, or as determined by a condition. | Executes a block of code at least once, then repeats while the condition is true. |
| **Condition Check** | Checked **before** entering the loop. | Checked **before** each iteration. | Checked **after** executing the loop body. |
| **Usage** | Use when the number of iterations is not known beforehand. | Use when the number of iterations is known or can be calculated. | Use when the code must run at least once, regardless of the condition. |
| **Syntax Simplicity** | Simpler for indefinite iterations. | Best for iterations with initialization, condition, and increment/decrement in a single line. | Explicitly guarantees at least one execution. |
| **Risk of Infinite Loops** | Higher if the condition is not properly managed. | Lower since loop control is centralized. | Similar to while, but at least one execution occurs. |

**1. while Loop**

* **Syntax**:

c

CopyEdit

while (condition) {

// Code block to execute

}

* **Characteristics**:
  + Condition is checked **before** each iteration.
  + May execute zero times if the condition is false initially.
* **Best Scenario**:
  + When the number of iterations is not predetermined (e.g., reading input until the user enters a specific value).
* **Example**:

c

CopyEdit

int i = 1;

while (i <= 5) {

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

i++;

}

// Output: 1 2 3 4 5

**2. for Loop**

* **Syntax**:

c

CopyEdit

for (initialization; condition; increment/decrement) {

// Code block to execute

}

* **Characteristics**:
  + Combines initialization, condition, and update in a single statement.
  + Condition is checked **before** each iteration.
  + Ideal for fixed iteration counts.
* **Best Scenario**:
  + When the number of iterations is known beforehand (e.g., iterating over an array or running a loop a specific number of times).
* **Example**:

c

CopyEdit

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

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

}

// Output: 1 2 3 4 5

**3. do-while Loop**

* **Syntax**:

c

CopyEdit

do {

// Code block to execute

} while (condition);

* **Characteristics**:
  + The condition is checked **after** the code block executes.
  + Guarantees the loop runs at least once, regardless of the condition.
* **Best Scenario**:
  + When the code must run at least once before checking a condition (e.g., prompting the user for input and validating it).
* **Example**:

c

CopyEdit

int i = 1;

do {

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

i++;

} while (i <= 5);

// Output: 1 2 3 4 5

**Key Differences**

1. **Condition Check Timing**:
   * while and for check the condition **before** each iteration.
   * do-while checks the condition **after** the first execution.
2. **Guaranteed Execution**:
   * while and for might not execute if the condition is initially false.
   * do-while always executes at least once.
3. **Complexity**:
   * while: Best for simpler, condition-driven loops.
   * for: Best for iteration with a clear start, end, and step size.
   * do-while: Best for loops where the body must run before condition evaluation.

**When to Use Each Loop**

| **Loop Type** | **Scenario** |
| --- | --- |
| **while** | Reading input until a specific condition is met (e.g., user enters "exit"). |
| **for** | Iterating over an array, counting numbers, or performing operations a specific number of times. |
| **do-while** | Validating input where the prompt and initial execution must occur at least once before rechecking. |

These distinctions help programmers choose the most efficient loop structure for their specific task.

LAB EXERCISE: o Write a C program to print numbers from 1 to 10 using all three types of loops (while, for, do-while)

Ans. Done in Lab

Loop Control Statements

THEORY EXERCISE:•

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

Ans. In C programming, break, continue, and goto are control flow statements that allow you to alter the flow of execution in loops and other control structures. Here's an explanation and example of each:

### ****1.**** break ****Statement****

The break statement is used to exit a loop or switch statement prematurely, regardless of the loop condition. It is commonly used when a specific condition is met, and there's no need to continue further iterations.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

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

if (i == 5) {

break; // Exit the loop when i equals 5

}

printf("%d ", i);

}

return 0;

}

**Output:**  
1 2 3 4  
(The loop stops when i is 5.)

### ****2.**** continue ****Statement****

The continue statement skips the rest of the current iteration of the loop and jumps to the next iteration. It is often used to skip specific iterations based on a condition.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

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

if (i == 5) {

continue; // Skip the rest of the loop body when i equals 5

}

printf("%d ", i);

}

return 0;

}

**Output:**  
1 2 3 4 6 7 8 9 10  
(i == 5 is skipped.)

### ****3.**** goto ****Statement****

The goto statement allows you to transfer control to a labeled statement within the same function. It should be used cautiously as it can make code harder to read and maintain.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

int num = 1;

start:

printf("%d ", num);

num++;

if (num <= 5) {

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

}

printf("\nEnd of program.\n");

return 0;

}

**Output:**  
1 2 3 4 5  
End of program.

### Key Points:

* **break**: Exits a loop or switch block prematurely.
* **continue**: Skips the current iteration and proceeds to the next iteration.
* **goto**: Jumps to a labeled statement, useful in specific scenarios like error handling.

#### Avoid overusing goto:

While break and continue are straightforward and widely used, goto is generally discouraged as it can lead to "spaghetti code." Use it only when it simplifies error handling or cleanup operations.

LAB EXERCISE:

Write a C program that uses the break statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 using the continue statement.

Ans. Done in Lab

Functions in C THEORY EXERCISE:• o

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

Ans. Functions in C are blocks of code designed to perform specific tasks. They allow modularity, code reuse, and better readability. A function in C typically has the following components:

1. **Function Declaration (Prototype)**: Declares the function's name, return type, and parameters to inform the compiler about its existence.
2. **Function Definition**: Contains the actual implementation of the function.
3. **Function Call**: Executes the function from another part of the program.

### ****1. Function Declaration****

A function declaration (or prototype) specifies the function's signature and is typically placed before the main() function or in a header file. It informs the compiler about the function's:

* Return type
* Name
* Parameters (types and order)

#### Syntax:

c

CopyEdit

return\_type function\_name(parameter\_type1, parameter\_type2, ...);

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

A function definition contains the implementation of the function. It includes the code that executes when the function is called.

#### Syntax:

c

CopyEdit

return\_type function\_name(parameter\_type1 parameter1, parameter\_type2 parameter2, ...) {

// Code to execute

return value; // Optional, depends on return type

}

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

A function is invoked (called) from another function (like main()) using its name and passing the required arguments.

#### Syntax:

c

CopyEdit

function\_name(argument1, argument2, ...);

### ****Example: Function Declaration, Definition, and Call****

Here’s a complete program demonstrating these concepts:

c

CopyEdit

#include <stdio.h>

// Function Declaration

int add(int a, int b);

int main() {

int num1 = 5, num2 = 10, result;

// Function Call

result = add(num1, num2);

printf("The sum of %d and %d is %d.\n", num1, num2, result);

return 0;

}

// Function Definition

int add(int a, int b) {

return a + b; // Return the sum of two numbers

}

**Output:**

python

CopyEdit

The sum of 5 and 10 is 15.

### ****Explanation of the Example:****

1. **Declaration:** int add(int a, int b); tells the compiler that there is a function add that takes two int arguments and returns an int.
2. **Call:** result = add(num1, num2); invokes the function with the arguments num1 and num2.
3. **Definition:** The function add calculates the sum of a and b and returns the result.

### ****Key Points:****

* **Return Type:** Specifies the type of value the function returns. Use void if no value is returned.
* **Parameters:** Allow passing data to the function. Functions can have zero or more parameters.
* **Modularity:** Functions help break a program into smaller, manageable pieces.
* **Code Reuse:** Functions can be called multiple times with different arguments.

#### Example with void return type:

c

CopyEdit

#include <stdio.h>

// Function Declaration

void greet();

// Function Call

int main() {

greet();

return 0;

}

// Function Definition

void greet() {

printf("Hello, World!\n");

}

**Output:**  
Hello, World!

LAB EXERCISE: o Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.

Ans. Done in lab

9. Arrays in C THEORY EXERCISE:• o Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

Ans. An **array** in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow storing and managing multiple values using a single variable name and accessing elements using an index.

### ****Features of Arrays****

1. **Fixed Size:** The size of an array is defined during its declaration and cannot be changed dynamically.
2. **Homogeneous Data:** All elements must be of the same type.
3. **Indexed Access:** Array elements are accessed using zero-based indexing.

### ****Types of Arrays in C****

1. **One-Dimensional Array**
2. **Multi-Dimensional Array** (e.g., two-dimensional and three-dimensional arrays)

### ****1. One-Dimensional Array****

A one-dimensional array is a linear collection of elements, often visualized as a row of values.

#### Declaration:

c

CopyEdit

data\_type array\_name[size];

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

int numbers[5] = {1, 2, 3, 4, 5}; // Declare and initialize a 1D array

// Accessing elements

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

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

}

return 0;

}

**Output:**

css

CopyEdit

numbers[0] = 1

numbers[1] = 2

numbers[2] = 3

numbers[3] = 4

numbers[4] = 5

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

A multi-dimensional array is an array of arrays. The most common type is a **two-dimensional array**, often used to represent matrices or tables.

#### Declaration:

c

CopyEdit

data\_type array\_name[size1][size2]; // For a 2D array

data\_type array\_name[size1][size2][size3]; // For a 3D array

#### Example: Two-Dimensional Array

c

CopyEdit

#include <stdio.h>

int main() {

int matrix[2][3] = {

{1, 2, 3}, // First row

{4, 5, 6} // Second row

};

// Accessing elements

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

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

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

}

}

return 0;

}

**Output:**

css

CopyEdit

matrix[0][0] = 1

matrix[0][1] = 2

matrix[0][2] = 3

matrix[1][0] = 4

matrix[1][1] = 5

matrix[1][2] = 6

### ****Key Differences Between One-Dimensional and Multi-Dimensional Arrays****

| **Aspect** | **One-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| **Structure** | Linear (like a list) | Tabular or multi-level (like a table or cube) |
| **Declaration** | data\_type array\_name[size]; | data\_type array\_name[size1][size2]; |
| **Memory Representation** | Single row of contiguous memory locations | Array of arrays (e.g., rows and columns) |
| **Access** | Single index (array[index]) | Multiple indices (array[row][column]) |
| **Use Case** | Simple lists (e.g., student marks, scores) | Matrices, tables, or grids (e.g., chessboard) |

### ****Example: Three-Dimensional Array****

A three-dimensional array can represent a cube-like structure.

c

CopyEdit

#include <stdio.h>

int main() {

int cube[2][2][2] = {

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

};

// Accessing elements

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

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

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

printf("cube[%d][%d][%d] = %d\n", i, j, k, cube[i][j][k]);

}

}

}

return 0;

}

**Output:**

css

CopyEdit

cube[0][0][0] = 1

cube[0][0][1] = 2

cube[0][1][0] = 3

cube[0][1][1] = 4

cube[1][0][0] = 5

cube[1][0][1] = 6

cube[1][1][0] = 7

cube[1][1][1] = 8

### ****Conclusion****

* **One-Dimensional Arrays** are simple lists with linear indexing.
* **Multi-Dimensional Arrays** extend the concept to tables, grids, or higher-dimensional structures with multiple indices for access.

LAB EXERCISE: o Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

Ans. Done in Lab

Pointers in C THEORY EXERCISE:

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

Ans. A **pointer** in C is a variable that stores the **memory address** of another variable. Pointers are one of the most powerful features of C, allowing direct memory manipulation, dynamic memory allocation, and efficient data handling.

### ****1. Declaring and Initializing Pointers****

#### ****Declaration****

A pointer is declared by specifying the type of variable it will point to, followed by an asterisk (\*).

#### Syntax:

c

CopyEdit

data\_type \*pointer\_name;

* data\_type specifies the type of data the pointer will point to.
* \* indicates that the variable is a pointer.

#### ****Initialization****

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

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

int num = 10; // Normal variable

int \*ptr = &num; // Pointer initialized with the address of 'num'

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

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

printf("Value stored in ptr: %p\n", ptr);

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

return 0;

}

**Output:**

yaml

CopyEdit

Value of num: 10

Address of num: 0x7ffee5b5a7bc (example, will vary)

Value stored in ptr: 0x7ffee5b5a7bc

Value pointed to by ptr: 10

### ****2. Why Are Pointers Important in C?****

Pointers are essential in C programming for several reasons:

#### ****a. Efficient Memory Access****

Pointers allow direct access to memory, making operations faster and more efficient.

#### ****b. Dynamic Memory Allocation****

Pointers are used with functions like malloc(), calloc(), and free() to allocate and deallocate memory dynamically during runtime.

#### ****c. Passing by Reference****

Pointers enable functions to modify variables directly by passing their addresses, avoiding the overhead of copying large data structures.

#### ****d. Data Structures****

Pointers are the foundation for building dynamic data structures like linked lists, trees, and graphs.

#### ****e. Array Manipulation****

Pointers allow efficient traversal and manipulation of arrays.

#### ****f. Function Pointers****

Pointers can be used to store and call functions, enabling dynamic behavior.

### ****3. Example: Basic Pointer Operations****

c

CopyEdit

#include <stdio.h>

int main() {

int a = 20, b = 30;

int \*p1 = &a, \*p2 = &b;

printf("Before swapping: a = %d, b = %d\n", a, b);

// Swap using pointers

int temp = \*p1;

\*p1 = \*p2;

\*p2 = temp;

printf("After swapping: a = %d, b = %d\n", a, b);

return 0;

}

**Output:**

less

CopyEdit

Before swapping: a = 20, b = 30

After swapping: a = 30, b = 20

### ****4. Null Pointers****

A pointer that is not initialized explicitly can point to garbage values. To avoid this, pointers are often initialized to NULL.

#### Syntax:

c

CopyEdit

int \*ptr = NULL;

A NULL pointer does not point to any valid memory location.

### ****5. Example: Dynamic Memory Allocation****

c

CopyEdit

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*ptr;

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

// Allocate memory dynamically

ptr = (int \*)malloc(n \* sizeof(int));

// Check if memory allocation was successful

if (ptr == NULL) {

printf("Memory allocation failed!\n");

return 1;

}

// Assign values

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

ptr[i] = i + 1;

}

// Print values

printf("Array elements: ");

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

printf("%d ", ptr[i]);

}

// Free the allocated memory

free(ptr);

return 0;

}

### ****Key Points:****

1. **Declaration**: int \*ptr;
2. **Initialization**: ptr = &variable;
3. **Dereferencing**: Access the value using \*ptr.
4. **Null Pointer**: A pointer with no valid address, ptr = NULL.
5. **Dynamic Memory Allocation**: Allocate memory during runtime using pointers.

Pointers are a fundamental concept in C programming, providing flexibility and control over memory and enabling powerful programming techniques.

Write a C program to demonstrate pointer usage. Use a pointer to modify the value of a variable and print the result.

Ans. Done in Lab

11. Strings in C THEORY EXERCISE:

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

Ans. String handling functions are part of the standard C library (<string.h>), and they provide various utilities for working with null-terminated strings (C strings). Here's an explanation of commonly used string functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(), along with examples:

### 1. strlen()

* **Purpose**: Computes the length of a string (excluding the null-terminator \0).
* **Prototype**:

c

CopyEdit

size\_t strlen(const char \*str);

* **Use Case**: Determine the length of a string to allocate memory dynamically or loop through characters.

#### Example:

c

CopyEdit

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Hello, World!";

printf("Length of string: %zu\n", strlen(str)); // Output: 13

return 0;

}

### 2. strcpy()

* **Purpose**: Copies a source string into a destination buffer, including the null-terminator.
* **Prototype**:

c

CopyEdit

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

* **Use Case**: Copy one string into another (e.g., for initializing a new string).

#### Example:

c

CopyEdit

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "Copy this string";

char dest[50];

strcpy(dest, src);

printf("Destination string: %s\n", dest); // Output: Copy this string

return 0;

}

### 3. strcat()

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

c

CopyEdit

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

* **Use Case**: Combine two strings into one.

#### Example:

c

CopyEdit

#include <stdio.h>

#include <string.h>

int main() {

char str1[50] = "Hello, ";

char str2[] = "World!";

strcat(str1, str2);

printf("Concatenated string: %s\n", str1); // Output: Hello, World!

return 0;

}

### 4. strcmp()

* **Purpose**: Compares two strings lexicographically (character by character).
* **Prototype**:

c

CopyEdit

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

* **Return Values**:
  + 0: Strings are equal.
  + < 0: str1 is less than str2.
  + > 0: str1 is greater than str2.
* **Use Case**: Check if two strings are identical or compare their order.

#### Example:

c

CopyEdit

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Apple";

char str2[] = "Orange";

if (strcmp(str1, str2) == 0) {

printf("Strings are equal\n");

} else {

printf("Strings are not equal\n"); // Output: Strings are not equal

}

return 0;

}

### 5. strchr()

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

c

CopyEdit

char \*strchr(const char \*str, int c);

* **Use Case**: Locate a specific character in a string (e.g., for parsing).

#### Example:

c

CopyEdit

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Find the first occurrence of a character.";

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

if (pos) {

printf("Found 'f' at position: %ld\n", pos - str); // Output: Found 'f' at position: 5

} else {

printf("'f' not found.\n");

}

return 0;

}

Write a C program that takes two strings from the user and concatenates them using strcat(). Display the concatenated string and its length using strlen().

Ans. Done in Lab

12. Structures in C THEORY EXERCISE:

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

### Ans. Structures in C

A **structure** in C is a user-defined data type that allows grouping of variables of different types into a single logical unit. Structures are useful for organizing and representing complex data more efficiently.

### Declaring a Structure

To declare a structure, use the struct keyword, followed by the structure's definition. The syntax is:

c

CopyEdit

struct StructureName {

dataType1 member1;

dataType2 member2;

...

dataTypeN memberN;

};

#### Example:

c

CopyEdit

struct Point {

int x;

int y;

};

### Initializing a Structure

You can initialize a structure during declaration or later in the program.

#### 1. ****During Declaration:****

c

CopyEdit

struct Point p1 = {10, 20};

#### 2. ****Assigning Values to Members:****

c

CopyEdit

struct Point p2;

p2.x = 30;

p2.y = 40;

### Accessing Structure Members

To access a structure's members, use the **dot operator (.)** with the structure variable.

#### Example:

c

CopyEdit

#include <stdio.h>

struct Point {

int x;

int y;

};

int main() {

struct Point p1;

p1.x = 5;

p1.y = 10;

printf("Point: (%d, %d)\n", p1.x, p1.y); // Output: Point: (5, 10)

return 0;

}

### Array of Structures

Structures can be used in arrays to store multiple objects of the same type.

#### Example:

c

CopyEdit

#include <stdio.h>

struct Student {

int id;

char name[50];

float marks;

};

int main() {

struct Student students[2] = {

{1, "Alice", 85.5},

{2, "Bob", 90.0}

};

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

printf("ID: %d, Name: %s, Marks: %.2f\n", students[i].id, students[i].name, students[i].marks);

}

return 0;

}

### Structure with Pointers

Structures can also include pointers or be accessed via pointers using the **arrow operator (->)**.

#### Example:

c

CopyEdit

#include <stdio.h>

struct Point {

int x;

int y;

};

int main() {

struct Point p = {15, 25};

struct Point \*ptr = &p;

printf("x = %d, y = %d\n", ptr->x, ptr->y); // Output: x = 15, y = 25

return 0;

}

### Nested Structures

Structures can contain other structures as members.

#### Example:

c

CopyEdit

#include <stdio.h>

struct Address {

char city[50];

int zipCode;

};

struct Student {

char name[50];

struct Address address;

};

int main() {

struct Student s = {"John", {"New York", 10001}};

printf("Name: %s, City: %s, Zip: %d\n", s.name, s.address.city, s.address.zipCode);

return 0;

}

### Use Cases of Structures

* Grouping related data (e.g., student details, employee records).
* Modeling complex entities (e.g., geometry points, vehicles).
* Creating data structures like linked lists, stacks, queues, etc.

### Summary Table

| **Operation** | **Syntax/Example** |
| --- | --- |
| **Declaration** | struct Point { int x, y; }; |
| **Initialization** | struct Point p = {10, 20}; |
| **Access Members** | p.x = 5; printf("%d", p.x); |
| **Pointer Access** | struct Point \*ptr = &p; ptr->x; |
| **Array of Structures** | struct Point arr[5]; |
| **Nested Structures** | Structures within structures. |

Structures are a fundamental feature in C, providing a powerful way to manage complex data types.

Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3 students and print them.

Ans. Done in Lab

13. File Handling in C THEORY EXERCISE:

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

### Ans. Importance of File Handling in C

File handling in C is essential for managing persistent storage of data. It allows programs to store data permanently in files and retrieve it later. This is crucial for applications like databases, logging systems, configuration management, and data sharing between programs.

### Common File Operations

1. **Opening a File**: Preparing the file for reading, writing, or appending.
2. **Closing a File**: Ensuring the file is saved and resources are freed.
3. **Reading from a File**: Extracting data stored in the file.
4. **Writing to a File**: Storing data into the file.

File operations in C are managed using the <stdio.h> library, which provides a set of functions for working with files.

### File Pointers

C uses a **file pointer** to interact with files. It is declared as:

c

CopyEdit

FILE \*filePointer;

### 1. ****Opening a File****

The fopen() function is used to open a file. The syntax is:

c

CopyEdit

FILE \*fopen(const char \*filename, const char \*mode);

#### Modes of Opening:

| **Mode** | **Description** |
| --- | --- |
| "r" | Open for reading. File must exist. |
| "w" | Open for writing. Creates an empty file or overwrites an existing one. |
| "a" | Open for appending. Adds data at the end of the file. |
| "r+" | Open for both reading and writing. File must exist. |
| "w+" | Open for both reading and writing. Overwrites an existing file or creates one. |
| "a+" | Open for both reading and appending. |

#### Example:

c

CopyEdit

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

if (fp == NULL) {

printf("File cannot be opened.\n");

}

### 2. ****Closing a File****

The fclose() function is used to close a file. It ensures that all buffers are flushed, and resources are freed.

#### Syntax:

c

CopyEdit

int fclose(FILE \*filePointer);

#### Example:

c

CopyEdit

fclose(fp);

### 3. ****Reading from a File****

* **fgetc()**: Reads a single character.
* **fgets()**: Reads a string up to a newline or EOF.
* **fread()**: Reads binary data.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

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

if (fp == NULL) {

printf("File cannot be opened.\n");

return 1;

}

char ch;

while ((ch = fgetc(fp)) != EOF) {

putchar(ch); // Print each character to the console

}

fclose(fp);

return 0;

}

### 4. ****Writing to a File****

* **fputc()**: Writes a single character.
* **fputs()**: Writes a string.
* **fwrite()**: Writes binary data.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

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

if (fp == NULL) {

printf("File cannot be opened.\n");

return 1;

}

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

fclose(fp);

printf("Data written to file.\n");

return 0;

}

### 5. ****Other File Operations****

* **ftell()**: Returns the current position in the file.
* **fseek()**: Moves the file pointer to a specific position.
* **feof()**: Checks if the end of the file has been reached.

#### Example:

c

CopyEdit

#include <stdio.h>

int main() {

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

if (fp == NULL) {

printf("File cannot be opened.\n");

return 1;

}

fseek(fp, 0, SEEK\_END); // Move to the end of the file

long size = ftell(fp); // Get the file size

printf("File size: %ld bytes\n", size);

fclose(fp);

return 0;

}

### Benefits of File Handling

1. **Persistent Storage**: Data remains available even after the program ends.
2. **Data Sharing**: Files allow data exchange between applications.
3. **Logging and Debugging**: Files help log events and errors.
4. **Data Management**: Enables structured and efficient storage of data.

Write a C program to create a file, write a string into it, close the file, then open the file again to read and display its contents

Ans. Done in Lab

**EXTRA LAB EXERCISES FOR IMPROVING PROGRAMMING LOGIC**

1. Operators LAB EXERCISE

1: Simple Calculator Write a C program that acts as a simple calculator. The program should take two numbers and an operator as input from the user and perform the respective operation (addition, subtraction, multiplication, division, or modulus) using operators. Challenge: Extend the program to handle invalid operator inputs.

Ans. Done in Lab

LAB EXERCISE 2: Check Number Properties

Write a C program that takes an integer from the user and checks the following using different operators: o Whether the number is even or odd. o Whether the number is positive, negative, orzero. o Whether the number is a multiple of both 3 and 5.

Ans. Done in Lab

2. Control Statements LAB EXERCISE

1: Grade Calculator

Write a C program that takes the marks of a student as input and displays the corresponding grade based on the following conditions: o Marks > 90: Grade A o Marks > 75 and <= 90: Grade B o Marks > 50 and <= 75: Grade C o Marks <= 50: Grade D Use if-else orswitch statements for the decision-making process.

Ans. Done in Lab

Write a C program that takes three numbers from the user and determines: o The largest number. o The smallest number. Challenge: Solve the problem using both if-else and switch-case statements.•

Ans. Done in lab

Write a C program that checks whether a given number is a prime number or not using a for loop. Challenge: Modify the program to print all prime numbers between 1 and a given number.•

Ans. Done in Lab

Write a C program that takes a string as input and reverses it using a function. Challenge: Write the program without using built-in string handling functions•

Ans. Done in Lab