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.

**Introduction**

C programming language, developed in the early 1970s, has played a pivotal role in the evolution of computer science and software development. Its design philosophy, efficiency, and versatility have made it a foundational language in the programming world. This essay explores the history and evolution of C, its significance in the realm of programming, and the reasons for its continued relevance in modern computing.

**Origins of C**

The C programming language was developed by Dennis Ritchie at Bell Labs between 1969 and 1973. It was initially created as an evolution of the B language, which itself was derived from BCPL (Basic Combined Programming Language). Ritchie, along with his colleague Ken Thompson, aimed to create a language that could efficiently implement the Unix operating system, which was also being developed at Bell Labs during this period.

C was designed to provide low-level access to memory and system resources while maintaining a high-level syntax that was easier to read and write than assembly language. The first significant implementation of C was in 1972, and by 1978, Ritchie and Brian Kernighan published "The C Programming Language," which became the definitive guide for the language and contributed to its widespread adoption.

**Standardization and Evolution**

The 1980s marked a significant period for C, as the language began to gain traction in both academic and commercial settings. In 1983, the American National Standards Institute (ANSI) formed a committee to standardize C, resulting in the ANSI C standard, also known as C89 or C90. This standardization helped unify various dialects of C that had emerged, ensuring compatibility and portability across different systems.

In the following years, C continued to evolve. The introduction of C99 in 1999 brought several new features, including support for inline functions, variable-length arrays, and new data types such as long long int. The C11 standard, released in 2011, further enhanced the language with features like multi-threading support and improved Unicode handling. The most recent standard, C18, was published in 2018, primarily focusing on bug fixes and clarifications rather than introducing new features.

**Importance of C Programming**

C's importance in the programming landscape cannot be overstated. It serves as the foundation for many modern programming languages, including C++, C#, and Objective-C. Its influence extends to languages like Java and Python, which borrow concepts and syntax from C. The language's efficiency and performance make it ideal for system programming, embedded systems, and applications requiring direct hardware manipulation.

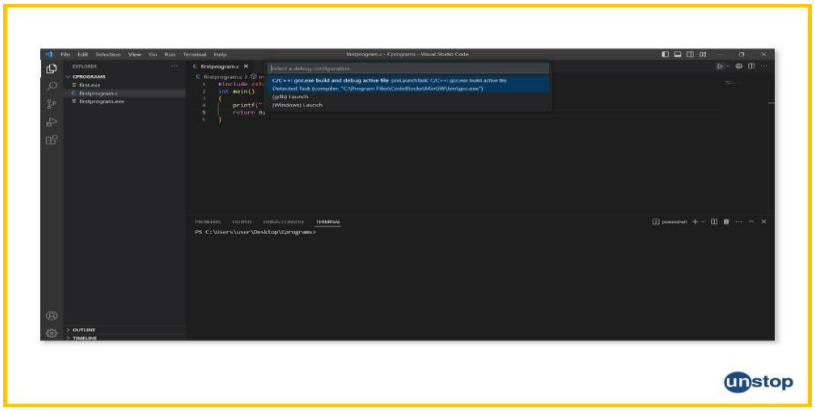
One of C's most significant contributions is its role in the development of operating systems. The Unix operating system, written in C, set a precedent for portability and modularity, influencing the design of subsequent operating systems, including Linux and macOS. C's ability to interact closely with hardware has made it the language of choice for developing firmware and drivers, ensuring that it remains relevant in the age of IoT (Internet of Things) and embedded systems.

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.

To install a C compiler like GCC and an IDE like DevC++, VS Code, or Code::Blocks, you'll generally download the compiler, potentially an IDE, and configure your system to recognize the compiler's location. For some IDEs, you might also need to install specific extensions or plugins.




1. Installing GCC (GNU Compiler Collection)

- **Windows:**
    - Download MinGW-w64, a free and open-source compiler suite for Windows.
    - Extract the downloaded file to a folder (e.g., C:\MinGW).
    - Install the MinGW Installation Manager.
    - In the manager, select the GCC package and click "Install".

        - Add the MinGW's bin directory (e.g., C:\MinGW\bin) to your system's PATH environment variable.
- **Linux:**
    - Use your distribution's package manager (e.g., apt on Debian/Ubuntu, yum on Fedora/CentOS).
    - Install GCC and G++: sudo apt-get install gcc g++ or sudo yum install gcc gcc-c++.

2. Setting up an IDE (Integrated Development Environment)

- **DevC++:**
    - Download the DevC++ installer from the official website.
    - Run the installer and follow the on-screen instructions.
    - DevC++ typically comes with GCC, so you should be ready to compile after installation.
- **VS Code:**
    - Download and install VS Code from the official website.
    - Install the C/C++ extension from the Extensions view (Ctrl+Shift+X, then search for "C++").
    - Ensure GCC (or another C compiler) is installed on your system and its path is included in your PATH environment variable.
    - Configure VS Code to use your compiler:
        - Create a launch.json and settings.json file in your workspace.
        - Specify the compiler path in the launch.json and settings.json files.
- **Code::Blocks:**
    - Download the Code::Blocks installer (with or without MinGW) from the official website.
    - Run the installer and follow the on-screen instructions.
    - If you choose the version with MinGW, it should automatically install and configure the GCC compiler.
    - If you have an existing compiler (like MinGW), you might need to manually configure Code::Blocks to use it.
    - Navigate to "Settings" > "Compilers" and specify the compiler path.

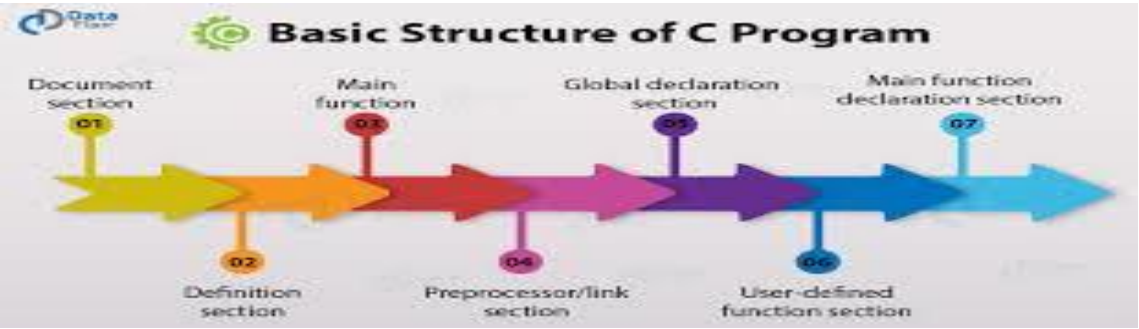
3. Compiling and Running Your Code

- **With GCC (from command line):**
    - Save your C code in a file (e.g., hello.c).
    - Open a terminal or command prompt and navigate to the directory where you saved the file.
    - Compile the code: gcc hello.c -o hello.exe (on Windows) or gcc hello.c -o hello (on Linux).
    - Run the compiled executable: .\hello.exe (on Windows) or ./hello (on Linux).
- **With an IDE:**
    - Open your IDE and create a new project.
    - Write your C code in the editor.
    - Use the IDE's build or compile button to compile the code.
    - Use the IDE's run button to execute the compiled program.

3. 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.




**Basic Structure of a C Program**

1. **Headers**:
    - Headers are included at the beginning of a C program to provide declarations for functions and macros that are used in the program. The most common header is **<stdio.h>**, which is used for input and output functions like **printf** and **scanf**.

      Example: #include <stdio.h> // Standard Input/Output header

**Main Function**:

- The **main** function is the entry point of every C program. It is where the execution of the program begins. The **main** function can return an integer value, typically **0** to indicate successful execution.

**Comments**:

- Comments are used to explain the code and make it more readable. They are ignored by the compiler. C supports two types of comments: single-line comments (using **//**) and multi-line comments (using **/* ... */**).

**Data Types**:

- C supports several built-in data types, including:
    - **int**: for integers
    - **float**: for floating-point numbers
    - **double**: for double-precision floating-point numbers
    - **char**: for single characters

**Variables**:

- Variables are used to store data that can be modified during program execution. Each variable must be declared with a specific data type before it can be used.

**Statements:**

- Statements are instructions that the program executes.
- Example: printf("Hello, world!");

4. 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.

**1. Arithmetic Operators**

Arithmetic operators are used to perform basic mathematical operations.

- **Addition (+)**: Adds two operands.
    - Example: **a + b**
- **Subtraction (-)**: Subtracts the second operand from the first.
    - Example: **a - b**
- **Multiplication (*)**: Multiplies two operands.

    - Example: **a * b**
- **Division (/)**: Divides the numerator by the denominator.
    - Example: **a / b** (Note: If both **a** and **b** are integers, the result is also an integer.)
- **Modulus (%)**: Returns the remainder of a division operation.
    - Example: **a % b**

**2. Relational Operators**

Relational operators are used to compare two values.

- **Equal to (==)**: Checks if two operands are equal.
    - Example: **a == b**
- **Not equal to (!=)**: Checks if two operands are not equal.
    - Example: **a != b**
- **Greater than (>)**: Checks if the left operand is greater than the right.
    - Example: **a > b**
- **Less than (<)**: Checks if the left operand is less than the right.
    - Example: **a < b**
- **Greater than or equal to (>=)**: Checks if the left operand is greater than or equal to the right.
    - Example: **a >= b**
- **Less than or equal to (<=)**: Checks if the left operand is less than or equal to the right.
    - Example: **a <= b**

**3. Logical Operators**

Logical operators are used to combine multiple conditions.

- **Logical AND (&&)**: Returns true if both operands are true.
    - Example: **(a > b) && (c > d)**
- **Logical OR (||)**: Returns true if at least one of the operands is true.
    - Example: **(a > b) || (c > d)**
- **Logical NOT (!)**: Reverses the logical state of its operand.

- Example: **!(a > b)**

**4. Assignment Operators**

Assignment operators are used to assign values to variables.

- **Simple assignment (=)**: Assigns the right operand's value to the left operand.
  - Example: **a = b**
- **Add and assign (+=)**: Adds the right operand to the left operand and assigns the result to the left operand.
  - Example: **a += b** (equivalent to **a = a + b**)
- **Subtract and assign (-=)**: Subtracts the right operand from the left operand and assigns the result to the left operand.
  - Example: **a -= b**
- **Multiply and assign (*=)**: Multiplies the left operand by the right operand and assigns the result to the left operand.
  - Example: **a *= b**
- **Divide and assign (/=)**: Divides the left operand by the right operand and assigns the result to the left operand.
  - Example: **a /= b**
- **Modulus and assign (%=)**: Takes the modulus using two operands and assigns the result to the left operand.
  - Example: **a %= b**

**5. Increment/Decrement Operators**

These operators are used to increase or decrease the value of a variable by one.

- **Increment (++)**: Increases the value of a variable by one.
  - Example: **++a** (pre-increment) or **a++** (post-increment)
- **Decrement (--)**: Decreases the value of a variable by one.
  - Example: **--a** (pre-decrement) or **a--** (post-decrement)

**6. Bitwise Operators**

Bitwise operators perform operations on binary representations of integers.

- **Bitwise AND (&)**: Compares each bit of two operands; returns 1 if both bits are 1.

    - Example: **a & b**
- **Bitwise OR (|)**: Compares each bit of two operands; returns 1 if at least one bit is 1.
    - Example: **a | b**
- **Bitwise XOR (^)**: Compares each bit of two operands; returns 1 if the bits are different.
    - Example: **a ^ b**
- **Bitwise NOT (~)**: Inverts all bits of the operand.
    - Example: **~a**
- **Left shift (<<)**: Shifts bits to the left, filling with

5. Control Flow Statements in C

• THEORY EXERCISE:

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

**1. if Statement**

The **if** statement evaluates a condition and executes a block of code if the condition is true.

**Syntax:**

c

```
if (condition) {
   // code to be executed if condition is true
}
```

**Example:**

c

```
#include <stdio.h>
int main() {
int a = 10;
if (a > 5) {
printf("a is greater than 5\n");
}
return 0;
```

**2. else Statement**

The **else** statement can be used in conjunction with the **if** statement to execute a block of code when the condition is false.

**Syntax:**

c

```
if (condition) {
 // code if condition is true
} else {
 // code if condition is false
}
```

**3. Nested if-else**

Nested **if-else** statements allow for multiple conditions to be checked in a hierarchical manner.

**Syntax:**

c

```
if (condition1) {
 // code if condition1 is true
} else if (condition2) {
 // code if condition2 is true
} else {
 // code if both conditions are false
}
```

**4. switch Statement**

The **switch** statement is used to execute one block of code among many based on the value of a variable or expression. It is often used as an alternative to multiple **if-else** statements when comparing the same variable against different values.

**Syntax:**

c

```
switch (expression) {
case constant1:
 // code to be executed if expression equals constant1
 break;
case constant2:
 // code to be executed if expression equals constant2
 break;
  // more cases...
 default:
  // code to be executed if expression doesn't match any case
}
```

6. Looping in C

• THEORY EXERCISE:

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

**1. while Loop**

```
while (condition) {
   // code to be executed as long as condition is true
}
```

**Characteristics:**

- The **while** loop checks the condition before executing the loop body.
- If the condition is false at the start, the loop body will not execute at all.
- It is suitable for situations where the number of iterations is not known beforehand and depends on a condition.

**When to Use:**

- Use a **while** loop when the number of iterations is not predetermined and depends on a condition that may change during execution.
- Ideal for reading data until a certain condition is met (e.g., reading user input until a specific value is entered).

**2. for Loop**

**Syntax:**

```
for (initialization; condition; increment/decrement) {
   // code to be executed as long as condition is true
}
```

**Characteristics:**

- The **for** loop is typically used when the number of iterations is known beforehand.
- It combines initialization, condition checking, and increment/decrement in a single line, making it concise and easy to read.
- The loop body executes as long as the condition is true.

**When to Use:**

- Use a **for** loop when the number of iterations is known in advance (e.g., iterating over an array or a fixed range).
- Ideal for counting iterations or when you need to perform a specific action a set number of times.

**3. do-while Loop**

**Syntax:**

```
do {
   // code to be executed
} while (condition);
```

**Characteristics:**

- The **do-while** loop executes the loop body at least once before checking the condition.
- The condition is checked after the loop body has executed, which guarantees that the loop body will run at least once.

**When to Use:**

- Use a **do-while** loop when you want to ensure that the loop body is executed at least once, regardless of the condition.
- Ideal for scenarios where the initial execution of the loop body is necessary (e.g., prompting a user for input and validating it).

**Summary of Comparisons**

| Feature | while Loop | for Loop | do-while Loop |
|---|---|---|---|
| Condition Check | Before executing the loop body | Before executing the loop body | After executing the loop body |
| Execution Guarantee | May not execute if condition is false | | |

7. Loop Control Statements

• THEORY EXERCISE:

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

**1. break Statement**

The **break** statement is used to terminate the nearest enclosing loop (**for**, **while**, or **do-while**) or a **switch** statement. When a **break** statement is encountered, control is transferred to the statement immediately following the loop or **switch**.

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // Exit the loop when i is 5
        }
        printf("%d\n", i);
}
    printf("Loop terminated at i = 5\n");
    return 0;
}
```

**2. continue Statement**

The **continue** statement is used to skip the current iteration of a loop and proceed to the next iteration. In a **for** loop, it jumps to the increment/decrement step, while in a **while** or **do-while** loop, it jumps to the condition check.

**Example:**

```
#include <stdio.h>


int main() {
   for (int i = 0; i < 10; i++) {
      if (i % 2 == 0) {
         continue; // Skip the even numbers
      }
      printf("%d\n", i);
   }


   return 0;
}
```

**3. goto Statement**

The **goto** statement is used to transfer control to a labeled statement within the same function. It can lead to less readable and maintainable code, so its use is generally discouraged in favor of structured control flow constructs like loops and conditionals.

**Example:**

```
#include <stdio.h>


int main() {
   int i = 0;


   loop_start:
   if (i < 5) {
      printf("%d\n", i);
      i++;
      goto loop_start; // Jump back to the labeled statement
```

```
    }


    return 0;
}
```

- **break**: Exits the nearest enclosing loop or **switch** statement. Use it to terminate loops based on a condition.
- **continue**: Skips the current iteration of a loop and proceeds to the next iteration. Use it to filter out specific cases within loops.
- **goto**: Transfers control to a labeled statement. Use it cautiously, as it can lead to unstructured and hard-to-read code.

8. Functions in C

• THEORY EXERCISE:

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

**Function Declaration**: This tells the compiler about the function's name, return type, and parameters. It is also known as a function prototype. The declaration is usually placed at the beginning of the program or in a header file.

**Syntax**:

return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...);

**Example**:

int add(int a, int b);

**Function Definition**: This is where the actual body of the function is written. It includes the code that will be executed when the function is called.

**Syntax**:

return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...) {

    // function body

}

**Example**:

int add(int a, int b) {

```
    return a + b;
}
```

**Function Call**: This is how you invoke or execute the function. You can call a function by using its name followed by parentheses containing any required arguments.

**Syntax**:

function_name(argument1, argument2, ...);

**Example**:

int result = add(5, 3);

1. **Function Declaration**: **int add(int a, int b);** informs the compiler that there is a function named **add** that takes two integers as parameters and returns an integer.
2. **Function Definition**: The function **add** is defined after the **main** function. It takes two integers, adds them, and returns the result.
3. **Function Call**: Inside the **main** function, **add(num1, num2);** calls the **add** function with **num1** and **num2** as arguments. The result is stored in the variable **result**, which is then printed.

9. Arrays in C

• THEORY EXERCISE:

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

**One-Dimensional Arrays**

A **one-dimensional array** (or simply an array) is a linear collection of elements. You can think of it as a list of items, where each item can be accessed using a single index.

**Declaration and Initialization**

**Syntax**:

data_type array_name[array_size];

**Multi-Dimensional Arrays**

A **multi-dimensional array** is an array of arrays. The most common type is the two-dimensional array, which can be thought of as a table with rows and columns.

**Declaration and Initialization**

**Syntax**:

data_type array_name[size1][size2];

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

| **Feature** | **One-Dimensional Array** | **Multi-Dimensional Array** |
|---|---|---|
| Structure | Linear (single row) | Tabular (multiple rows and columns) |
| Declaration Syntax | data_type array_name[size]; | data_type array_name[size1][size2]; |
| Accessing Elements | array_name[index]; | array_name[index1][index2]; |
| Example | int arr[5]; | int matrix[3][4]; |
| Use Case | Storing a list of items | Storing data in a grid or table format |

10. Pointers in C

• THEORY EXERCISE:

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

In C programming, a **pointer** is a variable that stores the memory address of another variable. Pointers are a powerful feature of C that allow for dynamic memory management, efficient array handling, and the ability to create complex data structures like linked lists and trees.

**Declaration and Initialization of Pointers**

**Declaration**

To declare a pointer, you use the asterisk (*****) symbol before the pointer's name. The type of the pointer must match the type of the variable it points to.

**Syntax**:

data_type *pointer_name;

**Initialization**

You can initialize a pointer by assigning it the address of a variable using the address-of operator (**&**).

**Importance of Pointers in C**

1. **Dynamic Memory Management**: Pointers allow for dynamic allocation and deallocation of memory using functions like **malloc()**, **calloc()**, and **free()**. This is essential for creating data structures whose size can change at runtime.

2. **Efficient Array Handling**: Pointers can be used to iterate through arrays efficiently. Instead of using array indexing, you can use pointer arithmetic to access array elements.
3. **Function Arguments**: Pointers enable passing large structures or arrays to functions without copying the entire data. This is done by passing the address of the variable, which is more efficient in terms of memory and performance.
4. **Data Structures**: Pointers are fundamental for implementing complex data structures like linked lists, trees, and graphs. They allow for flexible and dynamic data organization.
5. **Low-Level Programming**: Pointers provide a way to manipulate memory directly, which is useful in systems programming, embedded systems, and performance-critical applications.

11. Strings in C

• THEORY EXERCISE:

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

**1. strlen()**

**Description**: The **strlen()** function calculates the length of a string (excluding the null terminator).

**Syntax**:

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

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

**Example**:

```
int main() {
    const char *str = "Hello, World!";
    size_t length = strlen(str);
    printf("Length of the string: %zu\n", length); // Output: 13
    return 0;
}
```

**2. strcpy()**

**Description**: The **strcpy()** function copies a string from the source to the destination.

**Syntax**:

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

**Example**:

```
#include <stdio.h>

#include <string.h>


int main() {

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

   char destination[50]; // Ensure destination is large enough


   strcpy(destination, source);

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

   return 0;

}
```

**Strcat()**

**Description**: The **strcat()** function concatenates (appends) one string to the end of another.

**Syntax**:

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

```
#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;
}
```

**strcmp()**

**Description**: The **strcmp()** function compares two strings lexicographically.

**Syntax**:

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

**Example**:

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

int main() {
    const char *str1 = "Hello";
    const char *str2 = "World";

    int result = strcmp(str1, str2);
    if (result < 0) {
        printf("%s is less than %s\n", str1, str2);
    } else if (result > 0) {
        printf("%s is greater than %s\n", str1, str2);
    } else {
        printf("%s is equal to %s\n", str1, str2);
    }
    return 0;
}
```

**Description**: The **strchr()** function locates the first occurrence of a character in a string.

**Syntax**:

char *strchr(const char *str, int character);

**Example**:

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

int main() {
    const char *str = "Hello, World!";
    char ch = 'o';

    char *ptr = strchr(str, ch);
    if (ptr != NULL) {
        printf("First occurrence of '%c': %s\n", ch, ptr); // Output: o, World!
    } else {
        printf("Character not found.\n");
    }
    return 0;
}
```

**Use Case**: **strchr()** is useful for parsing strings, such as finding delimiters in a CSV file or searching for specific characters in user input.

12. Structures in C

• THEORY EXERCISE:

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

In C, a structure (struct) is a composite data type that groups variables of different types into a single unit. It is useful for representing complex data objects with multiple attributes.

**1. Declaring a Structure**

You declare a structure using the struct keyword, followed by the structure name and a block containing its members.

Example:

```
struct Person {
   char name[50];
   int age;
   float height;
};
```

This defines a structure named Person with three members: name (a string), age (an integer), and height (a floating-point number).

**2. Initializing a Structure**

You can initialize a structure variable in several ways:

- **At the time of declaration:**

```
struct Person person1 = {"Alice", 30, 5.5f};
```

- **After declaration, by assigning values to members:**

```
struct Person person2;
strcpy(person2.name, "Bob");  // Use string copy for arrays
person2.age = 25;
person2.height = 6.0f;
```

**3. Accessing Structure Members**

To access or modify members of a structure variable, use the dot operator (.):

```
printf("Name: %s\n", person1.name);
printf("Age: %d\n", person1.age);
printf("Height: %.1f\n", person1.height);
```

If you have a pointer to a structure, use the arrow operator (->) to access members:

```
struct Person *ptr = &person1;
printf("Name: %s\n", ptr->name);
```

- Use struct to define a new data type grouping related variables.
- Initialize structures either at declaration or by assigning to members.
- Access members with . for variables and -> for pointers.

File Handling in C

• THEORY EXERCISE:

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

File handling in C is crucial because it allows programs to store and retrieve data persistently, beyond the program's runtime. This capability is essential for applications that need to save user data, configuration settings, or large datasets that cannot fit entirely in memory.

In C, file operations are performed using functions from the standard I/O library (<stdio.h>). The key file operations include:

1. Opening a file:
    - Use fopen() to open a file.
    - Syntax: FILE *fopen(const char *filename, const char *mode);
    - Modes include "r" (read), "w" (write), "a" (append), and their binary counterparts "rb", "wb", "ab".
    - Returns a pointer to a FILE object or NULL if the file cannot be opened.
2. Closing a file:
    - Use fclose() to close an open file.
    - Syntax: int fclose(FILE *stream);
    - Returns 0 on success.
3. Reading from a file:
    - Use functions like fgetc(), fgets(), or fread().
    - fgetc(FILE *stream) reads a single character.
    - fgets(char *str, int n, FILE *stream) reads a string up to n-1 characters or newline.
    - fread(void *ptr, size_t size, size_t count, FILE *stream) reads binary data.
4. Writing to a file:
    - Use functions like fputc(), fputs(), or fwrite().
    - fputc(int char, FILE *stream) writes a single character.
    - fputs(const char *str, FILE *stream) writes a string.

- fwrite(const void *ptr, size_t size, size_t count, FILE *stream) writes binary data.

Example:

```
#include <stdio.h>

int main() {
   FILE *fp;

   // Open file for writing
   fp = fopen("example.txt", "w");
   if (fp == NULL) {
      perror("Error opening file");
      return 1;
   }

   // Write to file
   fputs("Hello, file handling in C!\n", fp);

   // Close file
   fclose(fp);

   // Open file for reading
   fp = fopen("example.txt", "r");
   if (fp == NULL) {
      perror("Error opening file");
      return 1;
   }
```

```
    // Read and print file content
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer);
    }

    // Close file
    fclose(fp);

    return 0;
}
```