Module 1 : Introduction to C

Learning Objectives

- Understand the fundamental differences between Python and C programming languages
- Implement basic input/output operations in C
- Declare and use variables with appropriate data types
- Apply arithmetic operators in C expressions
- Design and implement flow control structures (conditionals and loops)
- Transition effectively from Python syntax to C syntax

1. Introduction: From Python to C

1.1 Key Differences Overview

Aspect	Python	C
Compilation	Interpreted	Compiled
Type System	Dynamic typing	Static typing
Memory Management	Automatic	Manual
Syntax Style	Indentation-based	Brace-based
Performance	Slower execution	Faster execution
Development Speed	Faster to write	More verbose

Some of you might ask, What does it mean by `Static Typing` vs `Dynamic Typing` ?

Dynamic Typing (Python):

- Variables can change their data type during program execution
- Type checking happens at runtime
- No need to declare variable types explicitly
- More flexible but can lead to runtime errors

```
# Python - Dynamic Typing
x = 5        # x is an integer
x = "Hello"  # Now x is a string (allowed!)
x = 3.14     # Now x is a float (also allowed!)
```

Static Typing (C):

- Variables must be declared with a specific data type
- Type checking happens at compile time
- Once declared, a variable cannot change its type
- Less flexible but catches type errors before program runs

```
// C - Static Typing
int x = 5;           // x is declared as integer
x = "Hello";         // ERROR! Cannot assign string to integer variable
```

```
float y = 3.14f;     // y must be declared as float to store decimal numbers
```

Advantages of Static Typing (C):

- Errors caught during compilation, not during execution
- Better performance (no runtime type checking needed)
- More predictable memory usage
- Clearer code documentation (types are explicit)

Advantages of Dynamic Typing (Python):

- Faster prototyping and development
- More flexible for rapid changes
- Less verbose code
- Easier for beginners to learn

1.2 Basic Program Structure

Python:

```
# Simple Python program
print("Hello, World!")
```

C:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Key Points:

- C requires explicit inclusion of header files (such as `#include <stdio.h>`)
- Every C program must have a `main()` function
- C statements end with semicolons (`;`)
- C uses curly braces `{}` to define code blocks
- Functions must explicitly return a value except for void function (we will learn more about this on the next module)

2. Input/Output Operations

2.1 Output Operations

2.1.1 Basic Output - printf()

Function Signature:

```
int printf(const char *format, ...);
```

Python vs C Comparison:

Python	C
`print("Hello")`	`printf("Hello\n");`
`print("Value:", x)`	`printf("Value: %d\n", x);`
`print(f"x = {x}")`	`printf("x = %d\n", x);`

Keep in mind that `print()` in python automatically creates a new line by default

2.1.2 Format Specifiers

Data Type	Format Specifier	Example
int	`%d` or `%i`	`printf("%d", 42);`
float	`%f`	`printf("%.2f", 3.14);`
double	`%lf`	`printf("%.2lf", 3.14159);`
char	`%c`	`printf("%c", 'A');`
string	`%s`	`printf("%s", "Hello");`
hexadecimal	`%x` or `%X`	`printf("%x", 255);`
unsigned int	`%u`	`printf("%u", 42u);`

2.1.3 Advanced printf() Features

Width and Precision:

```
printf("%5d", 42);        // Right-aligned in 5 characters: "   42"
printf("%-5d", 42);       // Left-aligned in 5 characters: "42   "
printf("%05d", 42);       // Zero-padded: "00042"
printf("%.2f", 3.14159);  // 2 decimal places: "3.14"
```

```
printf("%8.2f", 3.14159); // 8 characters, 2 decimals: "    3.14"
```

2.1.4 Escape Characters

Escape characters are special character sequences that represent characters that are difficult or impossible to type directly. They start with a backslash (`\`).

Common Escape Characters:

Escape Sequence	Character	Description	Example
`\n`	Newline	Moves cursor to next line	`printf("Line 1\nLine 2");`
`\t`	Tab	Horizontal tab (8 spaces)	`printf("Name:\tJohn");`
`\"`	Double Quote	Literal double quote	`printf("He said \"Hello\"");`
`\'`	Single Quote	Literal single quote	`printf("It\'s working");`
`\\`	Backslash	Literal backslash	`printf("Path: C:\\Program Files");`
`\r`	Carriage Return	Return to beginning of line	`printf("Loading\rDone");`
`\b`	Backspace	Move cursor back one position	`printf("ABC\bD");` → "ABD"
`\0`	Null Character	String terminator	`char str[] = "Hi\0lo";`
`\a`	Alert (Bell)	System beep/alert sound	`printf("\aError!");`
`\f`	Form Feed	Page break	`printf("Page 1\fPage 2");`
`\v`	Vertical Tab	Vertical tab	`printf("Line 1\vLine 2");`

Python vs C Escape Characters:

Purpose	Python	C
New line	`print("Line 1\nLine 2")`	`printf("Line 1\nLine 2");`
Tab spacing	`print("Name:\tAge")`	`printf("Name:\tAge");`
Quote in string	`print("She said \"Hi\"")`	`printf("She said \"Hi\"");`
Backslash	`print("C:\\folder")`	`printf("C:\\\\folder");`

Practical Examples:

```
// Creating formatted output with escape characters
printf("Student Information:\n");
printf("Name:\t\tJohn Doe\n");
printf("Age:\t\t20\n");
printf("GPA:\t\t3.75\n");
```

```
// Output:
// Student Information:
// Name:□□John Doe
// Age:□□20
// GPA:□□3.75
```

```
// Using quotes within strings
printf("The teacher said, \"Programming is fun!\"\n");
// Output: The teacher said, "Programming is fun!"

// File paths (especially important for Windows)
printf("Save file to: C:\\Documents\\Programs\\myfile.txt\n");
// Output: Save file to: C:\Documents\Programs\myfile.txt
```

Important Notes:

- In C strings, `\0` (null character) automatically terminates the string
- When counting string length, `\n`, `\t`, etc. each count as ONE character
- Escape characters work in both printf() format strings and character/string literals

2.2 Input Operations

2.2.1 Basic Input - scanf()

Function Signature:

```
int scanf(const char *format, ...);
```

Python vs C Comparison:

Python	C
`x = int(input())`	`scanf("%d", &x);`
`name = input()`	`scanf("%s", name);`
`x = float(input())`	`scanf("%f", &x);`

2.2.2 Important scanf() Considerations

Address Operator (&):

- Most variables need `&` before the variable name
- Exception: strings (character arrays) don't need `&`

```
int age;
char name[50];
float height;

scanf("%d", &age);     // & required for int
scanf("%s", name);     // & NOT needed for string
scanf("%f", &height);  // & required for float
```

Input Buffer Issues and Whitespace Handling:

The Whitespace Problem: When you press Enter after typing input, scanf() reads the data but leaves the newline character (`\n`) in the input buffer. This can cause problems with subsequent input operations.

```
// Problematic code:
int num;
char ch;

printf("Enter a number: ");
scanf("%d", &num);              // User types "5" and presses Enter
                                // Buffer now contains: \n (leftover newline)
printf("Enter a character: ");
scanf("%c", &ch);               // This reads the leftover \n, not user input!
printf("Character: %c\n", ch);
// Outputs:
// Character: (it shows nothing because it prints a newline)
```

What happens step by step:

1. User types "5" and presses Enter → Input buffer: `5\n`
2. `scanf("%d", &num)` reads "5" → Buffer remaining: `\n`
3. `scanf("%c", &ch)` immediately reads the leftover `\n`
4. Program doesn't wait for new character input

Solutions:

Solution 1: Space before %c

```
int num;
char ch;

printf("Enter a number: ");
```

```
scanf("%d", &num);
printf("Enter a character: ");
scanf(" %c", &ch);  // Space before %c consumes all whitespace (spaces, tabs, newlines)
```

Solution 2: Explicit buffer clearing

```
int num;
char ch;

printf("Enter a number: ");
scanf("%d", &num);

// Clear the input buffer
while (getchar() != '\n');  // Read and discard until newline

printf("Enter a character: ");
scanf("%c", &ch);
```

Solution 3: Using getchar() to consume newline

```
int num;
char ch;

printf("Enter a number: ");
scanf("%d", &num);
getchar();  // Consume the leftover newline

printf("Enter a character: ");
scanf("%c", &ch);
```

Whitespace Characters in C:

- `\n` (newline) - ASCII 10
- `\t` (tab) - ASCII 9
- `\r` (carriage return) - ASCII 13
- ` ` (space) - ASCII 32
- `\f` (form feed) - ASCII 12
- `\v` (vertical tab) - ASCII 11

Important scanf() Whitespace Rules:

- `%d`, `%f`, `%s` automatically skip leading whitespace
- `%c` does NOT skip whitespace (reads exactly one character)

- Adding a space in format string (`%c`) makes scanf() skip whitespace
- `%[^\n]` does not skip leading whitespace but stops at newline

Advanced scanf() Format Specifiers:

1. Character Set Specifiers `[...]`:

```
char name[50];

// Read only alphabetic characters
scanf("%[a-zA-Z]", name);

// Read everything except newline
scanf("%[^\n]", name);  // Reads entire line including spaces

// Read only digits
scanf("%[0-9]", name);

// Read only vowels
scanf("%[aeiouAEIOU]", name);
```

2. Excluding Character Sets `[^...]`:

```
char input[100];

// Read everything EXCEPT newline (gets full line with spaces)
scanf("%[^\n]", input);

// Read everything EXCEPT spaces and tabs
scanf("%[^ \t]", input);

// Read everything EXCEPT digits
scanf("%[^0-9]", input);

// Read until comma is encountered
scanf("%[^,]", input);
```

3. Width Specifiers:

```
char buffer[10];
```

```
// Read maximum 9 characters (leaving room for \0)
scanf("%9s", buffer);

// Read exactly 5 characters
scanf("%5c", buffer);
```

4. Practical Examples:

```
// Example 1: Reading full name (including spaces)
char full_name[100];
printf("Enter your full name: ");
scanf(" %[^\n]", full_name);  // Space before % consumes previous newline

// Example 2: Reading until specific delimiter
char email[50];
printf("Enter email: ");
scanf("%[^@]", email);  // Read until @ symbol

// Example 3: Input validation
char grade[10];
printf("Enter grade (A, B, C, D, F): ");
scanf("%[ABCDFabcdf]", grade);  // Only accept valid grades
```

5. Combining Multiple Inputs:

```
int day, month, year;
char separator;

// Reading date in format: dd/mm/yyyy or dd-mm-yyyy
printf("Enter date (dd/mm/yyyy or dd-mm-yyyy): ");
scanf("%d%c%d%c%d", &day, &separator, &month, &separator, &year);

// Alternative: reading with specific separators
printf("Enter date (dd/mm/yyyy): ");
scanf("%d/%d/%d", &day, &month, &year);
```

2.2.3 Alternative Input Methods

getchar() and putchar():

```
char ch;
ch = getchar();  // Read single character
putchar(ch);     // Output single character
```

fgets() for Safe String Input:

```
char name[50];
printf("Enter your name: ");
fgets(name, sizeof(name), stdin);
```

3. Variables and Data Types

3.1 Variable Declaration

Python vs C:

Python	C
`x = 5`	`int x = 5;`
`name = "John"`	`char name[] = "John";`
`pi = 3.14`	`float pi = 3.14f;`

3.2 Basic Data Types

3.2.1 Integer Types

Type	Size (bytes)	Range	Usage
`char`	1	-128 to 127	Small integers, characters
`short`	2	-32,768 to 32,767	Small integers
`int`	4	-2,147,483,648 to 2,147,483,647	Standard integers
`long`	4/8	System dependent	Large integers
`long long`	8	Very large range	Very large integers

Unsigned Variants:

```
unsigned char uc;    // 0 to 255
unsigned int ui;     // 0 to 4,294,967,295
unsigned short us;   // 0 to 65,535
```

3.2.2 Floating-Point Types

Type	Size	Precision	Range
`float`	4 bytes	~7 digits	±3.4 × 10^±38
`double`	8 bytes	~15 digits	±1.7 × 10^±308
`long double`	12/16 bytes	Extended precision	System dependent

3.2.3 Character and String Types

Single Characters:

```
char letter = 'A';        // Single character
char digit = '5';         // Character representation of digit
char newline = '\n';      // Escape sequence
```

Strings (Character Arrays):

```
char name[20] = "John";           // Fixed-size array
char message[] = "Hello World";   // Size determined by initializer
char buffer[100];                 // Uninitialized array
```

Python vs C String Comparison:

Python	C
`name = "John"`	`char name[] = "John";`
`len(name)`	`strlen(name)`
`name[0]`	`name[0]`
`name + " Doe"`	`strcat(name, " Doe");`

3.3 Variable Declaration Rules

1. **Must declare before use** (unlike Python)
2. **Case-sensitive** (`age` ≠ `Age`)
3. **Cannot start with digits** (`2x` is invalid)
4. **Cannot use keywords** (`int`, `if`, `while`, etc.)
5. **Should use meaningful names** (`student_count` not `sc`)

3.4 Constants

```
// Method 1: #define preprocessor directive
#define PI 3.14159
#define MAX_SIZE 100

// Method 2: const keyword
const int ARRAY_SIZE = 50;
const float GRAVITY = 9.81f;
```

Python vs C Constants:

Python	C
`PI = 3.14159`	`#define PI 3.14159`
`PI = 3.14159`	`const float PI = 3.14159f;`

4. Arithmetic Operators

4.1 Basic Arithmetic Operators

Operator	Operation	Python Example	C Example
`+`	Addition	`a + b`	`a + b`
`-`	Subtraction	`a - b`	`a - b`
`*`	Multiplication	`a * b`	`a * b`
`/`	Division	`a / b`	`a / b`
`%`	Modulus	`a % b`	`a % b`

4.2 Important Division Differences

Integer Division:

```
# Python 3
print(7 / 3)   # Output: 2.333...
print(7 // 3)  # Output: 2 (floor division)
```

```
// C
printf("%f\n", 7.0 / 3.0);  // Output: 2.333333
printf("%d\n", 7 / 3);      // Output: 2 (integer division)
printf("%f\n", (float)7 / 3); // Output: 2.333333 (type casting)
```

4.3 Assignment Operators

Operator	Equivalent	Python	C
`=`	Basic assignment	`x = 5`	`x = 5;`
`+=`	Add and assign	`x += 3`	`x += 3;`
`-=`	Subtract and assign	`x -= 3`	`x -= 3;`
`*=`	Multiply and assign	`x *= 3`	`x *= 3;`
`/=`	Divide and assign	`x /= 3`	`x /= 3;`
`%=`	Modulus and assign	`x %= 3`	`x %= 3;`

4.4 Increment and Decrement Operators

Pre-increment vs Post-increment:

```
int x = 5;
int y, z;

y = ++x;  // Pre-increment: x becomes 6, then y = 6
z = x++;  // Post-increment: z = 6, then x becomes 7

// Equivalent operations:
x = x + 1;  // Same as x++ or ++x when used alone
x += 1;     // Same as above
```

Python vs C:

Python	C
`x += 1`	`x++` or `++x` or `x += 1`
`x -= 1`	`x--` or `--x` or `x -= 1`

4.5 Operator Precedence

Priority	Operators	Associativity
1 (Highest)	`++`, `--` (postfix)	Left to right
2	`++`, `--` (prefix), `+`, `-` (unary)	Right to left
3	`*`, `/`, `%`	Left to right
4	`+`, `-` (binary)	Left to right
5 (Lowest)	`=`, `+=`, `-=`, etc.	Right to left

5. Flow Control

5.1 Conditional Statements

5.1.1 if Statement

Python vs C Syntax:

Python:

```
if condition:
    statement1
    statement2
elif another_condition:
    statement3
else:
    statement4
```

C:

```
if (condition) {
    statement1;
    statement2;
} else if (another_condition) {
    statement3;
} else {
    statement4;
}
```

Key Differences:

- C requires parentheses around conditions
- C uses curly braces `{}` instead of indentation
- C requires semicolons after statements

5.1.2 Relational Operators

Operator	Meaning	Python	C
`==`	Equal to	`a == b`	`a == b`

Operator	Meaning	Python	C
`!=`	Not equal to	`a != b`	`a != b`
`<`	Less than	`a < b`	`a < b`
`>`	Greater than	`a > b`	`a > b`
`<=`	Less than or equal	`a <= b`	`a <= b`
`>=`	Greater than or equal	`a >= b`	`a >= b`

5.1.3 Logical Operators

Operator	Meaning	Python	C
`&&`	Logical AND	`and` or `&`	`&&`
`||`	Logical OR	`or` or `|`	`||`
`!`	Logical NOT	`not` or `~`	`!`

Examples in C :

```
// Python: if age >= 18 and score > 80:
if (age >= 18 && score > 80) {
    printf("Eligible for scholarship\n");
}

// Python: if not (x < 0 or x > 100):
if (!(x < 0 || x > 100)) {
    printf("Valid percentage\n");
}
```

5.1.4 Executing Code in `if` Conditions

While primarily used for conditions, C allows expressions that evaluate to a non-zero value (true) or zero (false) within the parentheses. This means you can sometimes perform assignments or function calls directly within the condition, though it's often discouraged for readability.

```
int x = 10;
if (x = 5) { // Assigns 5 to x, then evaluates to 5 (true)
    printf("x is now 5 and this code runs.\n");
}
```

5.1.5 Single Statement `if`:

If an `if` or `else` block contains only a single statement, the curly braces `{}` are optional. However, it's good practice to always use them to avoid ambiguity and potential errors when adding more statements later.

```
if (score > 90)
    printf("Excellent!\n");
else
    printf("Keep trying.\n");
```

5.1.6 switch Statement

C provides `switch` as an alternative to multiple `if-else` statements:

```
switch (variable) {
    case value1:
        // statements
        break;
    case value2:
        // statements
        break;
    default:
        // statements
        break;
}
```

Example:

```
int grade;
printf("Enter grade (1-5): ");
scanf("%d", &grade);

switch (grade) {
    case 5:
        printf("Excellent!\n");
        break;
    case 4:
        printf("Very Good!\n");
        break;
    case 3:
        printf("Good!\n");
        break;
```

```
    case 2:
        printf("Fair!\n");
        break;
    case 1:
        printf("Poor!\n");
        break;
    default:
        printf("Invalid grade!\n");
        break;
}
```

Understanding `break` and Fall-through: In C's `switch` statement, the `break` keyword is essential. If `break` is omitted from a `case` block, execution will "fall through" to the next `case` block (and subsequent ones) until a `break` is encountered or the end of the `switch` statement is reached. This "fall-through" behavior can be intentionally used for specific logic, but it's a common source of bugs if not intended.

Example of Fall-through:

```
int day = 2; // Monday
switch (day) {
    case 1:
        printf("Weekend!\n");
        break;
    case 2:
    case 3:
    case 4:
    case 5:
        printf("Weekday.\n"); // Execution falls through from case 2, 3, 4 to 5
        break;
    case 6:
        printf("Weekend!\n");
        break;
    default:
        printf("Invalid day.\n");
        break;
}
// Output for day = 2: Weekday.
```

This example shows how `case 2`, `case 3`, `case 4`, and `case 5` all execute the same `printf("Weekday.\n");` because there are no `break` statements between them.

5.2 Iteration Statements (Loops)

5.2.1 for Loop

Python vs C Syntax:

Python:

```
for i in range(5):
    print(i)

for i in range(1, 10, 2):
    print(i)
```

C:

```
// Basic for loop
for (int i = 0; i < 5; i++) {
    printf("%d\n", i);
}

// Step by 2
for (int i = 1; i < 10; i += 2) {
    printf("%d\n", i);
}
```

For Loop Structure:

```
for (initialization; condition; increment/decrement) {
    // loop body
}
```

5.2.2 while Loop

Python vs C:

Python:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

C:

```
int i = 0;
while (i < 5) {
    printf("%d\n", i);
    i++;
}
```

5.2.3 do-while Loop

C provides `do-while` loop (not available in Python):

```
int choice;
do {
    printf("Enter choice (1-3): ");
    scanf("%d", &choice);

    if (choice < 1 || choice > 3) {
        printf("Invalid choice! Try again.\n");
    }
} while (choice < 1 || choice > 3);
```

Key Difference: `do-while` executes the loop body at least once, even if the condition is initially false.

5.2.4 Loop Control Statements

Statement	Python	C	Purpose
`break`	`break`	`break;`	Exit loop immediately
`continue`	`continue`	`continue;`	Skip to next iteration

Example:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    if (i > 7) {
        break;     // Stop when i > 7
    }
    printf("%d ", i);  // Output: 1 3 5 7
```

```
}
```

6. More Migration Guide: From Python to C

6.1 Common Syntax Differences

Feature	Python	C
Comments	`# This is a comment`	`// This is a comment`
Block Comments	`"""Multi-line"""`	`/* Multi-line */`
Code Blocks	Indentation	`{ }` braces
Statement End	Line break	`;` semicolon
Boolean Values	`True`, `False`	`1`, `0` (or `true`, `false` with `<stdbool.h>`)

6.2 Variable Declaration Migration

Python to C Translation Examples:

```
# Python
age = 25
height = 5.9
name = "Alice"
is_student = True
```

```
// C
int age = 25;
float height = 5.9f;
char name[] = "Alice";
int is_student = 1;  // or use bool with #include <stdbool.h>
```

6.3 Function Definition Migration (We will learn more about this on the next module)

Python:

```
def calculate_area(length, width):
    return length * width

result = calculate_area(5, 3)
print(result)
```

C:

```
#include <stdio.h>

// Function declaration (prototype)
int calculate_area(int length, int width);

int main() {
    int result = calculate_area(5, 3);
    printf("%d\n", result);
    return 0;
}

// Function definition
int calculate_area(int length, int width) {
    return length * width;
}
```

6.4 Common Pitfalls for Python Programmers

1. **Forgetting Semicolons:**

   ```
   int x = 5  // ERROR: Missing semicolon
   int x = 5; // CORRECT
   ```

2. **Using = instead of == in conditions:**

   ```
   if (x = 5) { ... }   // ERROR: Assignment, not comparison
   if (x == 5) { ... }  // CORRECT: Comparison
   ```

3. **Forgetting & in scanf():**

   ```
   scanf("%d", x);   // ERROR: Missing &
   scanf("%d", &x);  // CORRECT
   ```

4. **Array Index Out of Bounds: (We will learn more about this on the next module)**

```
int arr[5];
arr[5] = 10;  // ERROR: Index 5 is out of bounds (valid: 0-4)
arr[4] = 10;  // CORRECT: Last valid index
```

7. Best Basic Practices and Style Guidelines

7.1 Naming Conventions

- **Variables:** Use descriptive names (`student_count`, not `sc`)
- **Constants:** Use uppercase (`MAX_SIZE`, `PI`)
- **Functions (we will learn more about this in the next module):** Use verb-noun pattern (`calculate_area`, `print_result`)

7.2 Code Organization

```
#include <stdio.h>        // System headers
#include <stdlib.h>

#define MAX_SIZE 100      // Constants

// Function prototypes (We will learn more about this in the next module)
int add(int a, int b);
void print_result(int result);

int main() {
    // Main program logic
    return 0;
}

// Function definitions (We will learn more about this in the next module)
int add(int a, int b) {
    return a + b;
}
```

7.3 Error Handling

Input Validation:

```
int num;
printf("Enter a positive number: ");
scanf("%d", &num)

if (num < 0) {
    printf("Invalid input!\n");
}
```

8. Practical Examples

8.1 Complete Program Examples

Example 1: Simple Calculator

```
#include <stdio.h>

int main() {
    float num1, num2, result;
    char operator;

    printf("Enter first number: ");
    scanf("%f", &num1);

    printf("Enter operator (+, -, *, /): ");
    scanf(" %c", &operator);

    printf("Enter second number: ");
    scanf("%f", &num2);

    switch (operator) {
        case '+':
            result = num1 + num2;
            break;
        case '-':
            result = num1 - num2;
            break;
        case '*':
            result = num1 * num2;
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
            } else {
                printf("Error: Division by zero!\n");
                return 1;
```

```
            }
            break;
        default:
            printf("Error: Invalid operator!\n");
            return 1;
    }

    printf("%.2f %c %.2f = %.2f\n", num1, operator, num2, result);
    return 0;
}
```

Example 2: Grade Classification

```
#include <stdio.h>

int main() {
    int score;

    printf("Enter your score (0-100): ");
    scanf("%d", &score);

    if (score < 0 || score > 100) {
        printf("Invalid score!\n");
    } else if (score >= 90) {
        printf("Grade: A (Excellent)\n");
    } else if (score >= 80) {
        printf("Grade: B (Very Good)\n");
    } else if (score >= 70) {
        printf("Grade: C (Good)\n");
    } else if (score >= 60) {
        printf("Grade: D (Fair)\n");
    } else {
        printf("Grade: F (Fail)\n");
    }

    return 0;
}
```

8.2 Loop Examples

Example 1: Sum of Numbers

```
#include <stdio.h>

int main() {
    int n, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        sum += i;
    }

    printf("Sum of numbers from 1 to %d is: %d\n", n, sum);
    return 0;
}
```

Example 2: Multiplication Table

```
#include <stdio.h>

int main() {
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Multiplication table for %d:\n", num);
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", num, i, num * i);
    }

    return 0;
}
```

9. Common Debugging Tips

9.1 Compilation Errors

1. **Missing semicolons:** Add `;` at the end of statements
2. **Undeclared variables:** Declare variables before using them
3. **Type mismatches:** Ensure compatible types in assignments (*this mostly happens in function parameters or calling, we will learn more about this on the next module*)
4. **Missing headers:** Include necessary header files

9.2 Runtime Errors

1. **Segmentation faults:** Check array bounds and pointer usage.
   - *this error is the hardest part when it comes to debugging because its highly related to a wrong memory allocation or pointer usage such as accessing an unaccessible variable or array index. We will learn more about this in the next module*
2. **Infinite loops:** Verify loop conditions and increment/decrement
3. **Wrong output:** Check format specifiers in printf/scanf

9.3 Debugging Techniques

```
// Add debug prints to trace program execution
printf("Debug: x = %d, y = %d\n", x, y);

// Check intermediate results
int temp = a + b;
printf("Intermediate result: %d\n", temp);
int result = temp * c;
```