Module 2 – Introduction to Programming

Overview of C Programming

THEORY EXERCISE: 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, developed in the early 1970s by Dennis Ritchie at Bell Labs, has been a cornerstone in the field of computer science. Its evolution and sustained relevance offer insights into its foundational role in modern computing.

Origins and Evolution

C emerged as an enhancement of the B programming language, which itself was derived from BCPL. Dennis Ritchie introduced C to leverage the capabilities of the PDP-11 computer, adding data types and other features to overcome B's limitations. By 1973, C had evolved sufficiently to allow the Unix operating system kernel to be rewritten in it, marking a significant milestone in computing history.

The publication of "The C Programming Language" by Brian Kernighan and Dennis Ritchie in 1978, often referred to as "K&R C," played a pivotal role in popularizing C. This book served as the de facto standard for the language until formal standardization efforts began.

In 1989, the American National Standards Institute (ANSI) standardized C, resulting in ANSI C, also known as C89. Subsequent revisions, including C99, C11, and C17, introduced features like inline functions, new data types, and improved Unicode support, ensuring that C remained contemporary with evolving programming paradigms.

Importance and Enduring Relevance

C's enduring importance can be attributed to several key factors:

1. **Performance and Efficiency**: C provides low-level memory access and minimal runtime overhead, making it ideal for system programming where performance is critical.
2. **Portability**: Programs written in C can be compiled and run on various platforms with minimal modifications, a feature that was instrumental in the widespread adoption of Unix.
3. **Foundation for Other Languages**: Many modern languages, including C++, C#, and Java, have roots in C, inheriting its syntax and concepts. This makes C a valuable foundation for understanding these languages.
4. **Extensive Use in System Development**: Operating systems like Windows, Linux, and macOS have components implemented in C. Its use extends to embedded systems, databases, and performance-critical applications.
5. **Educational Value**: Learning C instills a strong understanding of fundamental programming concepts such as pointers, memory management, and data structures, which are applicable across various languages and systems.

Despite the emergence of numerous programming languages tailored for specific applications, C's blend of efficiency, portability, and simplicity ensures its continued use in modern software development. Its influence pervades contemporary computing, underscoring its status as a timeless and essential programming language.

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

C programming plays a pivotal role in various domains due to its efficiency, portability, and close-to-hardware capabilities. Here are three real-world applications where C is extensively utilized:

1. **Operating Systems**

   C is fundamental in the development of operating systems. Its low-level access to memory and system processes allows for efficient and effective OS development. Notably, the Unix operating system was originally written in C, and many modern operating systems, including various versions of Windows, Linux, and macOS, have core components developed in C. This usage underscores C's capability to interact directly with hardware components, manage system resources, and ensure high performance and stability

2. **Embedded Systems**

   Embedded systems are specialized computing systems that perform dedicated functions within larger mechanical or electrical systems. C is extensively used in programming these systems due to its ability to provide low-level access to hardware, efficient memory usage, and minimal runtime overhead. Applications range from consumer electronics like smartphones and home appliances to industrial machines and automotive systems. The language's efficiency and control make it ideal for developing firmware and managing hardware resources in embedded applications

3. **Game Development**

   In the realm of game development, performance and speed are critical. C, often in conjunction with C++, is used to develop game engines and high-performance game applications. Its ability to directly manipulate hardware and memory resources allows developers to optimize games for speed and responsiveness. Many renowned game engines and games have been developed using C, highlighting its significance in creating complex gaming systems that require real-time processing and high efficiency.

---

2. Setting Up Environment

THEORY EXERCISE: 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 set up a C programming environment using Dev-C++, follow these steps:

1. Download Dev-C++

- Visit the official Dev-C++ website: bloodshed.net.
- Click on the "Download" link to obtain the latest version of Dev-C++.

2. Install Dev-C++

- Locate the downloaded installer file and double-click to run it.
- Follow the on-screen instructions:
    - Accept the license agreement.
    - Choose the components to install (it's recommended to include all default components).
    - Select the installation directory or proceed with the default path.
- Click "Install" to begin the installation process.

3. Configure Dev-C++

- Launch Dev-C++ after installation.
- To enable debugging:
    - Navigate to the "Tools" menu and select "Compiler Options."
    - In the "Settings" tab, click on "Linker" in the left panel.
    - Set "Generate debugging information" to "Yes."
    - Click "OK" to save the changes.

4. Create a New Project

- Go to the "File" menu and select "New Project."
- Choose "Empty Project" and ensure "C project" is selected.
- Provide a name for your project and click "OK."
- Save the project in your desired directory.

5. Add a Source File

- Navigate to the "File" menu and select "New Source File."
- Write your C code in the editor window that appears.
- Save the file with a `.c` extension (e.g., `main.c`).
- To add existing source files:
    - Go to the "Project" menu and select "Add to Project."
    - Choose the files you want to include.

6. Compile and Run Your Program

- To compile:
    - Navigate to the "Execute" menu and select "Compile" (or press `Ctrl+F9`).
- To run:
    - Go to the "Execute" menu and select "Run" (or press `Ctrl+F10`).

By following these steps, you'll have a functional C development environment using Dev-C++ on your Windows system.

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

---

3. Basic Structure of a C Program

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

A C program consists of several fundamental components:

- **Header Files**: These are included at the beginning of the program using the `#include` directive to provide access to standard library functions.

  #include <stdio.h> // Standard Input/Output functions

- **Main Function**: The entry point of every C program where execution begins.

  Syntax:

  int main(void) {

  // Code statements

  return 0;

  }

- **Comments**: Non-executable statements used to explain code.
  // Single-line comment
- /* hello i am the

  This is a multi-line comment

  */

data Types and Variables: Define the kind of data a variable can hold.

```
 int age = 25;           // Integer variable
float salary = 55000.5; // Floating-point variable
char grade = 'A';       // Character variable
```

EXAMPLE

```
#include <stdio.h>
int main(void) {
    int num1 = 5;
    int num2 = 10;
    int sum;
    // Calculate sum
    sum = num1 + num2;
    // Display result
    printf("The sum of %d and %d is %d\n", num1, num2, sum);
    return 0;
}
```

OUTPUT

```
The sum of 5 and 10 is 15

--------------------------------
Process exited after 1.405 seconds with return value 0
Press any key to continue . . .
```

This program calculates and displays the sum of two integers.

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

---

4. Operators in C

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

In C programming, operators are symbols that instruct the compiler to perform specific mathematical or logical manipulations. Here's a concise overview of the primary operator types:

1. **Arithmetic 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.
    - Modulus (`%`): Returns the remainder of the division of two integers.
2. **Relational Operators**:
    - (`==`): Checks if two operands are equal.
    - (`!=`): Checks if two operands are not equal.
    - (`>`): Checks if the left operand is greater than the right.
    - (`<`): Checks if the left operand is less than the right.
    - (`>=`): Checks if the left operand is greater than or equal to the right.
    - (`<=`): Checks if the left operand is less than or equal to the right.
3. **Logical Operators**:
    - AND (`&&`): Returns true if both operands are true.
    - OR (`||`): Returns true if at least one operand is true.
    - NOT (`!`): Returns true if the operand is false, and vice versa.
4. **Assignment Operators**
    - (`=`): Assigns the right operand's value to the left operand.
    - (`+=`): Adds the right operand to the left operand and assigns the result to the left operand.
    - (`-=`): Subtracts the right operand from the left operand and assigns the result to the left operand.
    - (`*=`): Multiplies the left operand by the right operand and assigns the result to the left operand.

- (`/=`): Divides the left operand by the right operand and assigns the result to the left operand.
- (`%=`): Applies the modulus operator and assigns the result to the left operand.

5. **Increment and Decrement Operators**:
   - (`++`): Increases the value by 1.
   - (`--`): Decreases the value by 1.

   These can be used in two forms:

   - Prefix:. The operation is performed before the value is used in an expression.
   - Postfix: The current value is used in the expression before the operation is performed.
6. **Bitwise Operators**:
   - AND (`&`): Sets each bit to 1 if both corresponding bits are 1.
   - OR (`|`): Sets each bit to 1 if at least one of the corresponding bits is 1.
   - XOR (`^`): Sets each bit to 1 if only one of the corresponding bits is 1.
   - NOT (`~`): Inverts all the bits.
   - Left shift (`<<`): Shifts bits to the left, filling with zeros.
   - Right shift (`>>`): Shifts bits to the right, filling with the sign bit (for signed types) or zeros (for unsigned types).
7. **Conditional (Ternary) Operator**: A shorthand for `if-else` statements.
   - Syntax: `condition ? expression_if_true : expression_if_false;`
   - Example: `int result = (a > b) ? a : b;` // Assigns the greater of `a` or `b` to `result`.

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

---

5. Control Flow Statements in C

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

In C programming, decision-making statements control the flow of execution based on specified conditions. These statements allow the program to choose different paths of execution. The primary decision-making statements in C are:

`if` Statement: Executes a block of code if a specified condition is true.

Syntax:

```
if (condition) {
```

```
    // Code to execute if condition is true
}
```

Example:

```
 int number = 10;
if (number > 0) {
    printf("The number is positive.\n");
}
```

`if-else` Statement: Executes one block of code if a condition is true, and another block if the condition is false.

Syntax:

```
 if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Example:

```
 int number = -5;
if (number >= 0) {
    printf("The number is non-negative.\n");
} else {
    printf("The number is negative.\n");
}
```

Nested `if-else` Statement: An `if` or `else` statement inside another `if` or `else`.

Syntax:

```
 if (condition1) {
    // Code if condition1 is true
    if (condition2) {
        // Code if condition2 is also true
    } else {
        // Code if condition2 is false
    }
} else {
    // Code if condition1 is false
}
```

Example:

```
 int number = 0;
if (number >= 0) {
    if (number == 0) {
        printf("The number is zero.\n");
```

```
    } else {
        printf("The number is positive.\n");
    }
} else {
    printf("The number is negative.\n");
}
```

`switch` Statement: Allows a variable to be tested against a list of values, each with associated code blocks.

Syntax:

```
switch (variable) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    // More cases...
    default:
        // Code if variable doesn't match any case
}
```

Example:

```
char grade = 'B';
switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
    case 'C':
        printf("Well done.\n");
        break;
    case 'D':
        printf("You passed.\n");
        break;
    case 'F':
        printf("Better try again.\n");
        break;
    default:
        printf("Invalid grade.\n");
}
```

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.).

---

6. 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.

1. `while` Loop

Structure: Checks the condition before executing the loop body.

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

2. `for` Loop

Structure: Combines initialization, condition-checking, and iteration in a single line.

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

3. `do-while` Loop

Structure: Executes the loop body first, then checks the condition.

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

Comparison:

- **Condition:**
  - `while` and `for` L*oops:* Evaluate the condition before executing the loop body; if the condition is false initially, the loop body may not execute at all.
  - `do-while` Loop: Evaluates the condition after executing the loop body; thus, the loop body executes at least once.
- **Use Cases:**
  - `while` Loop*:* Used when the number of iterations is unknown, and the loop continues until a specific condition is met.
  - `for` Loop: Preferred when the number of iterations is known or can be determined before entering the loop.

- `do-while` Loop: Ideal when the loop must execute at least once, regardless of the condition

---

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

---

7. Loop Control Statements

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

In C programming, the `break`, `continue`, and `goto` statements are used to control the flow of a program by altering the normal sequence of execution.

1. `break` Statement:

The `break` statement is used to terminate the execution of a loop (like `for`, `while`, or `do-while`) or a `switch` statement prematurely, regardless of the loop's condition.

- **Usage:** When the `break` statement is encountered, it immediately exits the loop or `switch` statement and the program continues execution after the loop or `switch`.

Example of `break`:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // Exits the loop when i equals 5
        }
        printf("%d ", i);
    }
    printf("\nLoop exited\n");
    return 0;
}
```

Output:

1 2 3 4
Loop exited

In this example, the loop is exited when `i` equals 5 due to the `break` statement.

2. `continue` Statement:

The `continue` statement is used to skip the current iteration of a loop and move to the next iteration of the loop. It only affects the current iteration of the loop.

- **Usage:** When the `continue` statement is encountered, the remaining statements in the loop are skipped for the current iteration, and the loop moves to the next iteration.

Example of `continue`:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue;  // Skips even numbers and continues to next iteration
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

1 3 5 7 9

In this example, the `continue` statement skips the even numbers (i.e., 2, 4, 6, 8, 10) and only prints the odd numbers.

3. `goto` Statement:

The `goto` statement is used to transfer control unconditionally to another part of the program. It can jump to any part of the program where a label is defined.

- **Usage:** You need to define a label and then use `goto` to jump to that label in the program. However, the use of `goto` is generally discouraged because it makes the code less readable and maintainable. It's often better to use structured control flow like loops and functions.

Example of `goto`:

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

```
int main() {
    int i = 0;
    start:  // Label definition
    if (i < 5) {
        printf("%d ", i);
        i++;
        goto start;  // Jumps back to the 'start' label
    }

    printf("\nEnd of program.\n");
    return 0;
}
```

Output:

```
0 1 2 3 4
End of program.
```

In this example, the program jumps back to the `start` label until `i` reaches 5, causing the numbers 0 to 4 to be printed.

Key Differences:

- **`break`**: Exits the loop or switch statement.
- **`continue`**: Skips the current iteration of a loop and continues with the next iteration.
- **`goto`**: Unconditionally jumps to a specified label, which can lead to less structured code.

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

---

8. Functions in C

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

A **function** in C is a block of code designed to perform a specific task. Functions help make your code more organized, reusable, and easier to maintain.

1. Function Declaration (Prototype)

A function declaration provides the compiler with the function's name, return type, and parameters. It doesn't define what the function does, just what it should look like.

Syntax:

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

Example:

int add(int, int); // Declares the function 'add' with two int parameters

2. Function Definition

The function definition contains the actual code that performs the task.

Syntax:

```
return_type function_name(parameter_type1 parameter1, parameter_type2
parameter2, ...) {
    // Code to execute
}
```

Example:

```
int add(int a, int b) {
    return a + b;  // Adds two integers and returns the result
}
```

3. Function Call

A function call is where you actually use the function to perform its task.

Syntax:

function_name(argument1, argument2, ...);

Example:

int result = add(5, 3); // Calls the 'add' function with 5 and 3

Complete Example:

```c
#include <stdio.h>

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

int main() {
    int num1 = 5, num2 = 3;
    int result = add(num1, num2);  // Function Call
    printf("Result: %d\n", result);
    return 0;
}

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

Types of Functions:

Void Function: Doesn't return any value.

```c
 void greet() {
    printf("Hello!");
}
```

-

Non-Void Function: Returns a value.

```c
 int multiply(int a, int b) {
    return a * b;
}
```

Functions are essential for making your C programs modular, reusable, and easier to maintain.

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

---

9. Arrays in C

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

An **array** in C is a collection of variables of the same type that are stored in contiguous memory locations. Arrays allow you to store multiple values in a single variable, making it easier to manage and manipulate data.

Basic Concept:

- An array is defined by specifying the type of the elements, followed by the array name and size.

Syntax:
type array_name[size];
Example:
int numbers[5]; // Array of 5 integers

Types of Arrays

1. **One-Dimensional Array (1D Array)**:
    - A **one-dimensional array** is simply a list of elements that can be accessed using a single index.
    - It's like a row of values where each element has its own index.

Syntax:
type array_name[size];

2. **Multi-Dimensional Array**:
    - A **multi-dimensional array** is an array of arrays, where each element is an array itself. It can be two-dimensional, three-dimensional, or even higher, though typically, two-dimensional (2D) arrays are the most common.
    - A **two-dimensional array** (2D array) can be thought of as a table or grid with rows and columns.

Syntax:
type array_name[rows][columns];

Differences Between One-Dimensional and Multi-Dimensional Arrays

Aspect	One-Dimensional Array (1D)	Multi-Dimensional Array (2D, 3D, etc.)

Structure	A simple list of elements	Array of arrays (like a table or grid)
Access	Accessed by a single index	Accessed by multiple indices (e.g., 2 indices for 2D)
Example	`int numbers[5] = {1, 2, 3, 4, 5};`	`int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};`
Memory Representation	Stored in a single continuous block	Elements are stored as rows, each row being a 1D array
Use Case	Suitable for linear data	Suitable for grid-like or table data

Examples:

1. One-Dimensional Array (1D Array):

```
#include <stdio.h>
int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    printf("%d\n", numbers[0]);
    printf("%d\n", numbers[4]);
    return 0;
}
```

2. Two-Dimensional Array (2D Array):

```
#include <stdio.h>

int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    printf("%d\n", matrix[0][1]);  // Output: 2 (element at row 0, column 1)
    printf("%d\n", matrix[1][2]);  // Output: 6 (element at row 1, column 2)
    return 0;
}
```

LAB EXERCISE: 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.

---

10. Pointers in C

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

A **pointer** in C is a variable that stores the **memory address** of another variable. Instead of storing a value directly, a pointer stores the location of the value in memory.

Why Are Pointers Important in C?

1. **Direct Memory Access**: Pointers allow you to access and manipulate memory directly, providing greater control over the system's resources.
2. **Efficiency**: When passing large structures or arrays to functions, pointers allow you to pass references (addresses) rather than copying the entire data, improving performance.
3. **Dynamic Memory Allocation**: Pointers are essential for dynamically allocating and freeing memory at runtime using functions like `malloc()`, `calloc()`, and `free()`.
4. **Linked Data Structures**: Pointers are used to create dynamic data structures like linked lists, trees, and graphs.
5. **Function Pointers**: Pointers to functions allow you to call functions dynamically at runtime, providing flexibility in your program.

How Pointers Are Declared and Initialized

1. Pointer Declaration

A pointer is declared by specifying the type of data the pointer will point to, followed by an asterisk (`*`), which indicates that it's a pointer.

Syntax:

```
type *pointer_name;
```

2. Pointer Initialization

A pointer is initialized by assigning it the address of a variable using the **address-of operator (`&`)**. This operator gives the memory address of a variable.

Syntax:

```
pointer_name = &variable_name;
```

3. Dereferencing a Pointer

Once a pointer is initialized with a memory address, you can access the value stored at that address using the **dereference operator (*)**. Dereferencing allows you to get or modify the value the pointer is pointing to.

Syntax:

*pointer_name

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

---

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.

In C programming, string handling functions are essential for performing various operations on strings, such as determining their length, copying, concatenating, comparing, and searching within them. These functions are declared in the `<string.h>` header file. **`strlen()`**: Calculates the length of a string (excluding the null terminator `'\0'`).

Syntax:

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

Example:

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

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

`strcpy()`: Copies the source string into the destination string.

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);  // Copies src to dest
    printf("Destination string after strcpy: %s\n", dest);  // Output: Hello
    return 0;
}
```

`strcat()`: Appends the source string to the end of the destination string.

Syntax:

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

Example:

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

int main() {
    char str1[20] = "Hello";
    char str2[] = " World";

    strcat(str1, str2);  // Appends str2 to str1
    printf("Concatenated string: %s\n", str1);  // Output: Hello World
    return 0;
}
```

`strcmp()`: Compares two strings lexicographically.

Syntax:

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

Example:

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

int main() {
    char str1[] = "Hello";
    char str2[] = "World";

    int result = strcmp(str1, str2);
    if (result == 0) {
        printf("Strings are equal.\n");
    } else if (result < 0) {
        printf("str1 is less than str2.\n");
    } else {
        printf("str1 is greater than str2.\n");
    }
    return 0;
}
```

`strchr()`: Finds the first occurrence of a character in a string.

Syntax:

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

Example:

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

int main() {
    char str[] = "Hello, World!";
    char *result = strchr(str, 'W');

    if (result != NULL) {
        printf("Character found: %s\n", result);  // Output: World!
    } else {
        printf("Character not found.\n");
    }
    return 0;
}
```

LAB EXERCISE: 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().

---

12. Structures in C

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

In C programming, a **structure** (`struct`) is a user-defined data type that allows grouping variables of different types under a single name. This feature is particularly useful for modeling complex entities by bundling related information together.

Declaring a Structure

To declare a structure, use the `struct` keyword followed by the structure name and a block containing the member declarations.

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

In this example, `Person` is a structure with three members: `name`, `age`, and `salary`.

Creating Structure Variables

After declaring a structure, you can create variables of that type in two ways:

Direct Declaration:

```
struct Person person1, person2;
```

1.

Using `typedef` for Simplification:

```
typedef struct {
    char name[50];
```

```
    int age;
    float salary;
} Person;

Person person1, person2;
```

2. Here, `typedef` creates an alias `Person` for the structure, eliminating the need to use the `struct` keyword when declaring variables.

Initializing Structure Members

Structure members can be initialized at the time of declaration or assigned values individually.

Designated Initializers (C99 Standard):

```
Person person1 = { .name = "Alice", .age = 30, .salary = 50000.0f };
```

1. This method allows initializing specific members by name, and members not explicitly initialized are set to zero or null.

Positional Initialization:

```
Person person2 = { "Bob", 25, 45000.0f };
```

2. Values are assigned in the order the members are declared.

Individual Assignment:

```
Person person3;
strcpy(person3.name, "Charlie");
person3.age = 28;
person3.salary = 47000.0f;
```

3. Members are assigned values separately after the variable is declared.

Accessing Structure Members

To access or modify structure members, use the dot (`.`) operator with the structure variable.

```
printf("Name: %s\n", person1.name);
```

person2.age = 26;

Pointers to Structures

Working with pointers to structures is common, especially when passing structures to functions. Use the arrow (`->`) operator to access members via a pointer.

```
Person *ptr = &person1;
printf("Salary: %.2f\n", ptr->salary);
ptr->age = 31;
```

Example Usage

Here's a complete example demonstrating structure declaration, initialization, and member access:

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

typedef struct {
    char name[50];
    int age;
    float salary;
} Person;

int main() {
    Person person1 = { .name = "Alice", .age = 30, .salary = 50000.0f };
    Person person2;
    strcpy(person2.name, "Bob");
    person2.age = 25;
    person2.salary = 45000.0f;

    printf("Person 1: %s, Age: %d, Salary: %.2f\n", person1.name, person1.age,
person1.salary);
    printf("Person 2: %s, Age: %d, Salary: %.2f\n", person2.name, person2.age,
person2.salary);

    return 0;
}
```

This program defines a `Person` structure, initializes two variables, and prints their details.

Structures in C provide a way to group related variables, enabling the creation of complex data types that model real-world entities effectively.

LAB EXERCISE: 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.

---

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.

File handling in C is crucial for managing data storage and retrieval, enabling programs to persist data beyond their execution time. By interacting with files, programs can read input, store output, and maintain data between runs, which is essential for applications like data logging, configuration management, and user data storage.

Performing File Operations in C

The C Standard Library provides a set of functions for file operations, including opening, closing, reading, and writing files. These functions are declared in the `<stdio.h>` header file.

Opening a File

To open a file, use the `fopen()` function, which returns a pointer to a `FILE` object. This pointer is then used for subsequent file operations.

Syntax:

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

1.
    - `filename`: Name (and path) of the file to open.
    - `mode`: Specifies the operation type (e.g., reading, writing).
2. Common modes include:

    - `'r'`: Open for reading. Fails if the file doesn't exist.
    - `'w'`: Open for writing. Creates a new file or truncates an existing file.

- `'a'`: Open for appending. Creates a new file if it doesn't exist.
- `'r+'`: Open for reading and writing. Fails if the file doesn't exist.
- `'w+'`: Open for reading and writing. Creates a new file or truncates an existing file.
- `'a+'`: Open for reading and appending. Creates a new file if it doesn't exist.

Example:

```
 FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    perror("Error opening file");
    // Handle error
}
```

3. **Closing a File**

   After completing file operations, close the file using `fclose()` to free resources and ensure data integrity.

   Syntax:

   ```
   int fclose(FILE *stream);
   ```

4. `stream`: Pointer to the `FILE` object to close.

Example:

```
 if (fclose(file) != 0) {
    perror("Error closing file");
    // Handle error
}
```

5. **Reading from a File**

   To read data, functions like `fgetc()`, `fgets()`, or `fread()` can be used, depending on the requirement.

`fgetc()`: Reads a single character.

```
 int ch = fgetc(file);
if (ch == EOF) {
    // Handle end-of-file or error
}
```

`fgets()`: Reads a line of text.

```
 char buffer[100];
if (fgets(buffer, sizeof(buffer), file) != NULL) {
    // Process the line
} else {
    // Handle end-of-file or error
}
```

`fread()`: Reads binary data.

```
 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `ptr`: Pointer to a block of memory where the read data will be stored.
- `size`: Size of each element to read.
- `nmemb`: Number of elements to read.
- `stream`: Pointer to the `FILE` object.

Example:

```
 size_t bytesRead = fread(buffer, sizeof(char), sizeof(buffer), file);
if (bytesRead < sizeof(buffer)) {
    if (feof(file)) {
        // End of file reached
    } else if (ferror(file)) {
        // Handle error
    }
}
```

6. **Writing to a File**

To write data, functions like `fputc()`, `fputs()`, or `fwrite()` are used.

`fputc()`: Writes a single character.

```
 int fputc(int char, FILE *stream);
```

- `char`: The character to write.
- `stream`: Pointer to the `FILE` object.

Example:

```
 if (fputc('A', file) == EOF) {
```

```
    // Handle error
}
```

`fputs()`: Writes a string.

```
int fputs(const char *str, FILE *stream);
```

- `str`: Pointer to the null-terminated string to write.
- `stream`: Pointer to the `FILE` object.

Example:

```
if (fputs("Hello, World!\n", file) == EOF) {
    // Handle error
}
```

`fwrite()`: Writes binary data.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `ptr`: Pointer to the data to write.
- `size`: Size of each element to write.

LAB EXERCISE: 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.
