COMP 2510

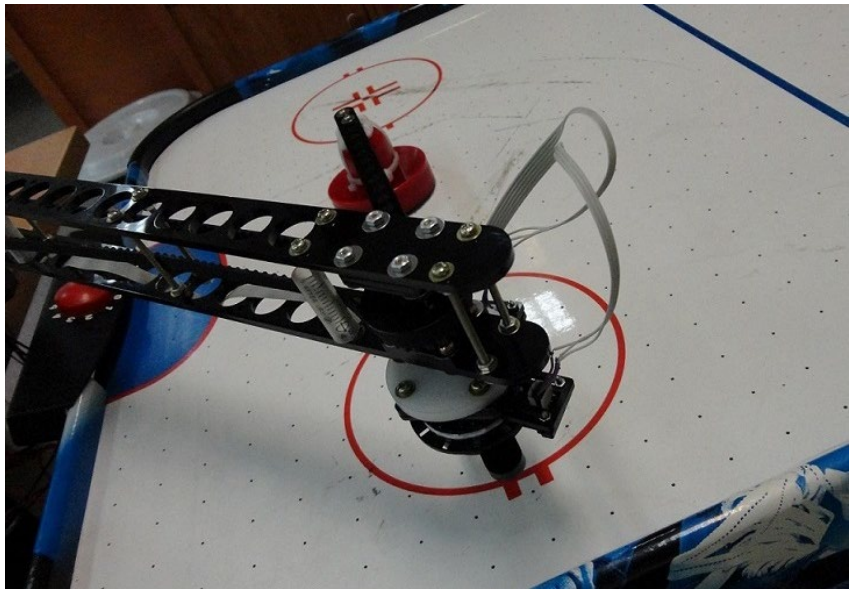
Week 1

Hesam Alizadeh

With special thanks to Frederic Guo

Introductions!

- Teacher / Computer Scientist / Student
  - M.Sc. Machine Learning & Robotics
  - M.Sc. Human Computer Interaction
  - 7+ years of professional experience in tech industry
- Contact: hesam_alizadeh@bcit.ca
- Course Outline: COMP 2510 – 202601 (BBY)
- Office hours: Online by appointment. Book using Calendly



C Programming language

- Created in 1972 by Dennis Ritchie
  - Initially developed “to construct utilities running on Unix”
  - Later used to rewrite the Unix kernel
- High-level language
  - Each statement corresponds to several machine language instructions (1-to-many correspondence).
- Compiled
  - Provides low-level access to memory
  - Faster runtime performance
- Imperative and Procedural

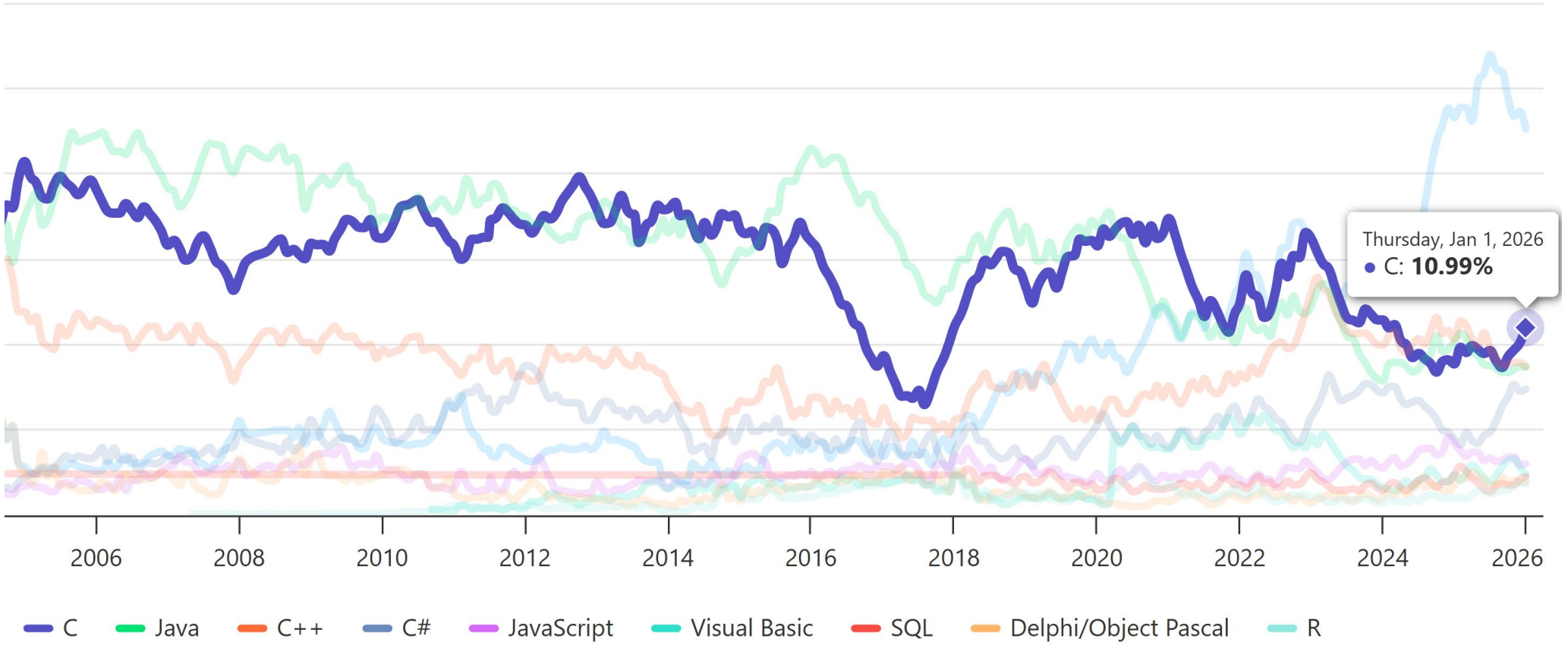
Dennis Ritchie

1941 - 2011

Won Turing Award (1983) alongside Ken Thompson for development of the UNIX operating system

> **C is quirky, flawed, and an enormous success**. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.
>
> — Dennis Ritchie

Popularity of C (as of Jan. 2026)

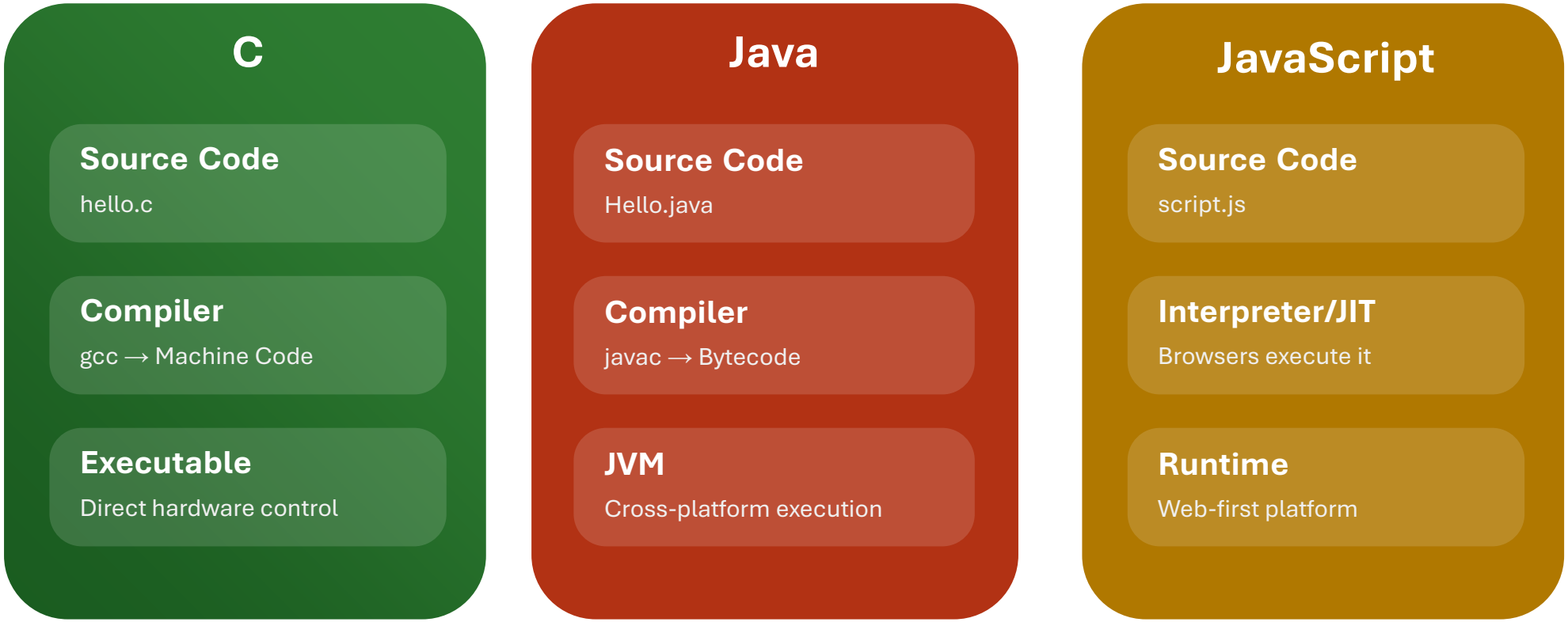


Source: TIOBE Index - TIOBE

Why Learning C?

- Direct memory management and control
  - You can write **highly efficient** code that maximizes CPU and RAM usage.
- The foundation for “modern" languages. See: Genealogical tree of programming languages
  - Knowing C helps you **understand** higher-level languages features like garbage collection, etc.
- Wide variety of use
  - **Operating Systems:** The Linux kernel, macOS, and Windows core components are heavily written in C.
  - **Interpreters and Databases:** MySQL, PHP, and the CPython interpreter are built in C.
  - **Embedded Systems:** C is the primary language for firmware and IoT devices

How your code runs



What do you think C# and python do?

Our First C Program

```c
#include <stdio.h>                  /* Link a library */

int main()                          /* The starting point */
{
    printf("Hello World!");         /* a print statement */

    return 0;                       /* return statement
                                       to end the program */
}
```

Java Equivalent:

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

- The main() function is always the first function called.
- Statements can be inside of a function OR outside of a function.

Let’s dive in!

If you haven’t set up your environment.

Please do so now by following these instructions:
Setting Up CLion & GCC

Console I/O in C

`#include <stdio.h>` gives access to standard input/output

`printf()`: displays formatted output to the standard output stream, which is typically your console screen

`scanf()`: Reads **formatted data from the standard input** (usually the keyboard) and store it in specified variables.

- `scanf` returns how many items it successfully read. We can use it to validate the input.

For more info: https://documentation.help/C-Cpp-Reference/scanf.html

Console I/O in C

1) Read in one number:

```
int num;
scanf ("%d", &num);
```

2) Read in multiple numbers

```
int num1, num2;
scanf ("%d %d", &num1, &num2);
```

3) Read in a char

```
char one_char;
scanf ("%c", &one_char);
```

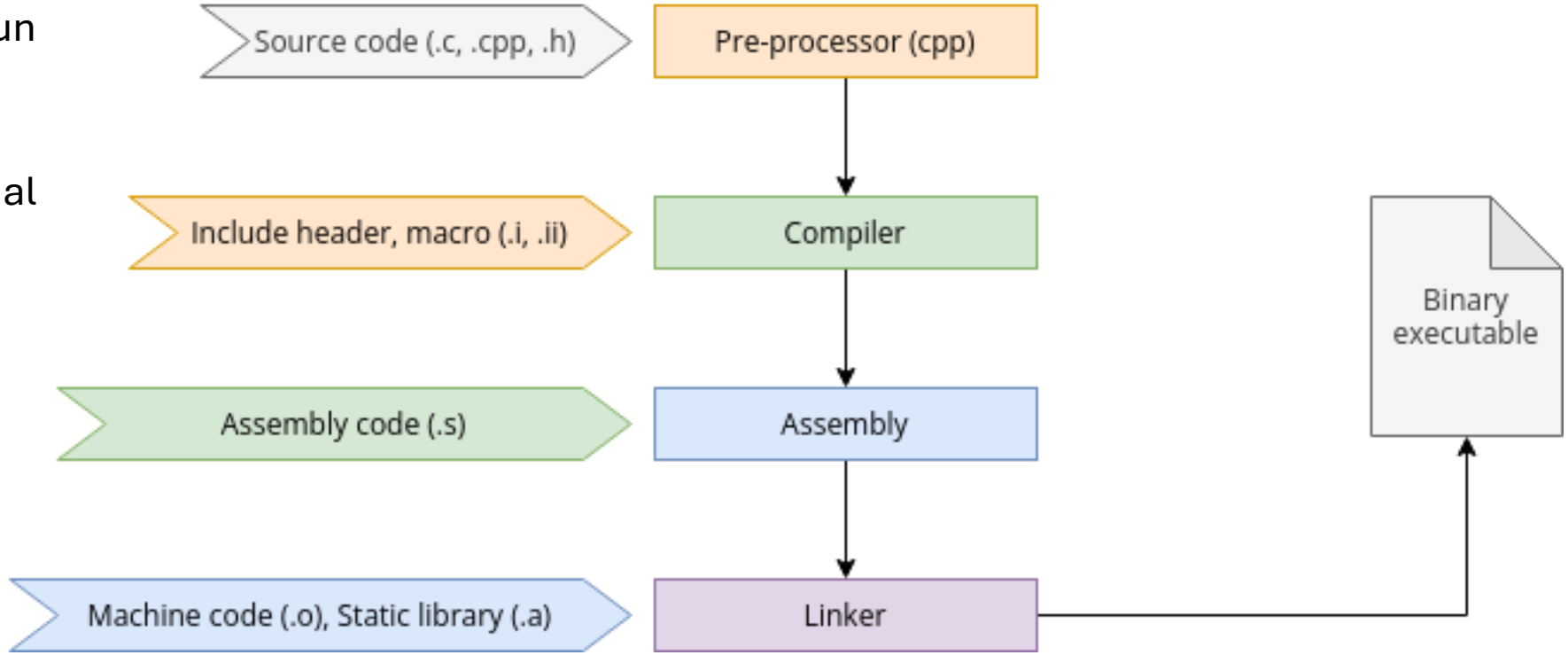
- %d is a control character and is used for scanning decimal integers.
- Notice the **&** character before d. This means that we are passing the address of x. We will learn about this concept in a few weeks.

For more info: https://documentation.help/C-Cpp-Reference/scanf.html

The GCC Compilation Process

The executable code is run directly by the Operating System (OS).

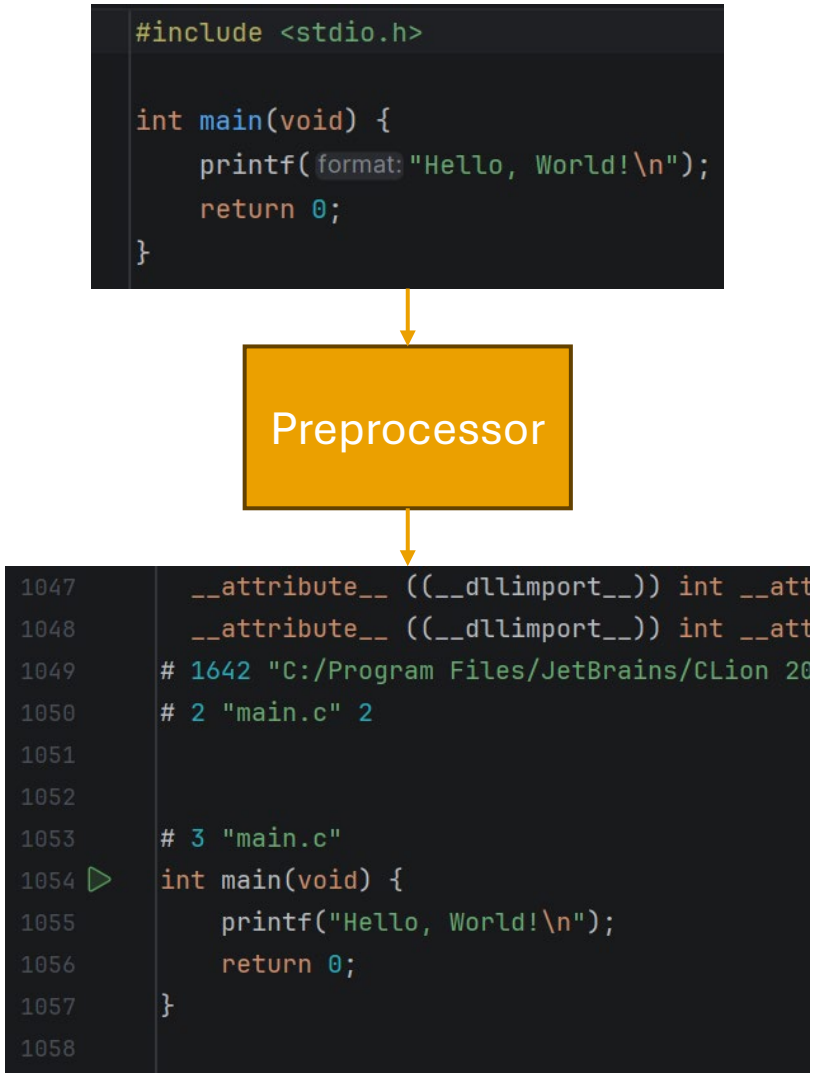
It does not require a virtual machine or interpreter.



Source: A programmer's guide to GNU C Compiler

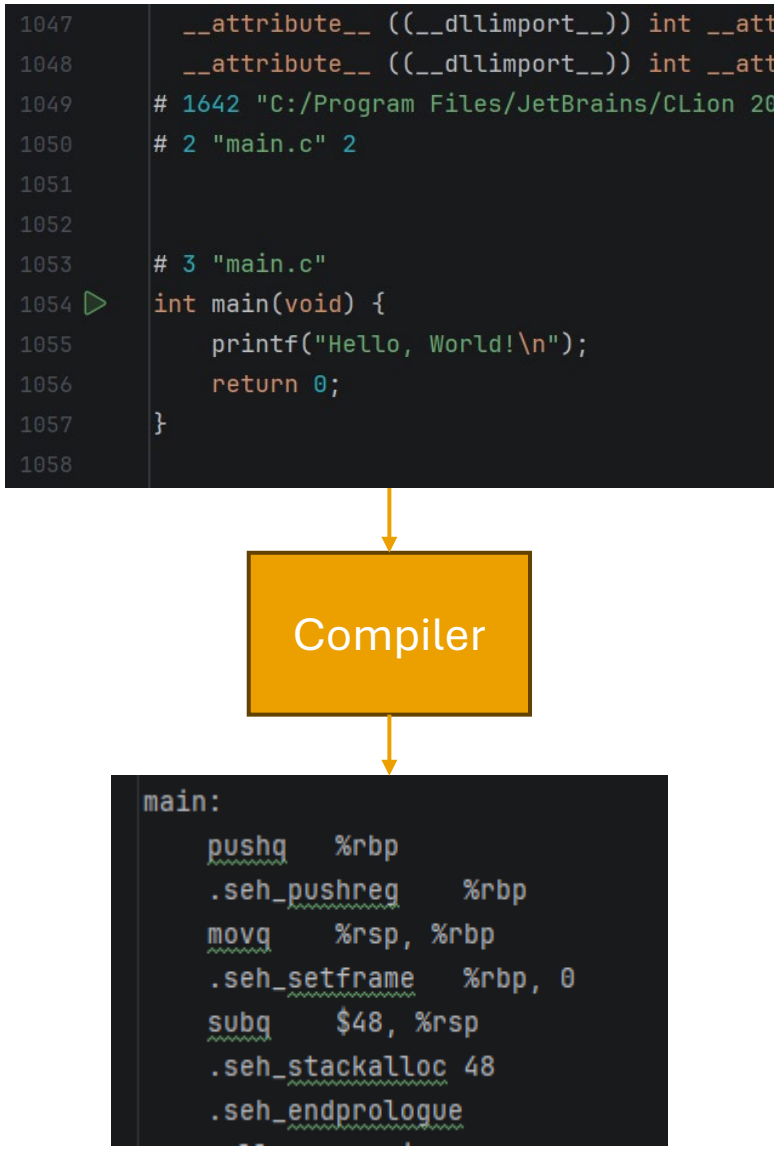
Preprocessor (cpp)

- **Input:** Source code file (.c, etc.).
- The preprocessor follows directives starting with # to modify the source before actual compilation begins. It's essentially **a text substitution tool**.
- When you use `#include <library.h>`:
  - The preprocessor finds the specified **header file** and copies its contents into your source code file.
  - Header files contain **declarations** (function prototypes, variable definitions, macros) that tell the compiler how to use the functions, but **not the actual implementation**
- **Output:** A preprocessed source file (.i file).
- **GCC Command (to stop here):** `gcc -E filename.c`



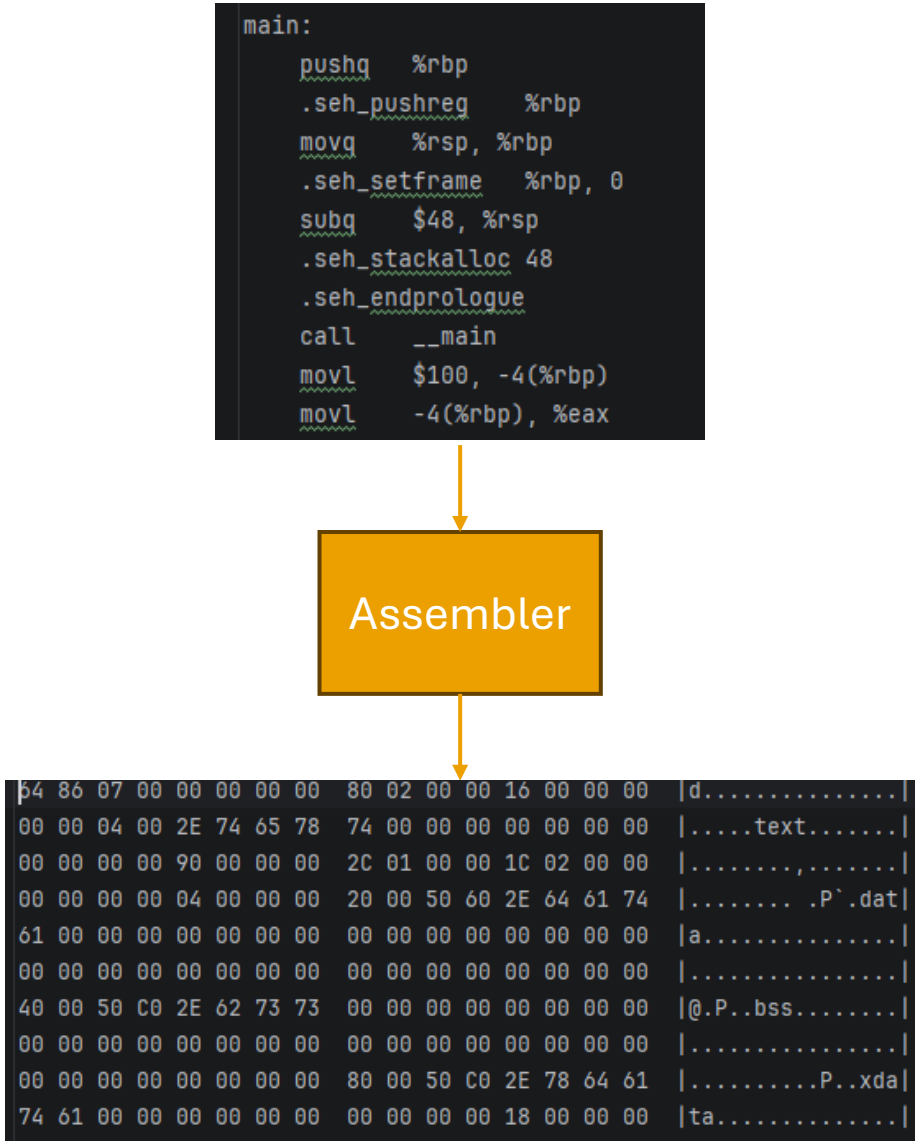
Compiler

- **Input:** Preprocessed source code (.i file).
- The **compiler** translates the clean, preprocessed C code into assembly language, which is a low-level, human-readable language **specific to the target processor architecture**.
- **Key Tasks**:
  - Checks for **syntax errors**
  - Performs various **code optimizations**.
- **Output**: An assembly file (.s file).
- **GCC Command (to stop here)**: `gcc -S filename.c`



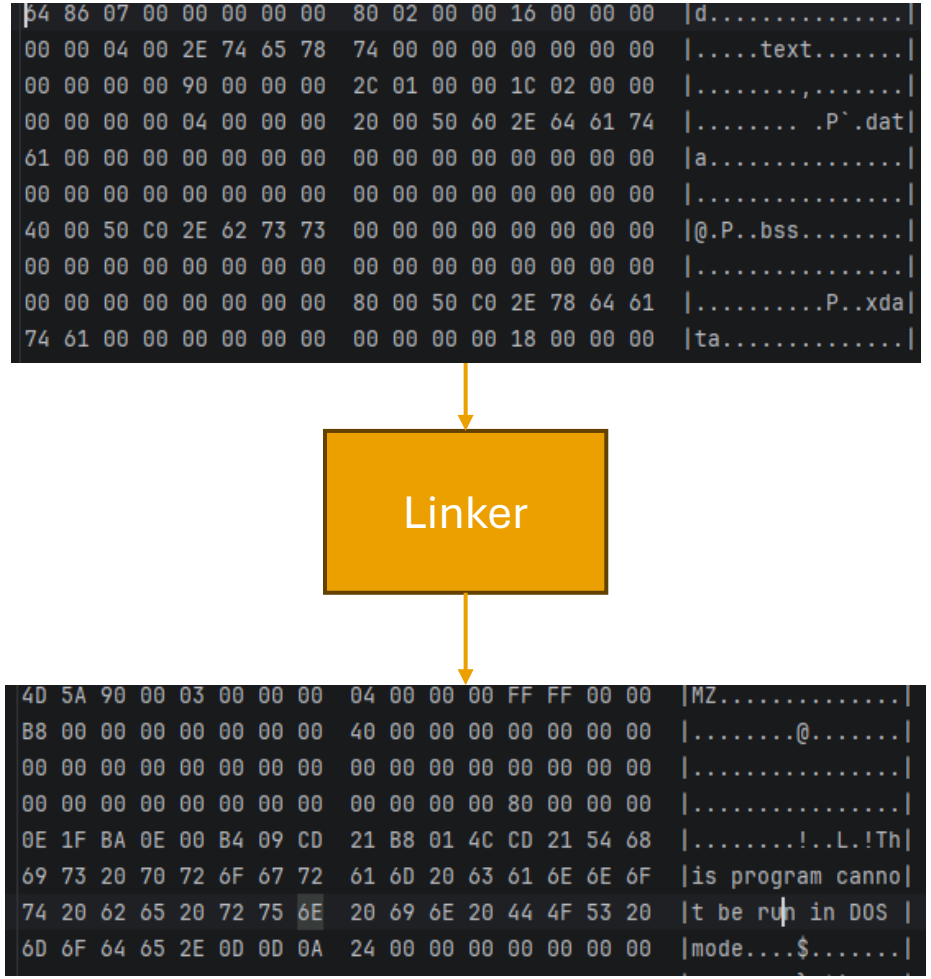
Assembler

- **Input**: Assembly code file (.s file).
- The assembler translates the assembly instructions into machine code (binary format).
- The resulting object file contains the actual instructions the processor can run, but function calls to external libraries are not yet resolved.
- **Output**: An object file (.o file).
- **GCC Command (to stop here)**: `gcc -c filename.c`



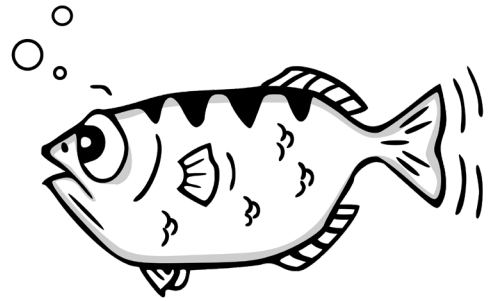
Linker

- **Input**: Object files (.o files) and libraries (static or dynamic).
- The linker combines all the object files and necessary library code into a single, executable program.
- **Key Tasks**:
  - Resolves all function calls (e.g., to printf()) by finding their actual machine code definitions and assigning final memory addresses.
  - Adds necessary start-up and tear-down code required by the operating system.
- **Output**: An executable file
- **GCC Command (full process):** `gcc filename.c`



The GNU Project Debugger (GDB)

- A powerful command-line tool used to inspect what is happening "inside" a program while it executes or what it was doing at the moment of a crash.
- **Key Capabilities:** Set breakpoints, step through code line-by-line, and modify variable values at runtime.
- **Setup (Standard Flag):** Programs must be compiled with the **-g** flag to include debugging symbols.
- **Warning Flags:** Catch errors before runtime by adding flags like **`-Wall -Wextra -Werror`** to your build command to treat common issues as critical errors.
- **Runtime Safety (Sanitizers):** You can use **`-fsanitize=address`** (AddressSanitizer) to automatically detect memory leaks and buffer overflows as your program runs.



GDB Quick Reference

Starting & Stopping

- `run` (or r): Start your program from the beginning.
- `quit` (or q): Exit the GDB environment.

Setting Breakpoints

- `break` [file:line/func] (or b): Pause execution at a specific location.
- `info break`: List all active breakpoints.
- `delete` [n]: Remove breakpoint number n.

Stepping Through Code

- `next` (or n): Execute the next line of code, **stepping over** function calls.
- `step` (or s): Execute the next line, **stepping into** function calls.
- `continue` (or c): Resume execution until the next breakpoint.

Inspecting State

- `print` [var] (or p): Show the current value of a variable.
- `info local`: Show all local variables
- `backtrace` (or bt): Display the program stack (shows which functions called the current one).
- `list` (or l): View the source code around the current line.

Main types of errors

- **Syntax Errors (Compile-time):** Violations of C's grammatical rules (e.g., mismatched **{ }** or missing **;**). These are caught by the compiler.
- **Runtime Errors:** Illegal operations that occur during program execution (e.g., **division by zero** or **segmentation faults**). The program compiles but crashes when run.
- **Logical Errors:** The program runs without crashing but produces incorrect results due to flawed reasoning (e.g., using + instead of * in a formula or **infinite loops**).
- **Linker Errors:** Occur during the linking phase when the executable cannot be created, often due to missing function definitions.
- **Semantic Errors:** Statements that are syntactically correct but have no meaning to the compiler, such as assigning a string to an integer or attempting a + b = c;.

Time for another dive!

1. Compile the code. Make sure to include the debugging & warning flags:

   ```
   gcc -g -Wall -Wextra -Werror main.c -o program
   ```

2. Run gdb

   ```
   gdb program
   ```

3. Set a breakpoint at the beginning of main():

   ```
   break main
   ```

4. Run the program:

   ```
   run
   ```

5. List the source code around the current line:

   ```
   list
   ```

6. Next line:

   ```
   next
   ```

7. Display local variables:

   ```
   info local
   ```

8. Print certain variable:

   ```
   print a
   ```

Common Basic Data types

Type	Value Range	Usage
int	-2,147,483,648 to 2,147,483,647	Whole numbers
unsigned int	0 to 4,294,967,295	Positive integers
char	-128 to 127 (signed), 0 to 255 (unsigned)	Characters
float	±3.40282e+38	Decimal numbers
double	±1.79769e+308	Precision decimal numbers
void	N/A	No value / generic pointers
* bool (_Bool)	0 (false), 1 (true)	Boolean logic

Note: Sizes vary by platform/compiler; use `sizeof(type)` on your machine.

* Bool type is recently added in C23 (the latest open standard for the C language). So, Make sure to set the standard to C23 in your IDE / terminal. For examples:

```
gcc -Wall -Wextra -std=c23
foo.c
```

Variables

One way to think about variables **“A human-readable name that refers to some data in memory.”**

- To declare a variable:

```
varType varName;
```

- For naming, you can use any characters in the range 0-9, A-Z, a-z, and underscore for variable names, with the following rules:
  - You can’t start a variable with a digit 0-9.
  - Names starting with _ may be reserved (especially _X and __X). Avoid leading underscores in your own identifiers.
  - You can’t start a variable name with an underscore followed by a capital A-Z.

Reference: Beej's Guide to C Programming - 3 Variables and Statements

Arithmetic Operations

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	`f + 7`
Subtraction	–	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Operator Precedence

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they're evaluated left to right.
* / %	Multiplication Division Remainder	Evaluated second. If there are several, they're evaluated left to right.
+ -	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Increment and Decrement Operators in C

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Escape Sequence

Escape sequence	Description
\n	Newline. Position the cursor at the beginning of the next line.
\t	Horizontal tab. Move the cursor to the next tab stop.
\a	Alert. Sound the system bell.
\\	Backslash. Insert a backslash character in a string.
\"	Double quote. Insert a double-quote character in a string.

For more: https://en.wikipedia.org/wiki/Escape_sequences_in_C

Assignment Statements

Operator	Operator name	Example	Description	Equivalent of
`=`	basic assignment	`a = b`	**a** becomes equal to **b**	N/A
`+=`	addition assignment	`a += b`	**a** becomes equal to the addition of **a** and **b**	`a = a + b`
`-=`	subtraction assignment	`a -= b`	**a** becomes equal to the subtraction of **b** from **a**	`a = a - b`
`*=`	multiplication assignment	`a *= b`	**a** becomes equal to the product of **a** and **b**	`a = a * b`
`/=`	division assignment	`a /= b`	**a** becomes equal to the division of **a** by **b**	`a = a / b`
`%=`	modulo assignment	`a %= b`	**a** becomes equal to the remainder of **a** divided by **b**	`a = a % b`
`&=`	bitwise AND assignment	`a &= b`	**a** becomes equal to the bitwise AND of **a** and **b**	`a = a & b`
`|=`	bitwise OR assignment	`a |= b`	**a** becomes equal to the bitwise OR of **a** and **b**	`a = a | b`
`^=`	bitwise XOR assignment	`a ^= b`	**a** becomes equal to the bitwise XOR of **a** and **b**	`a = a ^ b`
`<<=`	bitwise left shift assignment	`a <<= b`	**a** becomes equal to **a** left shifted by **b**	`a = a << b`
`>>=`	bitwise right shift assignment	`a >>= b`	**a** becomes equal to **a** right shifted by **b**	`a = a >> b`

Reserved keywords in C

alignas (C23)	extern	sizeof	_Alignas (C11)(deprecated in C23)
alignof (C23)	false (C23)	static	_Alignof (C11)(deprecated in C23)
auto	float	static_assert (C23)	_Atomic (C11)
bool (C23)	for	struct	_BitInt (C23)
break	goto	switch	_Bool (C99)(deprecated in C23)
case	if	thread_local (C23)	_Complex (C99)
char	inline (C99)	true (C23)	_Decimal128 (C23)
const	int	typedef	_Decimal32 (C23)
constexpr (C23)	long	typeof (C23)	_Decimal64 (C23)
continue	nullptr (C23)	typeof_unqual (C23)	_Generic (C11)
default	register	union	_Imaginary (C99)
do	restrict (C99)	unsigned	_Noreturn (C11)(deprecated in C23)
double	return	void	_Static_assert (C11)(deprecated in C23)
else	short	volatile	_Thread_local (C11)(deprecated in C23)
enum	signed	while	

Reference: C keywords

Decision Making

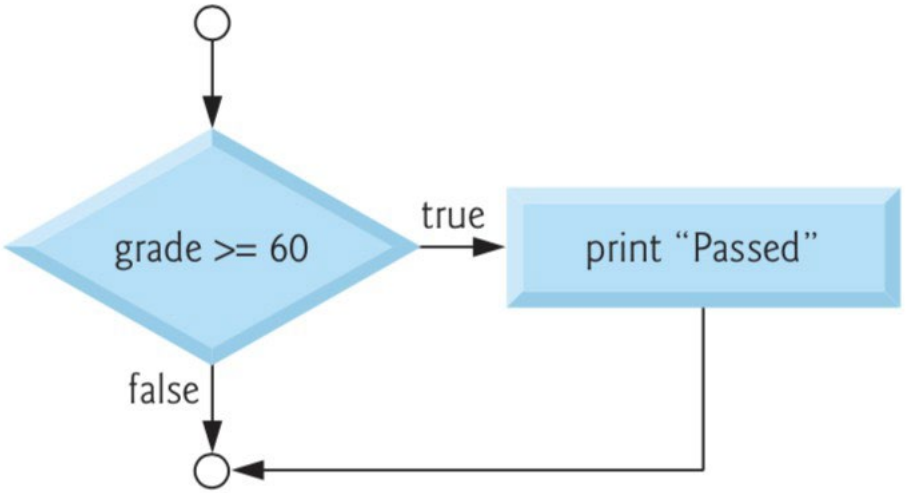
Algebraic equality or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Basic If Statement

```
if ( condition)

   one statement;

    /* only one statement to do if the condition
   fulfilled */
```

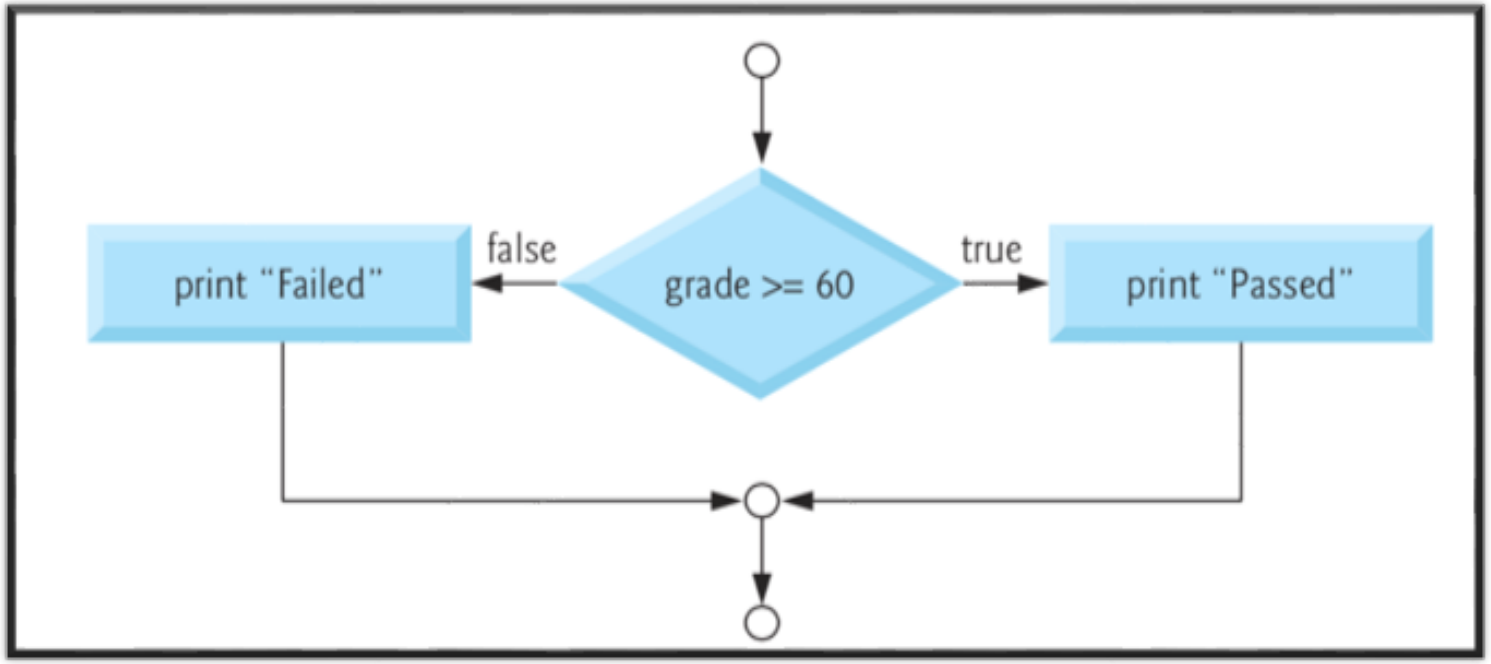
Vs

```
if ( condition) {

   statements
    /* one or multiple statements to do when the
   condition got fulfilled */

}
```



If - else statement

```
if ( grade >= 60) {
    printf("passed\n");
}
else {
   printf("failed\n");
}
```



CONDITIONAL OPERATOR ? :

- Use case #1: `condition ? statement1 : statement 2 ;`

```
grade >= 60 ? printf( "Passed\n" ) : printf( "Failed\n" );
```

- Use case #2: `condition ? value 1 : value 2 ;`

```
printf( "%s\n", grade >= 60 ? "Passed" : "Failed" );
```

Decision Making (Summary)

- Conditions use relational operators: == != < <= > >=
- Combine conditions with logical operators: && || !
- In C: 0 is false, non-zero is true
- Always use braces {} for clarity (even for one line)
- Difference with Java:
  - **Java** requires the condition inside the if statement to be a strict **boolean** expression.
  - **C** evaluates an expression as true if it is **non-zero** and false if it is **zero**.

```c
if (age < 0) {
    printf("Age can't be negative.\n");
} else if (age < 18) {
    printf("Minor\n");
} else {
    printf("Adult\n");
}
```