Module 2 : Functions in C

Learning Objectives:

- Understand the concept and importance of functions in C programming
- Declare and define functions with proper syntax
- Use function parameters and return values effectively
- Apply different parameter passing mechanisms (pass by value)
- Understand variable scope and lifetime concepts
- Implement recursive functions
- Use standard library functions effectively
- Migrate from Python functions to C functions
- Debug common function-related errors

1. Introduction to Functions

1.1 What are Functions?

Functions are self-contained blocks of code that perform specific tasks. They are fundamental building blocks that help organize code, promote reusability, and make programs more modular and maintainable.

Benefits of Functions:

- **Code Reusability:** Write once, use multiple times
- **Modularity:** Break complex problems into smaller, manageable pieces
- **Maintainability:** Easier to debug, test, and modify
- **Readability:** Makes code more organized and understandable
- **Abstraction:** Hide implementation details from the caller

1.2 Why Use Functions? (Comparison with and without)

To understand the practical benefits of functions, let's consider a simple task: calculating the area of three different rectangles.

Without Functions:

```
#include <stdio.h>

int main() {
    // Rectangle 1
    int length1 = 10;
    int width1 = 5;
    int area1 = length1 * width1;
    printf("Area of Rectangle 1: %d\n", area1);

    // Rectangle 2
    int length2 = 12;
    int width2 = 8;
    int area2 = length2 * width2;
    printf("Area of Rectangle 2: %d\n", area2);
```

```
    // Rectangle 3
    int length3 = 7;
    int width3 = 3;
    int area3 = length3 * width3;
    printf("Area of Rectangle 3: %d\n", area3);

    return 0;
}
```

In this example, the logic for calculating the area is repeated three times. If we needed to change how the area is calculated (e.g., add a margin), we would have to modify each instance, which is error-prone and inefficient.

With Functions:

```
#include <stdio.h>

// Function to calculate rectangle area
int calculate_rectangle_area(int length, int width) {
    return length * width;
}

int main() {
    // Rectangle 1
    int area1 = calculate_rectangle_area(10, 5);
    printf("Area of Rectangle 1: %d\n", area1);

    // Rectangle 2
    int area2 = calculate_rectangle_area(12, 8);
    printf("Area of Rectangle 2: %d\n", area2);

    // Rectangle 3
    int area3 = calculate_rectangle_area(7, 3);
    printf("Area of Rectangle 3: %d\n", area3);

    return 0;
}
```

By using a function `calculate_rectangle_area`, we write the area calculation logic only once. This demonstrates:

- **Reusability:** The `calculate_rectangle_area` function can be used multiple times with different inputs.
- **Readability:** The `main` function becomes cleaner and easier to understand, as the details of area calculation are encapsulated within the function.
- **Maintainability:** If the area calculation logic needs to change, we only need to modify it in one place (inside the function definition), and all calls to that function will automatically use the updated logic.

1.3 Python vs C Functions Comparison

Aspect	Python	C
Declaration	Not required	Function prototype usually required (unless defined before use)
Definition	`def function_name():`	`return_type function_name() { }`
Return Type	Dynamic (any type)	Must be explicitly declared
Parameters	Dynamic typing	Static typing required
Call Before Definition	Allowed	Requires prototype
Multiple Return Values	`return a, b`	Use pointers or structures

Python Example:

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print(result)  # Output: 8
```

C Equivalent:

```
#include <stdio.h>

// Function declaration (prototype)
int add_numbers(int a, int b);

int main() {
    int result = add_numbers(5, 3);
    printf("%d\n", result);  // Output: 8
    return 0;
}
```

```
// Function definition
int add_numbers(int a, int b) {
    return a + b;
}
```

2. Function Declaration, Definition, and Calling

2.1 Function Anatomy

A C function consists of several parts:

```
return_type function_name(parameter_list) {
    // Function body
    // Local variables
    // Statements
    return value; // (if return_type is not void)
}
```

Components:

1. **Return Type:** Data type of the value returned (int, float, char, void, etc.)
2. **Function Name:** Identifier for the function
3. **Parameter List:** Input values (formal parameters)
4. **Function Body:** Statements enclosed in braces
5. **Return Statement:** Returns control and optionally a value

2.2 Function Declaration (Prototype)

Function prototypes declare the function's interface before its actual definition. They are necessary when a function is called before its definition in the source code. This allows the compiler to check for correct usage and enables forward referencing. If a function is defined *before* it is called, a prototype is not strictly necessary.

Syntax:

```
return_type function_name(parameter_types);
```

Examples:

```
// Function prototypes
int add(int a, int b);                    // Two int parameters, returns int
float calculate_area(float length, float width);  // Two float parameters, returns float
```

```
void print_message(void);                // No parameters, no return value
char get_grade(int score);              // One int parameter, returns char
```

Important Notes:

- Prototypes end with semicolon (`;`)
- Parameter names are optional in prototypes (but recommended for clarity)
- Must match exactly with function definition

2.3 Function Definition

The function definition contains the actual implementation:

```
#include <stdio.h>

// Function prototype
int multiply(int x, int y);

int main() {
    int result = multiply(4, 5);
    printf("Result: %d\n", result);
    return 0;
}

// Function definition
int multiply(int x, int y) {
    int product = x * y;
    return product;
}
```

2.4 Function Calling

After a function has been declared (prototyped) and defined, it can be executed, or "called," from another part of the program (e.g., from `main()` or another function). When a function is called, the program's control is transferred to that function.

Syntax for Calling a Function:

```
function_name(arguments);
```

- `function_name`: The name of the function to be executed.

- `arguments`: The values (actual parameters) passed to the function. These must match the type and order of the parameters in the function's declaration.

Example: In the `main()` function of the previous example, `multiply(4, 5);` is a function call.

```
int main() {
    int result = multiply(4, 5); // Calling the 'multiply' function
    printf("Result: %d\n", result);
    return 0;
}
```

Here, `4` and `5` are the arguments passed to the `multiply` function. The return value of `multiply` is then stored in the `result` variable.

2.5 void Functions

Keep in mind : Functions that don't return a value use `void` as the return type:

```
#include <stdio.h>

void print_header(void);
void print_line(int length);

int main() {
    print_header();
    print_line(30);
    return 0;
}

void print_header(void) {
    printf("=== STUDENT MANAGEMENT SYSTEM ===\n");
}

void print_line(int length) {
    for (int i = 0; i < length; i++) {
        printf("-");
    }
    printf("\n");
}
```

3. Parameters and Arguments

3.1 Terminology

- **Parameters (Formal Parameters):** Variables in the function definition
- **Arguments (Actual Parameters):** Values passed when calling the function

```
// 'a' and 'b' are parameters
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 3;
    // 'x' and 'y' are arguments
    int result = add(x, y);
    return 0;
}
```

3.2 Parameter Passing in C

C uses **pass by value** as its primary parameter passing mechanism. This means:

- A copy of the argument's value is made
- Changes to parameters inside the function don't affect the original variables
- The original variables remain unchanged

Example:

```
#include <stdio.h>

void modify_value(int num) {
    num = 100;  // This only changes the local copy
    printf("Inside function: %d\n", num);  // Output: 100
}

int main() {
    int original = 50;
```

```
    printf("Before function call: %d\n", original);  // Output: 50

    modify_value(original);

    printf("After function call: %d\n", original);   // Output: 50 (unchanged!)
    return 0;
}
```

3.3 Python vs C Parameter Passing

Python:

```
def modify_list(lst):
    lst.append(4)  # Modifies the original list

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list)  # Output: [1, 2, 3, 4]
```

C (Basic Types):

```
void modify_number(int num) {
    num = 100;  // Only modifies the copy
}

int main() {
    int number = 50;
    modify_number(number);
    // number is still 50
    return 0;
}
```

3.4 Multiple Parameters

Functions can have multiple parameters of different types:

```
#include <stdio.h>

// Function to calculate compound interest
```

```
float calculate_compound_interest(float principal, float rate, int time, int n) {
    float amount = principal;
    float rate_per_period = rate / (100.0 * n);
    int total_periods = n * time;

    for (int i = 0; i < total_periods; i++) {
        amount *= (1 + rate_per_period);
    }

    return amount - principal;  // Return only the interest
}

int main() {
    float principal = 1000.0;
    float annual_rate = 5.0;    // 5% per year
    int years = 2;
    int compounding_freq = 4;   // Quarterly

    float interest = calculate_compound_interest(principal, annual_rate, years,
compounding_freq);

    printf("Principal: $%.2f\n", principal);
    printf("Interest earned: $%.2f\n", interest);
    printf("Total amount: $%.2f\n", principal + interest);

    return 0;
}
```

4. Return Statement

4.1 Basic Return Usage

The `return` statement serves two purposes:

1. **Return control** to the calling function
2. **Return a value** (optional, depending on function type)

```
// Function that returns a value
int get_maximum(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Function that returns without a value
void print_status(int score) {
    if (score < 0) {
        printf("Invalid score!\n");
        return;  // Early exit
    }

    if (score >= 60) {
        printf("Passed!\n");
    } else {
        printf("Failed!\n");
    }
    // Implicit return at end of void function
}
```

4.2 Multiple Return Statements

A function can have multiple return statements, but only one will execute:

```
char determine_grade(int score) {
    if (score >= 90) {
        return 'A';
    }
    if (score >= 80) {
        return 'B';
    }
    if (score >= 70) {
        return 'C';
    }
    if (score >= 60) {
        return 'D';
    }
    return 'F';  // Default case
}
```

4.3 Returning Different Data Types

```
#include <stdio.h>

// Return integer
int get_absolute(int num) {
    if (num < 0) {
        return -num;
    } else {
        return num;
    }
}

// Return float
float celsius_to_fahrenheit(float celsius) {
    return (celsius * 9.0 / 5.0) + 32.0;
}

// Return character
char get_letter_grade(float percentage) {
    if (percentage >= 85.0) return 'A';
    if (percentage >= 75.0) return 'B';
    if (percentage >= 65.0) return 'C';
```

```
    if (percentage >= 50.0) return 'D';
    return 'F';
}

int main() {
    printf("Absolute value of -15: %d\n", get_absolute(-15));
    printf("25°C in Fahrenheit: %.1f°F\n", celsius_to_fahrenheit(25.0));
    printf("Grade for 78.5%%: %c\n", get_letter_grade(78.5));
    return 0;
}
```

5. Variable Scope and Lifetime

5.1 Local Variables

Variables declared inside a function are **local** to that function:

- **Scope:** Only accessible within the function where they're declared
- **Lifetime:** Created when function is called, destroyed when function returns
- **Storage:** Usually on the stack

```
#include <stdio.h>

void function_a() {
    int local_var = 10;  // Local to function_a
    printf("In function_a: %d\n", local_var);
}

void function_b() {
    int local_var = 20;  // Different variable, local to function_b
    printf("In function_b: %d\n", local_var);
}

int main() {
    int local_var = 5;   // Local to main
    printf("In main: %d\n", local_var);

    function_a();  // Output: In function_a: 10
    function_b();  // Output: In function_b: 20

    printf("Back in main: %d\n", local_var);  // Still 5
    return 0;
}
```

5.2 Global Variables

Variables declared outside all functions are **global**:

- **Scope:** Accessible from any function in the program

- **Lifetime:** Exist for the entire program execution
- **Storage:** In static memory area

```
#include <stdio.h>

// Global variables
int global_counter = 0;
float global_sum = 0.0;

void increment_counter() {
    global_counter++;  // Can access global variable
    printf("Counter: %d\n", global_counter);
}

void add_to_sum(float value) {
    global_sum += value;  // Can modify global variable
    printf("Sum: %.2f\n", global_sum);
}

int main() {
    increment_counter();  // Counter: 1
    increment_counter();  // Counter: 2

    add_to_sum(10.5);     // Sum: 10.50
    add_to_sum(7.3);      // Sum: 17.80

    printf("Final values - Counter: %d, Sum: %.2f\n", global_counter, global_sum);
    return 0;
}
```

5.3 Variable Shadowing

When a local variable has the same name as a global variable, the local variable "shadows" the global one:

```
#include <stdio.h>

int value = 100;  // Global variable

void test_shadowing() {
```

```
    int value = 50;  // Local variable shadows the global one
    printf("Local value: %d\n", value);  // Prints 50
}

int main() {
    printf("Global value: %d\n", value);  // Prints 100
    test_shadowing();
    printf("Global value after function: %d\n", value);  // Still 100
    return 0;
}
```

5.4 Best Practices for Variable Scope

1. **Minimize global variables:** Use them sparingly
2. **Prefer local variables:** Keep data close to where it's used
3. **Use meaningful names:** Avoid naming conflicts
4. **Initialize variables:** Always initialize before use

```
// Good practice
int calculate_area(int length, int width) {
    int area = length * width;  // Local variable, clearly named
    return area;
}

// Avoid this
int x, y, z;  // Global variables - hard to track
int calc(int a, int b) {
    z = a * b;  // Modifying global state
    return z;
}
```

6. Bonus: Some C Library Functions

Throughout this module, we've focused on defining our own functions. However, C also provides a rich set of built-in functions, known as **Standard Library Functions**. These functions are pre-written and grouped into various libraries (e.g., `<stdio.h>`, `<stdlib.h>`, `<math.h>`) to perform common tasks like input/output, memory management, and mathematical operations. When you use functions like `printf()` or `scanf()`, you are actually calling functions that have been defined in the standard input/output library (`stdio.h`). These functions abstract away complex implementations, allowing you to use them easily by simply including their respective header files.

6.1 Mathematical Functions

Include `<math.h>` header for mathematical functions:

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

int main() {
    double angle = 45.0;
    double radians = angle * M_PI / 180.0;  // Convert to radians

    printf("Mathematical Functions:\n");
    printf("sqrt(16) = %.2f\n", sqrt(16.0));           // Square root
    printf("pow(2, 3) = %.2f\n", pow(2.0, 3.0));       // 2^3
    printf("sin(45°) = %.4f\n", sin(radians));         // Sine
    printf("cos(45°) = %.4f\n", cos(radians));         // Cosine
    printf("log(10) = %.4f\n", log(10.0));             // Natural log
    printf("log10(100) = %.2f\n", log10(100.0));       // Base-10 log
    printf("ceil(4.3) = %.0f\n", ceil(4.3));           // Ceiling
    printf("floor(4.7) = %.0f\n", floor(4.7));         // Floor
    printf("fabs(-5.5) = %.1f\n", fabs(-5.5));         // Absolute value

    return 0;
}
```

Note: When compiling programs that use `<math.h>`, you might need to link the math library:

```
gcc -o program program.c -lm
```

6.2 String Functions

Include `<string.h>` header for string manipulation:

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

int main() {
    char str1[50] = "Hello";
    char str2[50] = "World";
    char str3[100];

    // String length
    printf("Length of '%s': %lu\n", str1, strlen(str1));

    // String copy
    strcpy(str3, str1);
    printf("After strcpy: str3 = '%s'\n", str3);

    // String concatenation
    strcat(str3, " ");
    strcat(str3, str2);
    printf("After strcat: str3 = '%s'\n", str3);

    // String comparison
    if (strcmp(str1, "Hello") == 0) {
        printf("str1 equals 'Hello'\n");
    }

    return 0;
}
```

6.3 Character Functions

Include `<ctype.h>` for character testing and conversion:

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

int main() {
    char ch = 'A';

    printf("Character testing for '%c':\n", ch);
    printf("isalpha: %d\n", isalpha(ch));      // Is alphabetic?
    printf("isdigit: %d\n", isdigit(ch));      // Is digit?
    printf("isupper: %d\n", isupper(ch));      // Is uppercase?
    printf("islower: %d\n", islower(ch));      // Is lowercase?
    printf("isspace: %d\n", isspace(ch));      // Is whitespace?

    printf("Lowercase: %c\n", tolower(ch));    // Convert to lowercase
    printf("Uppercase: %c\n", toupper('a'));   // Convert to uppercase

    return 0;
}
```

7. Recursion

7.1 Understanding Recursion

Recursion is a programming technique where a function calls itself. Every recursive function needs:

1. **Base case:** Condition that stops the recursion
2. **Recursive case:** Function calls itself with modified parameters

7.2 Python vs C Recursion

Python Factorial:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

C Factorial:

```
#include <stdio.h>

int factorial(int n) {
    // Base case
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    printf("%d! = %d\n", num, factorial(num));  // Output: 5! = 120
    return 0;
```

```
}
```

7.3 How Recursion Works

Factorial(5) execution trace:

```
factorial(5) = 5 * factorial(4)
             = 5 * 4 * factorial(3)
             = 5 * 4 * 3 * factorial(2)
             = 5 * 4 * 3 * 2 * factorial(1)
             = 5 * 4 * 3 * 2 * 1
             = 120
```

7.4 More Recursive Examples

Fibonacci Sequence

```
#include <stdio.h>

int fibonacci(int n) {
    // Base cases
    if (n == 0) return 0;
    if (n == 1) return 1;

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    printf("Fibonacci sequence:\n");
    for (int i = 0; i < 10; i++) {
        printf("F(%d) = %d\n", i, fibonacci(i));
    }
    return 0;
}
```

Power Calculation

```c
#include <stdio.h>

double power(double base, int exponent) {
    // Base case
    if (exponent == 0) {
        return 1.0;
    }

    // Handle negative exponents
    if (exponent < 0) {
        return 1.0 / power(base, -exponent);
    }

    // Recursive case
    return base * power(base, exponent - 1);
}

int main() {
    printf("2^5 = %.2f\n", power(2.0, 5));    // Output: 32.00
    printf("3^-2 = %.4f\n", power(3.0, -2));  // Output: 0.1111
    return 0;
}
```

Sum of Array Elements (Preview for next module)

```c
#include <stdio.h>

int sum_array(int arr[], int size) {
    // Base case
    if (size == 0) {
        return 0;
    }

    // Recursive case
    return arr[size - 1] + sum_array(arr, size - 1);
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int array_size = 5;
```

```
    printf("Sum = %d\n", sum_array(numbers, array_size));  // Output: 15
    return 0;
}
```

7.5 Recursion vs Iteration

Advantages of Recursion:

- More natural for certain problems (tree traversal, mathematical definitions)
- Cleaner, more readable code for some algorithms
- Elegant solution for divide-and-conquer problems

Disadvantages of Recursion:

- Higher memory usage (function call stack)
- Slower execution due to function call overhead
- Risk of stack overflow for deep recursion
- It can causes Crash or freezes when things go wrong (the reasons are already mentioned right above)

When to Use Recursion:

- Mathematical sequences (factorial, fibonacci)
- Tree and graph algorithms
- Divide and conquer problems
- When the problem naturally breaks down into similar subproblems

8. Function Examples and Applications

8.1 Menu-Driven Program

```
#include <stdio.h>

// Function prototypes
void display_menu(void);
int get_choice(void);
void calculator_add(void);
void calculator_multiply(void);
void display_table(int num);

int main() {
    int choice;

    do {
        display_menu();
        choice = get_choice();

        switch (choice) {
            case 1:
                calculator_add();
                break;
            case 2:
                calculator_multiply();
                break;
            case 3:
                printf("Enter number for multiplication table: ");
                int num;
                scanf("%d", &num);
                display_table(num);
                break;
            case 4:
```

```
                printf("Thank you for using the calculator!\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }

        if (choice != 4) {
            printf("\nPress Enter to continue...");
            getchar();
            getchar();  // Clear buffer
        }

    } while (choice != 4);

    return 0;
}

void display_menu(void) {
    printf("\n=== SIMPLE CALCULATOR ===\n");
    printf("1. Addition\n");
    printf("2. Multiplication\n");
    printf("3. Multiplication Table\n");
    printf("4. Exit\n");
    printf("========================\n");
}

int get_choice(void) {
    int choice;
    printf("Enter your choice (1-4): ");
    scanf("%d", &choice);
    return choice;
}

void calculator_add(void) {
    float num1, num2;
    printf("Enter first number: ");
    scanf("%f", &num1);
    printf("Enter second number: ");
    scanf("%f", &num2);
```

```
    printf("Result: %.2f + %.2f = %.2f\n", num1, num2, num1 + num2);
}

void calculator_multiply(void) {
    float num1, num2;
    printf("Enter first number: ");
    scanf("%f", &num1);
    printf("Enter second number: ");
    scanf("%f", &num2);
    printf("Result: %.2f × %.2f = %.2f\n", num1, num2, num1 * num2);
}

void display_table(int num) {
    printf("\nMultiplication table for %d:\n", num);
    printf("-------------------------\n");
    for (int i = 1; i <= 10; i++) {
        printf("%d × %d = %d\n", num, i, num * i);
    }
}
```

8.2 Temperature Conversion Program

```
#include <stdio.h>

// Function prototypes
float celsius_to_fahrenheit(float celsius);
float fahrenheit_to_celsius(float fahrenheit);
float celsius_to_kelvin(float celsius);
float kelvin_to_celsius(float kelvin);
void display_conversion_table(void);

int main() {
    int choice;
    float temp, result;

    printf("=== TEMPERATURE CONVERTER ===\n");
    printf("1. Celsius to Fahrenheit\n");
    printf("2. Fahrenheit to Celsius\n");
    printf("3. Celsius to Kelvin\n");
```

```
printf("4. Kelvin to Celsius\n");
printf("5. Display Conversion Table\n");
printf("============================\n");

printf("Enter choice (1-5): ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter temperature in Celsius: ");
        scanf("%f", &temp);
        result = celsius_to_fahrenheit(temp);
        printf("%.2f°C = %.2f°F\n", temp, result);
        break;

    case 2:
        printf("Enter temperature in Fahrenheit: ");
        scanf("%f", &temp);
        result = fahrenheit_to_celsius(temp);
        printf("%.2f°F = %.2f°C\n", temp, result);
        break;

    case 3:
        printf("Enter temperature in Celsius: ");
        scanf("%f", &temp);
        result = celsius_to_kelvin(temp);
        printf("%.2f°C = %.2fK\n", temp, result);
        break;

    case 4:
        printf("Enter temperature in Kelvin: ");
        scanf("%f", &temp);
        if (temp < 0) {
            printf("Error: Temperature cannot be below 0 Kelvin!\n");
        } else {
            result = kelvin_to_celsius(temp);
            printf("%.2fK = %.2f°C\n", temp, result);
        }
        break;
```

```
        case 5:
            display_conversion_table();
            break;

        default:
            printf("Invalid choice!\n");
    }

    return 0;
}

float celsius_to_fahrenheit(float celsius) {
    return (celsius * 9.0 / 5.0) + 32.0;
}

float fahrenheit_to_celsius(float fahrenheit) {
    return (fahrenheit - 32.0) * 5.0 / 9.0;
}

float celsius_to_kelvin(float celsius) {
    return celsius + 273.15;
}

float kelvin_to_celsius(float kelvin) {
    return kelvin - 273.15;
}

void display_conversion_table(void) {
    printf("\n=== TEMPERATURE CONVERSION TABLE ===\n");
    printf("Celsius  | Fahrenheit | Kelvin\n");
    printf("---------+------------+--------\n");

    for (int c = 0; c <= 100; c += 10) {
        float f = celsius_to_fahrenheit(c);
        float k = celsius_to_kelvin(c);
        printf("%8d | %10.1f | %6.1f\n", c, f, k);
    }
}
```

9. Common Errors and Debugging

9.1 Function Declaration Errors

Error 1: Missing Function Prototype

```
// ERROR: Function used before declaration
int main() {
    int result = add_numbers(5, 3);  // Error: 'add_numbers' not declared
    return 0;
}

int add_numbers(int a, int b) {
    return a + b;
}
```

Solution:

```
// CORRECT: Add function prototype
int add_numbers(int a, int b);  // Function prototype

int main() {
    int result = add_numbers(5, 3);  // Now it works
    return 0;
}

int add_numbers(int a, int b) {
    return a + b;
}
```

9.2 Return Type Mismatches

Error 2: Wrong Return Type

```
// ERROR: Function declared to return int but returns float
int divide(int a, int b) {
    return a / b;  // Integer division, loses decimal part
}

int main() {
    printf("Result: %d\n", divide(7, 2));  // Output: 3 (not 3.5)
    return 0;
}
```

Solution:

```
// CORRECT: Use appropriate return type
float divide(int a, int b) {
    return (float)a / b;  // Cast to float for proper division
}

int main() {
    printf("Result: %.2f\n", divide(7, 2));  // Output: 3.50
    return 0;
}
```

9.3 Parameter Issues

Error 3: Expecting Changes to Original Variables

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf("Inside function: a=%d, b=%d\n", a, b);  // Values swapped
}

int main() {
    int x = 5, y = 10;
    swap(x, y);
    printf("In main: x=%d, y=%d\n", x, y);  // x=5, y=10 (unchanged!)
    return 0;
}
```

Understanding: This behavior is correct in C due to pass by value. To actually swap values, you would need pointers (weill be covered in module about "Pointer").

9.4 Debugging Techniques for Functions

1. Add Debug Prints:

```
int factorial(int n) {
    printf("DEBUG: factorial(%d) called\n", n);  // Debug output

    if (n == 0 || n == 1) {
        printf("DEBUG: base case reached, returning 1\n");
        return 1;
    } else {
        int result = n * factorial(n - 1);
        printf("DEBUG: factorial(%d) = %d\n", n, result);
        return result;
    }
}
```

2. Test Functions Independently:

```
// Test individual functions with known inputs
int main() {
    // Test factorial function
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(5) == 120);
    printf("All factorial tests passed!\n");

    return 0;
}
```

3. Check Function Signatures:

- Verify prototype matches definition
- Check parameter types and count
- Ensure return type is correct