



# Introduction to Programming and Problem Solving

**Behram Khan**





# Lecture Overview

- In this first lecture we will cover the fundamental theoretical concepts that form the backbone of programming.
- This session focuses solely on theory, providing a solid foundation for understanding computers and programming
- Involving examples and interactive session to develop problem solving



# Learning Objectives

By the end of this lecture, students will be able to:

- Know about programming languages
- How high level languages are translated to low level
- Understand basic hello world program structure.



# Recap

- Timing Signal
- Control Signal
- CIR Register
- IBR Register



# Recap

- The CPU reads the instruction  $5 + 3$  and stores it in **CIR**.
- The **MAR** is set to the memory addresses where 5 and 3 are stored.
- The **MDR** retrieves the values 5 and 3 from those addresses.
- The **ACC** performs the addition  $(5 + 3)$  and stores the result (8).
- The **PC** moves to the next instruction's address.
- The **IBR** prepares the next instruction, allowing it to be quickly accessed once the current instruction is complete.



# Recap

- What is Logic?
- Transmitted
- General Purpose and Specific Purpose Computers



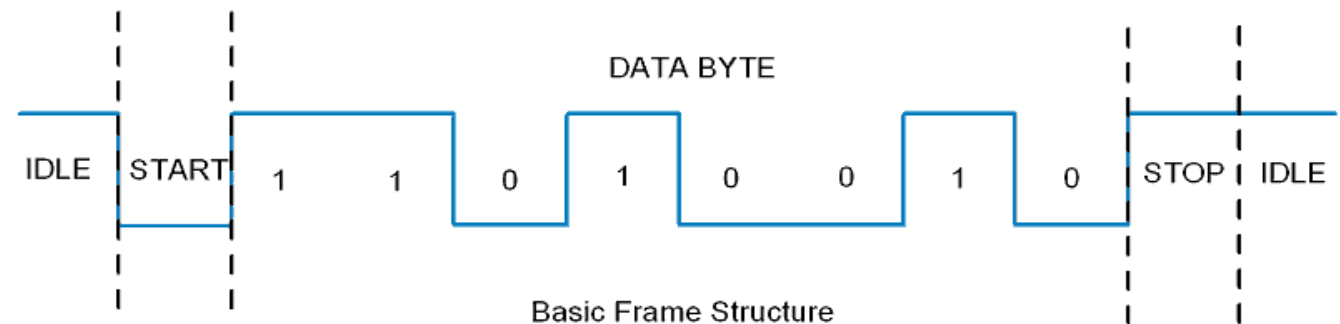
# Recap : Unit Of Memory

- What is Bit?

Smallest unit of data that a computer can process and store.

- 1,0
- Voltage States

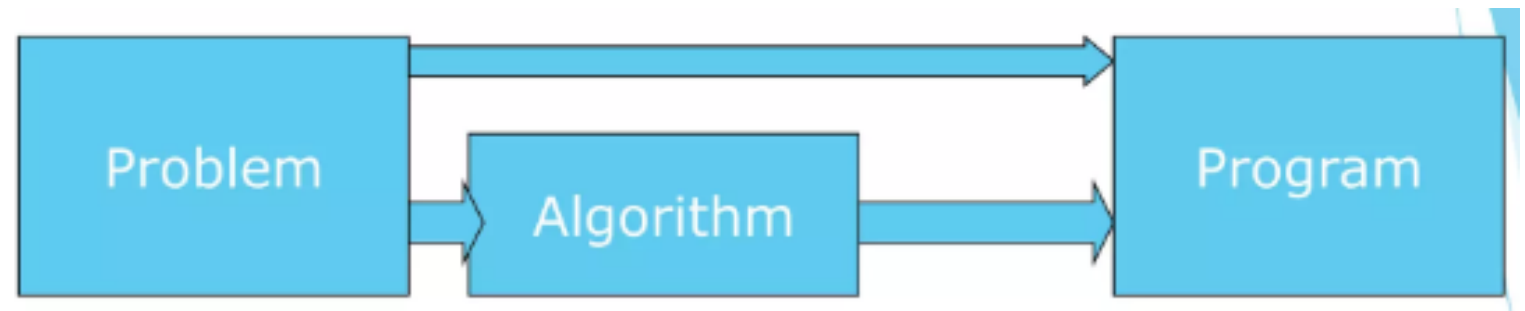
Unit	Abbreviation	Size (Bytes)
Bit	b	1/8
Byte	B	8 bits
Kilobyte	KB	1024 bytes
Megabyte	MB	1024 KB
Gigabyte	GB	1024 MB
Terabyte	TB	1024 GB





# Algorithm

- Step by step instructions to solving a problem
- Sequence of English statements
- It is not a program
- Algorithm can be expressed as a pseudo code or a flowchart.







# Algo : Average of two numbers

1. Input two numbers
2. Add the numbers
3. Divide the result by 2
4. Display the result



# Algo : Student has passed or failed

1. Input Marks
2. If( Marks are greater than or equal to 40)
3. Then set grade to pass
4. Else set grade to fail
5. Display the grade



# Algo : Write algorithm for grading system

Marks range	Grade
$\geq 80$	A
$\geq 70 \ \& \ < 80$	B
$\geq 60 \ \& \ < 70$	C
$\geq 50 \ \& \ < 60$	D
$< 50$	F








# Algo : Student has passed or failed

1. Get the student's marks.
2. Check if the marks are between 80 and 100, inclusive.  
If true, assign the grade "A".
3. If not, check if the marks are between 70 and 79, inclusive.  
If true, assign the grade "B".
1. If not, check if the marks are between 60 and 69, inclusive.  
If true, assign the grade "C".
1. If not, check if the marks are between 50 and 59, inclusive.  
If true, assign the grade "D".
1. If not, assign the grade "F".



# Flowchart

- Graphical representation of an algorithm.
- Helps see the flow of logic

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision



# Flowchart

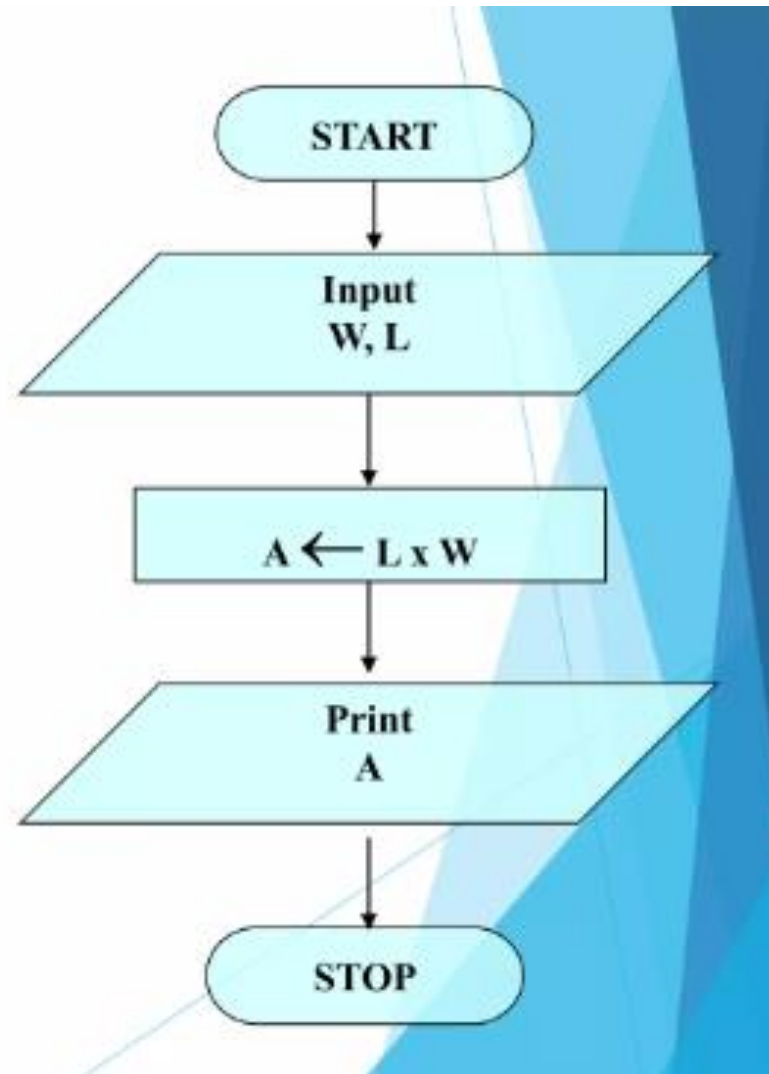
Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.

## Algorithm

- ▶ Step 1: Start
- ▶ Step 2: Take Width and Length as input
- ▶ Step 3: Calculate Area by  $\text{Width} * \text{Length}$
- ▶ Step 4: Print Area.
- ▶ Step 5: End




# Flowchart





# Memory

- Temporary storage place
- Volatile vs Non-Volatile (RAM vs ROM)
- Memory has addresses

MEMORY	
0X1000	4000
0X1004	 Calculator
0X1008	
0X100C	
0X1010	
0X1014	
0X1018	
ADDRESS	VALUE

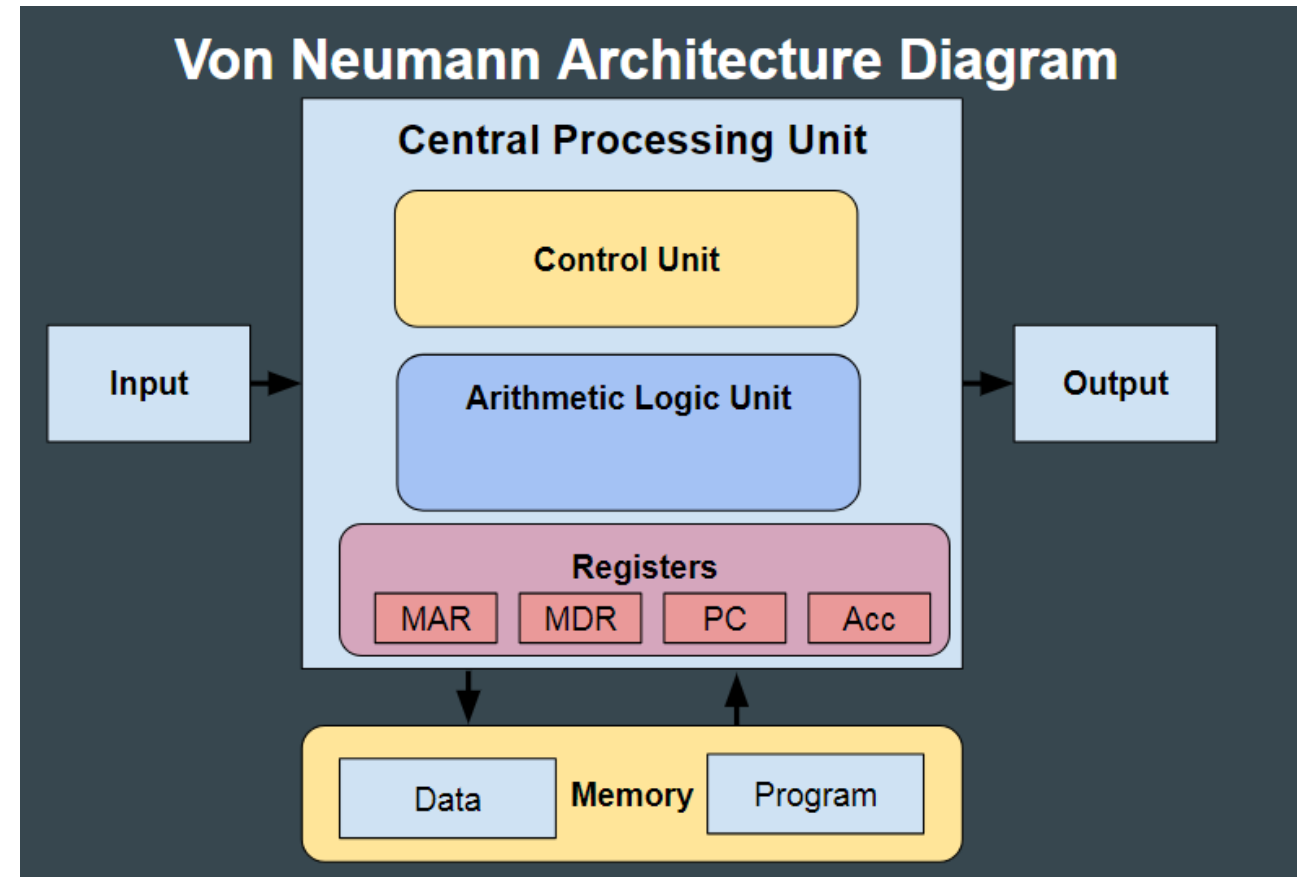




# Working of Architecture

## Student in class Analogy

- Human as CPU
  - CU
  - ALU
  - Registers
- Hands as Buses
- Bag as storage
- Handle as RAM





# How is Hello World Displayed?





outputs > C HelloWorld.c > main()

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

```
2
```

Tabnine | Edit | Test | Explain | Document | Ask

```
3  int main() {
```

```
4      printf("Hello World");
```

```
5      return 0;
```

```
6  }
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

NUGET

- PS D:\Coding stuff\C language\1st Sem B2024\testing> &cher.exe' '--stdin=Microsoft-MIEngine-In-npzg5pci.fln'icrosoft-MIEngine-Pid-hpa3efqv.idf' '--dbgExe=C:\msys64Hello World
- PS D:\Coding stuff\C language\1st Sem B2024\testing> █



# Programming Languages

**Programming languages** define a set of instructions for the CPU (Central Processing Unit) for performing any specific task.

2 types

- Low level Language
- High Level Language



# Low Level Languages

- Computer only understands Machine Code or Binary Language 1 and 0's.
- Machines can easily understand it
- Harder to write
- They are not very easy to understand
- Not Portable

**Assembly Language** : Symbolic representation of a computer's machine code.

ADD, SUB, MOV, STR



# High Level Languages

- Closer to human languages and are easier to read and write.
- Can run on multiple platforms
- C, C++, Java, Python



# Difference?

Level of abstraction(hiding unnecessary details)

Assembly Language (x86):

Code snippet

```
section .data
    message: db "Hello, world!", 0

section .text
    global _start

_start:
    mov ah, 0x09
    mov dx, message
    int 0x21

    mov ah, 0x4C
    mov al, 0x00
    int 0x21
```



C

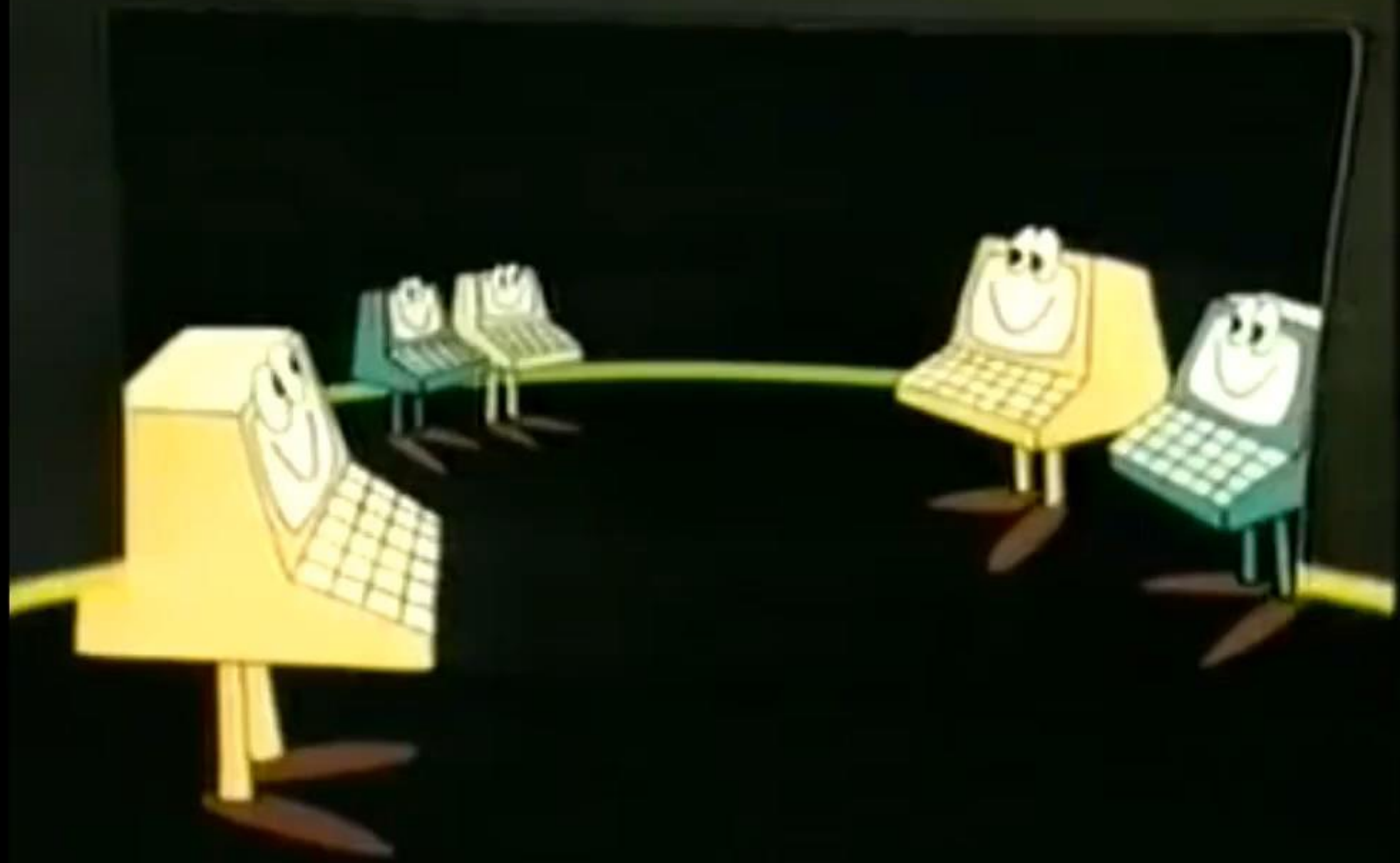
```
#include <stdio.h>

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



Now how will computer understand high level language  
as it only understands machine language (1 and 0's)?  
**WE TRANSLATE IT INTO LOW LEVEL LANGUAGE**







# Interpreter

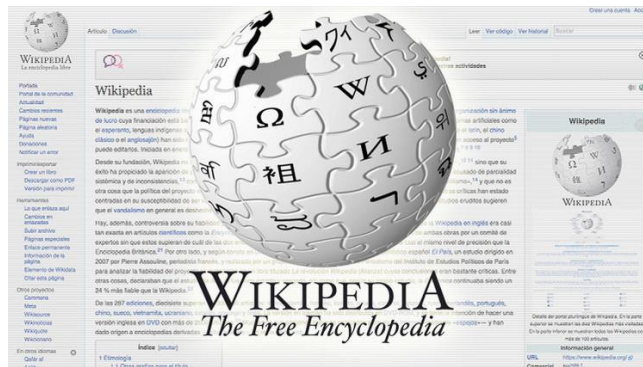
An interpreter reads the source code line by line or statement by statement and executes it directly.

- There is no intermediate object code or executable file generated.
- When the program runs
  1. the interpreter processes the source code line by line
  2. The interpreter reads one line of code
  3. It translates that line into machine code and executes it immediately
  4. Then it moves to the next line and repeats the process.
- The interpreter must be present to execute the code, meaning that the user must have the interpreter installed to run the program.



# Interpreter

- Have to share source code with interpreter every time.



Make a copy



Have Interpreters installed

Displayed on screen  
(Can be slow due to slower interpreter)



# Compiler

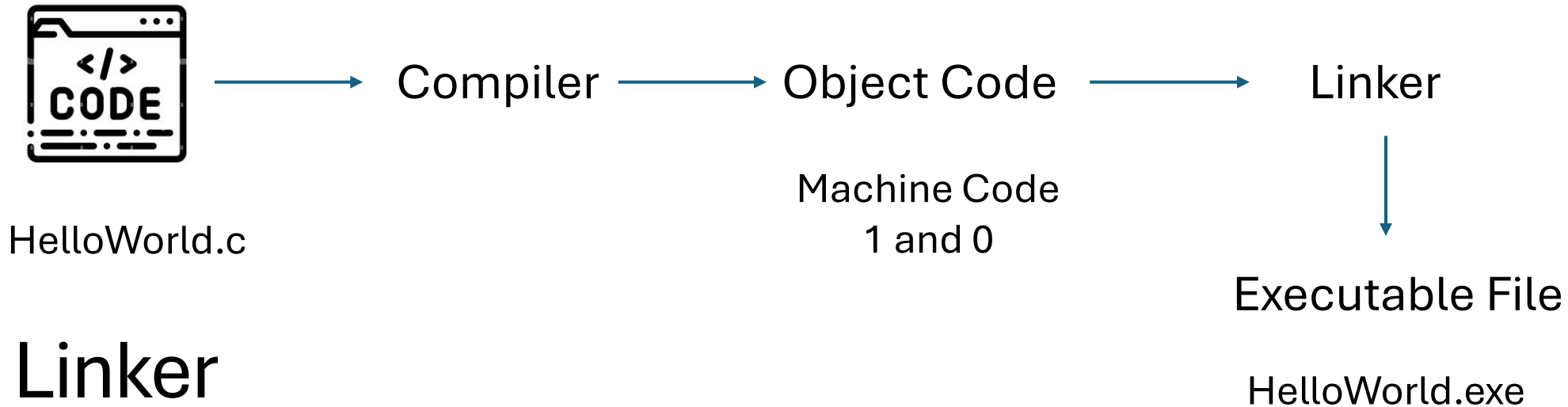
A compiler takes the entire source code of a program written in a high-level language and translates it into an object code or machine code.

The output of the compilation process is typically a standalone executable file that can be run on the target machine without the need for the original source code or the compiler.

- Compilation is done before the program runs
- Any changes to the source code require recompilation.
- Specific to platform (Architecture), Not cross independent



# Compiler



## Linker

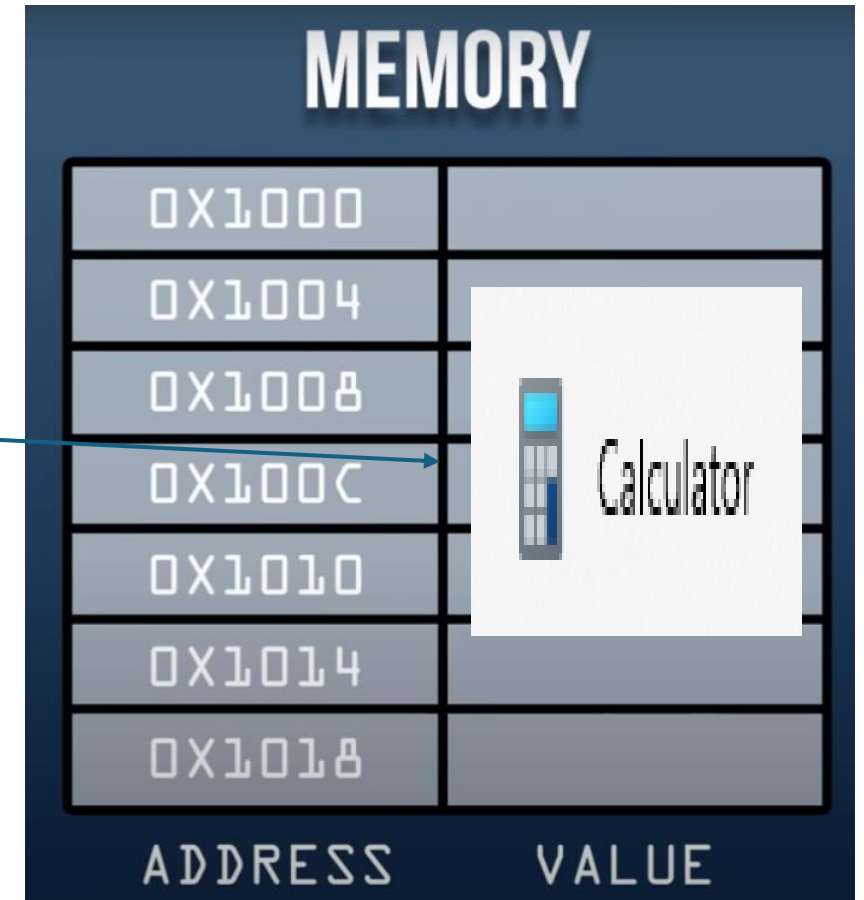
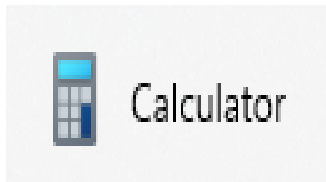
Converts Object Code into Executable File



# Loader

Loads file into memory when we execute it

Executable File → Memory/RAM



Allocates memory to the program



# Why Program In C Language



# C Language

The C programming language is a procedural (step by step, top to bottom) and general-purpose.

- **Foundation of Modern Languages:** C serves as the foundation for many programming languages (like C++, C#, and Objective-C). Understanding C helps grasp the concepts of other languages more easily.
- Need compiler GCC
- Widely used for system programming, embedded systems, Game Development





# Understand Program

1. #
2. Include
3. Header File
4. STDIO.H
5. Main Function
6. Body
7. Printf
8. String in double quotes (apostrophes)
9. Semicolon/Terminator

```
outputs > C HelloWorld.c > main()
1  #include <stdio.h>
2
   Tabnine | Edit | Test | Explain | Document
3  int main() {
4      printf("Hello World");
5      return 0;
6  }
```



# # and Include

- # is called **preprocessor directive**
- It tells the compiler to perform some special task
- Include **keyword** give command/instruction to include the <stdio.h> header file in our program.
- #include statement is like a toolkit, before we start doing our work, we take our tools out of it.
- We are telling the compiler to open a "toolbox" that contains important tools (functions) needed for input and output tasks, such as displaying text on the screen or reading input from the keyboard.



# Header file

- Header file is a file which contains functions
- $f(x) = 2x + 1$ .
- This function takes an input (x) and performs an operation on it (multiply by 2 and add 1) to produce an output.
- In programming, a function is similar. It takes input values, performs a series of operations, and returns an output value.
- `<stdio.h>` stands for Standard Input Output library
- Stdio contains functions and instructions to display output and take input.



# Main Function

- A Function which tells the compiler where our program starts from.
- Every C program must have a main function/starting point of the program.
- We write `int` before `main` to get error message.
- We add `()` parenthesis after `main` to tell compiler this is a function.
- We add `{` and `}` to tell compiler the starting and ending point of our program.
- This is just the way functions are written.



# Main Function

```
Int main()  
{  
    if (batterHitsBall)  
        return 1;  
    if (batterIsOut)  
        return 0;  
    if(ballIsWide)  
        return 2;  
}
```

**A Baller balling**

0 means good for baller  
1 means batter hit ball  
2 means ball is wide

0 is usually success  
1 or non zero value is  
usually error



# Terminator

- Tells compiler where our statement ends

```
outputs > C HelloWorld.c > main()
1  #include <stdio.h>
2
   Tabnine | Edit | Test | Explain | Document
3  int main() {
4      printf("Hello World");
5      return 0;
6  }
```



# Q/A?

## Recap:

- Programming Languages
  - High level
  - Low Level
- Interpreter and Compiler
- C Program basic structure



THANK YOU!