Functions, Modular Programming & Pointers

COMP 2510 - Week 4

Hesam Alizadeh

Agenda & Objectives

Today's Roadmap (2 Hours)

Session 1: Structure & Build Systems

- **Why Functions?** Abstraction and Reusability
- **Scope:** Who can see my variables?
- **Modules:** Breaking up the "Big File"
- **Makefiles:** Automating the build

Session 2: Memory Mastery

- **The Pointer Concept:** Indirection
- **Mechanics:** Address-of (`&`) and Dereference (`*`)
- **Application:** Modifying data across scopes

Learning Objectives

- Write robust C functions to reduce code repetition
- Understand where variables live (Stack vs. Global)
- Organize code into `.c` and `.h` files
- **Master the basics of Pointers** to manipulate memory directly

Context: Where are we?

Recap: Week 2

- **Control Structures:** Logic flow (`if`, `loops`)
- **Arrays & Strings:** Data storage

The Problem:
So far, all our code has been inside `main()`. As logic gets complex, `main()` becomes a "Spaghetti Code" mess.

This Week: Organization

We are moving from **Scripting** to **Engineering**.

1. **Functions:** Organizing Logic
2. **Modules:** Organizing Files
3. **Pointers:** Organizing Memory

Why Functions?

The Problem: Code Duplication

```c
int main() {
    // Calculate area of rectangle 1
    int w1 = 5, h1 = 3;
    int area1 = w1 * h1;
    printf("Area: %d\n", area1);

    // Calculate area of rectangle 2
    int w2 = 7, h2 = 4;
    int area2 = w2 * h2;
    printf("Area: %d\n", area2);

    // Calculate area of rectangle 3
    int w3 = 10, h3 = 6;
    int area3 = w3 * h3;
    printf("Area: %d\n", area3);
}
```

Your next task: "Add perimeter calculation to all rectangles. How many lines do you need to change? What if you miss one?"

The Solution: Functions

```c
// Write logic ONCE
int rectangle_area(int w, int h) {
    return w * h;
}

int main() {
    // Reuse as many times as needed
    printf("%d\n", rectangle_area(5, 3));
    printf("%d\n", rectangle_area(7, 4));
    printf("%d\n", rectangle_area(10, 6));
}
```

Abstraction: Hide complexity behind a simple name

Function Anatomy

1. Prototype (The Promise)

Tells the compiler: "This function exists."

```
  int add(int a, int b);
// ↑    ↑     ↑      ↑
// |    |     |      Second parameter
// |    |     First parameter
// |    Function name
// Return type
```

2. Definition (The Implementation)

The actual logic.

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

Why we need prototypes:

The compiler reads top-to-bottom. It can't use what it hasn't seen yet. So, we do all the declarations ahead of time.

```
// Option 1: Prototype first
int add(int a, int b);  // Promise to define it later

int main() {
    int result = add(5, 10);
    return 0;
}

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

```
// Option 2: Define before main
int add(int a, int b) {  // Define the function first
    return a + b;
}

int main() {
    int result = add(5, 10);
    return 0;
}
```

Variable Scope

Who can see what?

Just because you declared a variable doesn't mean the whole program can use it.

Key insight: Variables are visible only within their block (`{}`).

```c
int global_x = 100;  // Visible EVERYWHERE

void my_function() {
    int local_y = 50;  // Visible ONLY in my_function
    printf("%d", local_y);  // OK
}

int main() {
    printf("%d", global_x);  // OK
    printf("%d", local_y);   // local_y doesn't exist here
}
```

TYPE	WHERE	LIFETIME	VISIBILITY
Local	Inside `{}`	Until function ends	Only that function
Global	Outside all	Entire program	Every function

Warning: Avoid globals! Anyone can change them anywhere. Debugging becomes: "Who changed my variable?!"

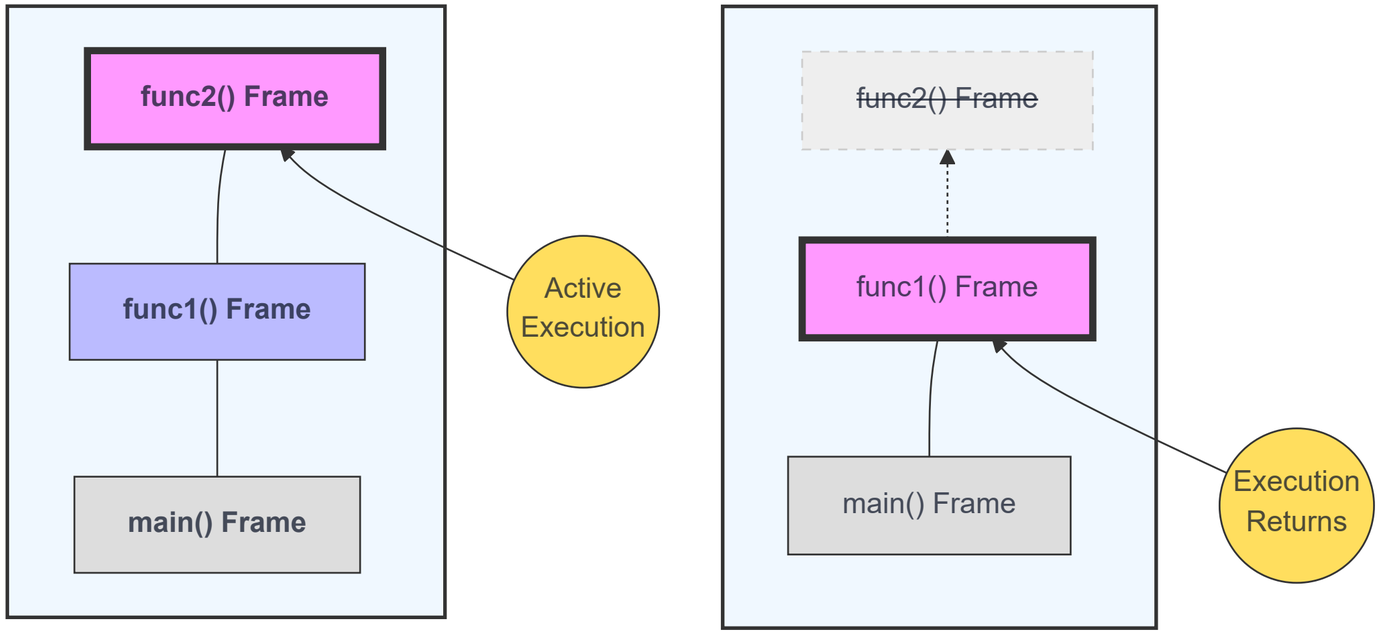
Visualizing The Stack

The "Stack" of Plates

Computer memory for functions works like a stack of plates in a cafeteria.

1. `main()` starts: Put a plate down in the stack. (**Push**)
2. `main` calls `func1`: Put a plate on top of it. (**Push**)
3. `func1` calls `func2`: Put another one on top. (**Push**)
4. `func2` finishes: **Remove the top plate. (Pop)**

Crucial: When the plate (frame) is removed, all local variables on it are destroyed.



Activity 1: Scope Analysis

Identify the output

```c
#include <stdio.h>
int x = 100; // Global

int main() {
    int x = 10; // Local main

    if (x == 10) {
        int x = 5; // Local if-block
        printf("%d ", x);
    }

    printf("%d", x);
    return 0;
}
```

Options:

A) `5 5` B) `5 10` C) `10 100` D) `5 100`

Hand-in update

Update the hand-in with your answer.

Parameter Passing

The "Photocopy" Rule

C uses **Pass-by-Value**.

When you pass a variable to a function, you are not passing the variable itself. You are passing a **photocopy** of the value.

If the function scribbles on the photocopy, the original document in `main` is unchanged.

```c
void attempt_change(int x) {
    x = 999;
    // Modifies the COPY (local x)
}

int main() {
    int val = 10;

    attempt_change(val);

    printf("%d", val);
    // Prints 10 (Original untouched)
}
```

Activity 2: Code Prediction

What will this code output?

```c
#include <stdio.h>

void modify(int x) {
    x = x + 10;
    printf("Inside: %d\n", x);
}

int main() {
    int val = 5;
    modify(val);
    printf("Outside: %d\n", val);
    return 0;
}
```

Turn to your neighbor: Explain why this prints different values inside vs. outside.
Q: Is there another way to achieve what we wanted to happen?

Hand-in update

Update the hand-in with your answer.

Modular Programming

The "Big File" Problem

Imagine if Wikipedia was just one single `.txt` file with 5 billion lines. Finding anything would be impossible.

Software is the same.
We don't write 50,000 lines of code in `main.c`.

The Solution: Split by Responsibility

```
my_project/
├── main.c          (Entry point, coordinates everything)
├── math_utils.c    (All calculations)
├── display.c       (All UI/graphics)
├── network.c       (All WiFi/internet)
└── logger.c        (All file logging)
```

Benefits:

- Easier to find code
- Multiple developers can work in parallel
- Reusable across projects

Interface vs Implementation

C separates these using two file types:

`.h` **files (Headers)** - The **Contract**

- "*I promise these functions exist with these signatures*"
- Contains function prototypes, constants, etc

`.c` **files (Source)** - The **Fulfillment**

- "*Here's how I keep that promise*"
- Actual function logic

Header Files in Practice

The Problem: Duplicate Includes

What happens here?

```
// main.c
#include "math_utils.h"
#include "display.h"  // This also includes math_utils.h

// Now math_utils.h contents appear TWICE!
// Compiler error: "multiple definition of..."
```

The Solution: Include Guards

```
// math_utils.h
#ifndef MATH_UTILS_H  // "if not defined"
#define MATH_UTILS_H  // "define it now"

// declarations go here
int add(int a, int b);
int multiply(int a, int b);

#endif  // End of guard
```

Using a Custom Header

main.c

```
#include <stdio.h>
#include "math_utils.h"  // Use quotes for local headers

int main() {
    printf("%d", add(5, 10));
    return 0;
}
```

Tip: Use `<angle brackets>` for system headers, `"quotes"` for your own.

Activity 3: Modular Construction

Where do these lines belong?

Scenario: You are writing a library `math_utils`.
Sort these snippets into the correct file:

A. `int square(int x);`

B. `printf("The square is %d", square(5));`

C. `int square(int x) { return x * x; }`

Options:

1. Header (.h)
2. Implementation (.c)
3. Main (.c)

Hand-in update

Update the hand-in with your answer.

Makefiles (The Manual Way)

With 3+ source files, compiling becomes tedious:

```
gcc main.c utils.c logger.c -o myapp
```

And recompiling everything when one file changes is slow.

The Solution: Make

A `Makefile` is a script that tells the `make` utility how to compile your project.

Anatomy of a Rule:

```
target: dependencies
     command
```

1. **Target:** What to build (e.g., `app`).
2. **Dependencies:** What is needed (e.g., `main.c`).
3. **Command:** The generic shell command.

The TAB Rule

Crucial: The indentation before the command **MUST be a TAB**. Spaces will fail.

```
CC = gcc
CFLAGS = -Wall -Wextra -g

# Target: Dependencies
myapp: main.c utils.c logger.c
     $(CC) $(CFLAGS) -o myapp main.c utils.c logger.c

# Clean rule (Good practice)
clean:
     rm -f myapp
```

To run this, simply type `make` in the terminal.

The Modern Way: CMake

Why CMake?

Writing Makefiles by hand is brittle (TABs vs spaces) and hard to manage for huge projects.

CMake is a "Build System Generator".
You don't write the Makefile. You write a high-level description, and CMake **generates** the Makefile for you.

Pros:

- Cross-platform (Linux, Mac, Windows).
- Handles complex dependencies automatically.
- No "Tab vs Space" issues.

The `CMakeLists.txt` File

Instead of a Makefile, you write this:

```
cmake_minimum_required(VERSION 3.10)

# 1. Project Name
project(MyCoolApp)

# 2. Compiler Flags (Warnings + Debug)
# Equivalent to CFLAGS = -Wall -g
add_compile_options(-Wall -Wextra -g)

# 3. Create Executable
# Syntax: add_executable(Name SourceFiles...)
add_executable(myapp
    main.c
    utils.c
    display.c
)
```

The CMake Workflow

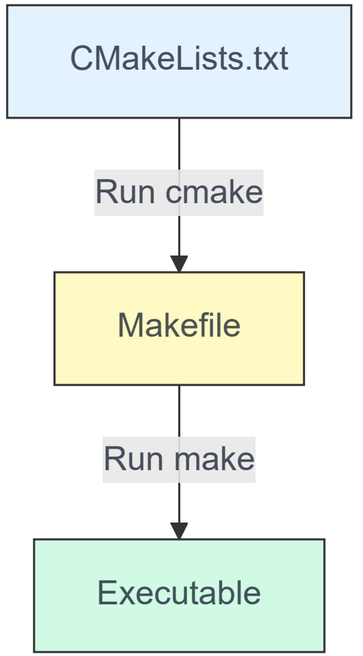
How to build with CMake

It is a two-step process.

1. **Generate:** Ask CMake to read your list and create the Makefile.
   `cmake .`
2. **Build:** Use the generated Makefile to compile the code.
   `make`

Why two steps?
Because on Windows, Step 1 might generate a Visual Studio project instead of a Makefile!

Visualization



Industry Standard:
Almost all professional C/C++ projects today use CMake.

Part 1 Summary

We covered:

1. **Functions:** Hiding complexity via abstraction and understanding the **Call Stack**.
2. **Scope:** Variables are local by default; they die when the function returns.
3. **Modules:** Splitting code into **Interfaces** (`.h`) and **Implementations** (`.c`).
4. **Build Systems:**
   - **Make:** The manual "Recipe Card" (strict syntax).
   - **CMake:** The modern "Factory" that generates Makefiles for us.

Next Up: The most powerful (and dangerous) feature of C: **Pointers**.

The Missing Piece

Remember the "Copy Problem"?

Functions receive copies, so they can't modify originals.

We have a gap in our toolkit:

NEED	CURRENT SOLUTION	PROBLEM
Read data	Pass-by-value	Works fine
Modify data	Global variables	Dangerous!
Safe modification	???	This is why we need pointers

Quick Review:

1. What's the difference between a function prototype and definition?
2. Why are include guards necessary?

Why Pointers Matter

Solving the "Heavy Data" Problem

Remember **Pass-by-Value** (The Photocopy)?
If you have a massive struct (e.g., a 1MB image), passing it to a function copies all 1MB. That is slow.

The Solution: Pass-by-Reference
Instead of passing the 1MB image, you pass a pointer (8 bytes). The function follows the pointer to the original data.

The "House Painter" Analogy

Option A (Direct Access):
You pick up your entire house, put it on a truck, and drive it to the painter.
Result: Exhausting and slow.

Option B (Indirect Access):
You write your address on a sticky note and hand it to the painter.
Result: Instant. They drive to your house and paint it.

> **Efficiency:** Pointers allow us to manipulate large data without the overhead of copying it.

Memory & Addresses

Every variable you create lives in a numbered slot in memory.

- **Variable Name:** `score` (The label)
- **Value:** `95` (The contents)
- **Address:** `0x7ffee4` (The shelf number)

To see an address in C, we use the **Address-of Operator (`&`)**.

```c
int score = 95;
printf("Value: %d\n", score);

// Prints the memory address
printf("Address: %p\n", &score);
```

The Pointer Variable

A pointer is just a variable. But instead of holding a number like `95`, it holds an **address** like `0x7ffee4`.

Declaration:

`type *name;`

```c
int *ptr;  // Will hold address of an int
```

Assignment:

`ptr = &score;`

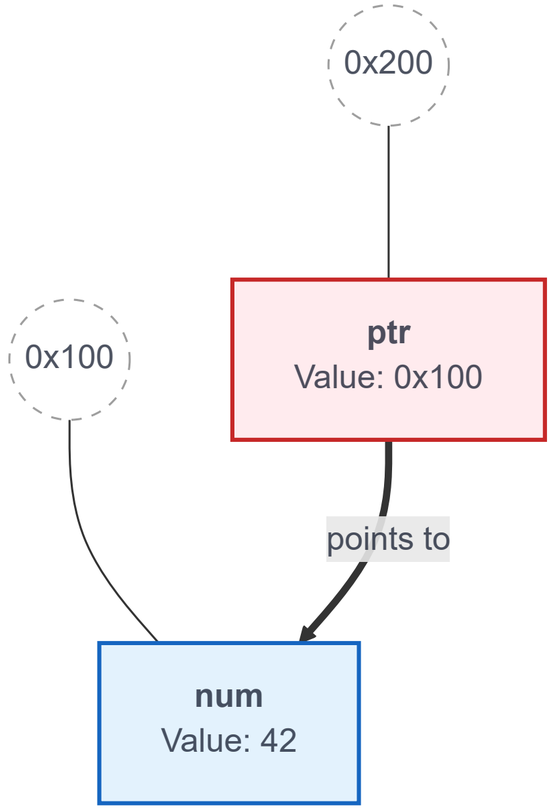
Now `ptr` "points to" `score`.

Visualizing Pointers

The Memory Map

1. We have an integer `num` with value `42` living at address `0x100`.
2. We have a pointer `ptr` living at `0x200`.
3. `ptr` holds the value `0x100`.
4. Therefore, `ptr` **points** to `num`.

```
int num = 42;
int *ptr = &num;
```



Dereferencing

Accessing the Data (`*`)

We have the address (the sticky note). How do we get the actual value (the book page)?

We use the **Dereference Operator (`*`)**.

- `ptr` → The address (e.g., `0x100`)
- `*ptr` → The value AT that address (e.g., `42`)

```c
int a = 10;
int *p = &a;

printf("%d", *p); // Prints 10

*p = 20;          // Goes to 0x100, puts 20 there
printf("%d", a);  // Prints 20. 'a' changed!
```

Context Note:

The symbol `*` is confusing because it is used in two ways:

1. **Declaration:** `int *p;` (I am creating a pointer)
2. **Usage:** `*p = 5;` (Follow the pointer to the data)

Activity 4: Address Logic

Assume the following memory map:

VARIABLE	ADDRESS	VALUE
`a`	`0x100`	`50`
`b`	`0x104`	`99`
`p`	`0x108`	`0x100`

```
int a = 50;
int *p = &a;
```

Questions:

1. What is the value of `p`?
2. What is the value of `&a`?
3. What is the value of `&p`?

Hand-in update

Update the hand-in with your answer.

Pointers & Functions

Solving the "Copy" Problem

Remember how functions couldn't change the original variable? Pointers solve this.

Simulating Pass-by-Reference:

1. **Function:** Asks for an address (`int *p`).
2. **Caller:** Sends the address (`&val`).
3. **Result:** Function follows the address to modify the original.

The Classic Swap

```c
// Works!
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;     // Write to original A
    *b = temp;   // Write to original B
}

int main() {
    int x = 5, y = 10;
    // Pass addresses, not values
    swap(&x, &y);
    // x is now 10, y is 5
}
```

Activity 5: Mini-Coding

Write a "Reset" Function

Write a function named `reset` that takes two integer pointers and sets the value of both variables to 0.

Signature:

`void reset(int *a, int *b);`

Usage:

```
int x = 100, y = 200;
reset(&x, &y);
// x and y should both be 0 now
```

Hand-in update

Update the hand-in with your answer.

Common Pitfalls

Working with raw memory is powerful but dangerous.

1. Uninitialized Pointers

Declaring `int *p;` creates a pointer to "garbage" (random memory). Dereferencing it causes a **Segmentation Fault** (crash).

2. NULL Pointers

If a pointer has nowhere to point, assign it `NULL`.

```
int *p = NULL;
// ... later ...
if (p != NULL) {
    // Safe to use
    *p = 10;
}
```

The Crash Zone

```
// DANGEROUS CODE

int *p;
// p contains random garbage address
// maybe 0x000000 or 0xA12B...

*p = 5;
// CRASH!
// Trying to write to unknown memory
```

Rule: Initialize pointers immediately: `int *p = NULL;` or `int *p = &x;`.

Activity 6: Spot the Bug

Fix the Crash

This code crashes because `ptr` is uninitialized.

Which line of code fixes it safely?

```c
int main() {
    int val = 50;
    int *ptr;  // DANGER!

    // [INSERT LINE HERE]

    *ptr = 100;
    printf("%d", val);
    return 0;
}
```

Options:

A) `ptr = 0;`

B) `ptr = &val;`

C) `*ptr = val;`

D) `&ptr = val;`

Hand-in update

Update the hand-in with your answer.

Submitting Your Work

Hand-in Instructions

1. **Compile screenshots** from all activities.
2. **Paste screenshots** into the provided template.
3. **Add written answers** where code wasn't required.
4. **Save as PDF** named `Lastname_Firstname_W4_Activity.pdf`.
5. **Upload** to the Assignment folder "Week 4 In-Class".

Deadline: End of day today

Lecture Summary

Key Takeaways

1. **Functions & Scope:** Logic abstraction and the "Stack" lifecycle.
2. **Structure & Builds:** Splitting files (`.c` / `.h`) and automating compilation with **Make/CMake**.
3. **Pointers:** Using addresses (`&`) to access data indirectly.
4. **Pass-by-Reference:** Using dereferencing (`*`) to modify variables efficiently.

Next Week (Week 5)

Pointers II: Arrays & Strings

- Pointer Arithmetic in depth
- String manipulation with pointers

Reminders:

- **Quiz 1:** In the lab