CSE 1111 — STRUCTURED PROGRAMMING LANGUAGE

Understanding the Structure of a C Program

From a blank file to a running program

hello.c

```c
#include <stdio.h>

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



TERMINAL

```
$ gcc hello.c -o hello
$ ./hello
```

Sheikh Tanvir Ahmed
Lecturer
Department of Computer Science and Engineering
United International University (UIU)

WHY IT MATTERS

Every C program has the same skeleton

You will write hundreds of C programs this trimester.

Almost all of them share the same outline.

Learn the outline once and every new program becomes easier to read.

THE SKELETON

```c
#include <stdio.h>

int main(void) {
    /* your statements */
    return 0;
}
```

Today: what each part does, and how this text file becomes a program the computer can run.

WHERE WE ARE GOING

By the end you can

1. Read a small C program and name its parts
2. Explain what #include and main() are for
3. Use statements, semicolons, braces and comments correctly
4. Describe preprocessing, compiling and linking
5. Compile and run a program from the terminal
6. Read a compiler error message and find the mistake

Read it before we explain it

C

```c
#include <stdio.h>

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

Which line prints the message? Which lines are just structure?

OUTPUT

```
$ ./hello
Hello, UIU!
```

The output is exactly what appears between the quotation marks, followed by a new line.

ANATOMY

Five parts you can name

```c
#include <stdio.h>

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

1. **Preprocessor directive.** Makes the printing tools available.
2. **main function.** Where the program starts running.
3. **Braces.** Mark the start and end of the function body.
4. **Statement.** One instruction, ending with a semicolon.
5. **return 0.** Tells the system the program finished normally.

A source file, and a header it borrows from

`hello.c`

- Plain text you write
- Ends with .c
- Any text editor works

`#include`



`stdio.h`

- A header file
- Describes printf, scanf and other input/output tools
- Part of the C standard library

ANGLE BRACKETS

`<stdio.h>` means “look in the standard places”. Your own header files use quotes: `"myfile.h"`.

The compiler does not know printf

NO #include

```c
int main(void) {
    printf("Hello\n");
    return 0;
}
```

CORRECT

```c
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```

COMPILER

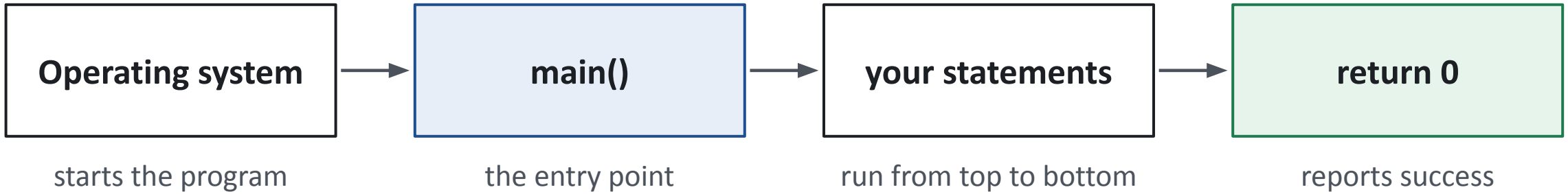
```
$ gcc bad.c
error: implicit declaration of function ‘printf’
 [-Wimplicit-function-declaration]
```

OUTPUT

```
Hello
```

MAIN()

Where execution begins



- A program must have exactly one `main`.
- `int` says main gives back a whole number.
- `void` says it takes no input here.
- `return 0` conventionally means “all went well”.

THE SHAPE

```c
int main(void) {
    /* ... */
    return 0;
}
```

One instruction, one semicolon

C

```c
#include <stdio.h>

int main(void) {
    printf("One\n");
    printf("Two\n");
    printf("Three\n");
    return 0;
}
```

Each highlighted line is a statement. They run in the order written.

OUTPUT

```
$ ./a.out
One
Two
Three
```

RULE

A simple statement ends with `;`. It tells the compiler where one instruction stops and the next begins.

BRACES AND BLOCKS

{ } group statements into one unit

```c
int main(void) {
    printf("start\n");
    {
        printf("inside a block\n");
    }
    return 0;
}
```

- Every { needs a matching }.
- A block can sit inside another block.
- The function body is itself a block.
- Indent each level so the nesting is visible.

> **REMEMBER**
>
> The compiler ignores indentation. Indentation is for the people reading your code, including you.

Notes for people, ignored by the compiler

C

```c
#include <stdio.h>

/* Prints the total of two marks.
   Written for practice. */
int main(void) {
    int a = 40, b = 35;   // two quiz marks
    printf("%d\n", a + b); // show their total
    return 0;
}
```

`/* ... */` can span many lines.

`// ...` runs to the end of the line.

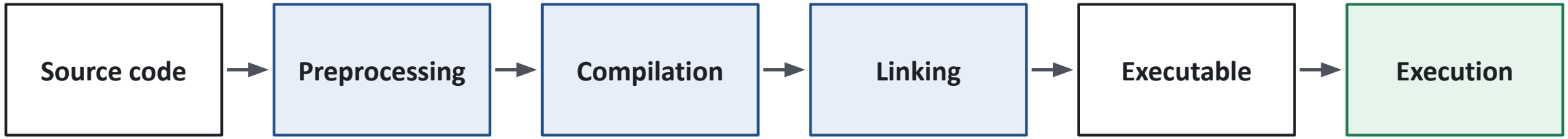
OUTPUT

```
$ ./a.out
75
```

GOOD HABIT

Explain **why** the code does something, not what a line obviously does.

Four steps before it runs



`hello.c`

expand #include and comments

translate C into machine code

join with library code

`hello`

the program runs

THE COMPILER DRIVER (gcc) RUNS THE MIDDLE THREE STAGES FOR YOU

Compiling checks your code and translates it. Execution is a separate step, and happens only if compiling succeeded.

What happens to hello.c

Stage	What it does	For hello.c
Preprocessing	Handles lines beginning with #, removes comments	pastes in the contents of stdio.h
Compilation	Checks syntax, translates C to machine code	reports errors such as a missing ;
Linking	Combines your code with library code	attaches the code behind printf
Execution	The operating system loads and runs the program	prints Hello, UIU!

If any stage fails, you get no executable.

Three kinds of problems, found at different times

Kind	Found	Example	What you see
Syntax error	while compiling	missing semicolon	compiler message, no executable
Runtime error	while running	dividing by zero	the program stops or misbehaves
Logic error	only by checking output	using + instead of *	runs fine, wrong answer

THINK

Which of the three is hardest to find? Why does the compiler help with only one of them?

Compile, then run

TERMINAL

```
$ gcc hello.c -o hello
$ ./hello
Hello, UIU!
```

- `gcc` is the compiler.
- `hello.c` is the source file.
- `-o hello` names the executable.
- `./hello` runs it from the current folder.

USEFUL FLAG

`gcc -Wall hello.c -o hello` asks the compiler to point out suspicious code, even when it compiles.

ON WINDOWS

The executable is usually `hello.exe`.

Start from the first error

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello\n")
    return 0;
}
```

COMPILER

```
$ gcc hello.c
hello.c:4:19:
error: expected ‘;’ before ‘return’
```

1. The file and line number tell you where to look.
2. The message says what the compiler expected.
3. The mistake can be reported on the line before the true problem.

Four frequent errors, and what gcc says

Mistake	Compiler message
Missing semicolon	`error: expected ‘;’ before ‘return’`
Wrong case: Main	`error: undefined reference to `main'`
Missing closing brace	`error: expected declaration or statement at end of input`
Missing #include	`error: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]`

PRACTICE

Name the parts

```c
#include <stdio.h>                  // A

int main(void) {                    // B
    int total = 75;                 // C
    printf("%d\n", total);          // D
    return 0;                       // E
}                                   // F
```

WHICH LETTER IS …

- the preprocessor line?
- the start of the function?
- a statement that prints?
- the line that reports success?
- the end of the function body?

Then: what is the output? (Hint: %d prints a whole number.)

PRACTICE

Find the syntax errors

broken.c

```c
#include <studio.h>

int Main(void) {
    printf("Welcome to UIU\n")
    return 0;
```

TASK

- There are four mistakes.
- Find each one.
- Say what the compiler would complain about.
- Write the corrected program.

> **STRATEGY**
>
> Read it like the compiler: top to bottom, one line at a time.

PRACTICE

Predict, order, explain

ORDER	Arrange: linking, preprocessing, execution, compilation.
EXPLAIN	What does `int main(void)` tell the compiler?
EXPLAIN	What happens if `#include <stdio.h>` is missing and the program uses `printf`?
THINK	Can a program compile without ever running? Can it run without compiling?

PREDICT THE OUTPUT

```
printf("A\n");
printf("B");
printf("C\n");
```

SUMMARY AND NEXT STEP

What to carry forward

- A C program is a **.c** text file with `#include`, `main()`, statements and braces.
- **gcc** preprocesses, compiles and links; then you run the executable.
- Syntax errors are caught when compiling; logic errors only by testing.

NEXT

How can a program store a student's age, name, or examination marks?

