C Compilation Environment Test

Software Engineering — Junior Lab Walkthrough

What We're Doing

Today you're going to set up a C compiler on your machine and compile a few small C programs. This is a quick environment check — we want to make sure everything works on your computer before we go deeper into C later this year.

You'll be working in a **MinGW64 shell**, which is a Unix-style terminal.

Part 1 — Install the GCC Compiler

Your machine already has MSYS2 installed. We just need to add the actual C compiler (GCC) to it.

Open the MinGW64 Shell

Find and double-click this file:

```
C:\msys64\mingw64.exe
```

A terminal window will open with a colored prompt. **This is NOT the regular Windows command prompt** — it's a Unix-style shell. You'll do all your work in here.

Install GCC

Type this command and press Enter:

```
pacman -S mingw-w64-x86_64-gcc
```

When it asks you to proceed with installation, type **Y** and press Enter. Let it finish — it may take a minute.

Verify It Worked

```
gcc --version
```

You should see something like `gcc (Rev...) 14.x.x` or similar. If you see version info, you're good. If you get `command not found`, raise your hand.

Part 2 — Your First C Program: Hello World

Create a Working Folder

```
mkdir -p ~/c_practice
cd ~/c_practice
```

This creates a folder called `c_practice` in your home directory and moves you into it. All your work today goes here.

Write the Program

Open the **nano** text editor to create a new file:

```
nano hello.c
```

Type this program **exactly** as shown:

```c
#include <stdio.h>

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

A few things to notice:

- `#include <stdio.h>` tells C to load the standard input/output library (that's what gives us `printf`)
- `int main()` is the entry point — every C program starts here
- `\n` means "new line" — like pressing Enter in the output
- Every statement ends with a semicolon `;`

Save and Exit Nano

1. Press **Ctrl+O** (that's the letter O, not zero) to write/save the file
2. Press **Enter** to confirm the filename
3. Press **Ctrl+X** to exit nano

Compile It

```
gcc hello.c -o hello.exe
```

This tells GCC to take your `hello.c` source code and compile it into an executable called `hello.exe`. If you see no output, that's a good thing — it means no errors.

If you DO see errors, read them carefully. They'll tell you the line number where something went wrong. Open the file again with `nano hello.c` and fix it.

Run It

```
./hello.exe
```

☑ **You should see:** `Hello, World!`

Congrats — you just wrote, compiled, and ran a C program.

Part 3 — Basic Math

Let's try something with variables and arithmetic. Create a new file:

```
nano math_test.c
```

Type this program:

```c
#include <stdio.h>

int main() {
    int a = 10;
    int b = 3;

    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n", a % b);

    return 0;
}
```

Things to notice:

- `int a = 10;` declares an integer variable. In C you **must** declare the type.
- `%d` is a **format specifier** — it's a placeholder that gets replaced by the value after the comma.
- `%%` prints a literal percent sign (since `%` is special in printf).
- `%` by itself is the **modulo** operator — it gives the remainder after division.

Save, exit nano, then compile and run:

```
gcc math_test.c -o math_test.exe
./math_test.exe
```

☑ **You should see:**

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
```

Wait — why is 10 / 3 equal to 3? Because both `a` and `b` are integers, C performs **integer division** and throws away the decimal part. It doesn't round — it truncates. The modulo operator (`%`) gives you the leftover: 10 divided by 3 is 3 remainder **1**.

Part 4 — User Input

Now let's make a program that takes input from you at runtime. Create `greeting.c`:

```
nano greeting.c
```

Type this:

```c
#include <stdio.h>

int main() {
    char name[50];

    printf("Enter your name: ");
    scanf("%49s", name);
    printf("Hello, %s! Welcome to C programming.\n", name);

    return 0;
}
```

Things to notice:

- `char name[50];` creates an array of 50 characters — this is how C handles text (called "strings").
- `scanf` reads input from the keyboard. `%49s` means "read up to 49 characters" (leaving room for the null terminator that marks the end of the string).
- `%s` in printf is the format specifier for strings, just like `%d` is for integers.

Compile and run:

```
gcc greeting.c -o greeting.exe
./greeting.exe
```

☑ **The program should ask for your name, then greet you by name.**

> **Note:** `scanf` with `%s` only reads up to the first space. So if you type "John Smith" it will only capture "John." Handling full names requires different techniques you'll learn later.

Bonus — Peek Under the Hood (Optional)

If you finish early, this section lets you see what actually happens when you compile a C program. That single `gcc` command you've been running actually performs **four separate steps**. You can run each one individually.

Make sure you're in your `c_practice` folder, then try these one at a time:

Step 1: Preprocessor

```
gcc -E hello.c -o hello.i
```

This expands all the `#include` directives, replacing `#include <stdio.h>` with the actual contents of that header file. Take a look:

```
nano hello.i
```

Scroll through it — you'll see it's HUGE compared to your tiny program. All of that came from `stdio.h`. Press **Ctrl+X** to exit.

Step 2: Compiler

```
gcc -S hello.c -o hello.s
```

This translates your C code into **assembly language** — the low-level instructions your CPU understands. Take a look:

```
nano hello.s
```

You'll see instructions like `movl`, `call`, `ret`. This is what your C code actually becomes. Press **Ctrl+X** to exit.

Step 3: Assembler

```
gcc -c hello.c -o hello.o
```

This converts the assembly into a **binary object file** — actual machine code, but not yet a complete program. You can't really read this one in a text editor (it's binary), but it exists.

Step 4: Linker

```
gcc hello.o -o hello.exe
```

This takes your object file and connects it to the system libraries (like the one that makes `printf` work), producing your final executable.

Inspect the Final Binary

```
objdump -d hello.exe | head -60
```

This **disassembles** your executable — turning the machine code back into assembly so you can read it. The `| head -60` part just shows the first 60 lines so you don't get overwhelmed.

The Full Pipeline

```
hello.c → [Preprocessor] → hello.i → [Compiler] → hello.s → [Assembler] → hello.o
→ [Linker] → hello.exe
```

When you run `gcc hello.c -o hello.exe`, all four of these steps happen automatically behind the scenes.

Completion Checklist

- [ ] GCC installed and `gcc --version` works
- [ ] `hello.exe` compiled and printed "Hello, World!"
- [ ] `math_test.exe` compiled and printed correct arithmetic output
- [ ] `greeting.exe` compiled and responded to your name input

When all four boxes are checked, show your screen to your instructor. You're done!