# Software

## How programs are prepared and run

# What we will learn

- Program → Running process
- Toolchain in detail
- Assembly, C, and mixing languages
- OS role in execution
- Design trade-offs

# Section 1: Software as a System

## What we will see

- Software layers
- Why tools exist
- Separation of responsibilities

## Software: beyond “apps”

- Not just applications
- Includes:
  - Tools
  - Runtimes
  - OS services

## Software layers

- **Application software**: Programs written to solve user problems
  - Firefox, Microsoft Word or your own program
- **System software**: Software that manages hardware and provides services to applications
  - Kernal (Linux or Windows), File System
- **Utility programs**: Tools that help develop, maintain, or analyze software
  - Development tools (compilers, debuggers). Example for compilers are gcc, Turbo C
- **Firmware**: Low-level software stored in non-volatile memory that starts or controls hardware
  - BIOS

## Think Deeper 🤔

- Why separate software into layers?
- What breaks if everything is “one big program”?

## Section 1 summary

- Software is a layered system
- Each layer has a role
- Separation reduces complexity

# Section 2 : Program Lifecycle

## What we will see

- From source code to execution
- Files created at each stage
- Who does what

# Example

```
hello.c
↓ compiler
hello.o
↓ linker
hello
↓ loader
process in memory
```

- Same program, different forms
- Errors appear at different stages
- Tools have clear boundaries

## Program lifecycle (bird’s eye)

1. Write source code
2. Translate
3. Combine
4. Load
5. Execute

# What exists at each stage?

| Stage | Output |
|---|---|
| Writing | **Source file** – human-readable program code |
| Translation | **Object file** – machine code, not yet runnable |
| Linking | **Executable** – complete machine code program |
| Loading | **Process** – running instance with memory and CPU state |

## Failure points

- Syntax errors → translation
  - Spelling and Grammar errors
- Missing symbols → linking
  - Referring to something that was never defined. Eg: Referring to a chapter or page number that was never written
- Memory issues → loading
  - Trying to fit into a space that is too small. Eg: Trying to move furniture into a room that is too small
- Logic bugs → execution
  - The instructions are correct, but the plan is wrong. Eg: Following a recipe perfectly and still getting bad food

# Any questions?

## Section 2 summary

- Program changes form many times
- Different tools handle each stage
- Errors appear at different stages

# Section 3 : Assembly Process & Assembler

## What we will see

- What role does assembly language play between human language like programs and machine langauge?
- What does an assembler take as input, and what does it produce as output?
- What actually happens inside the assembler when it translates a program?

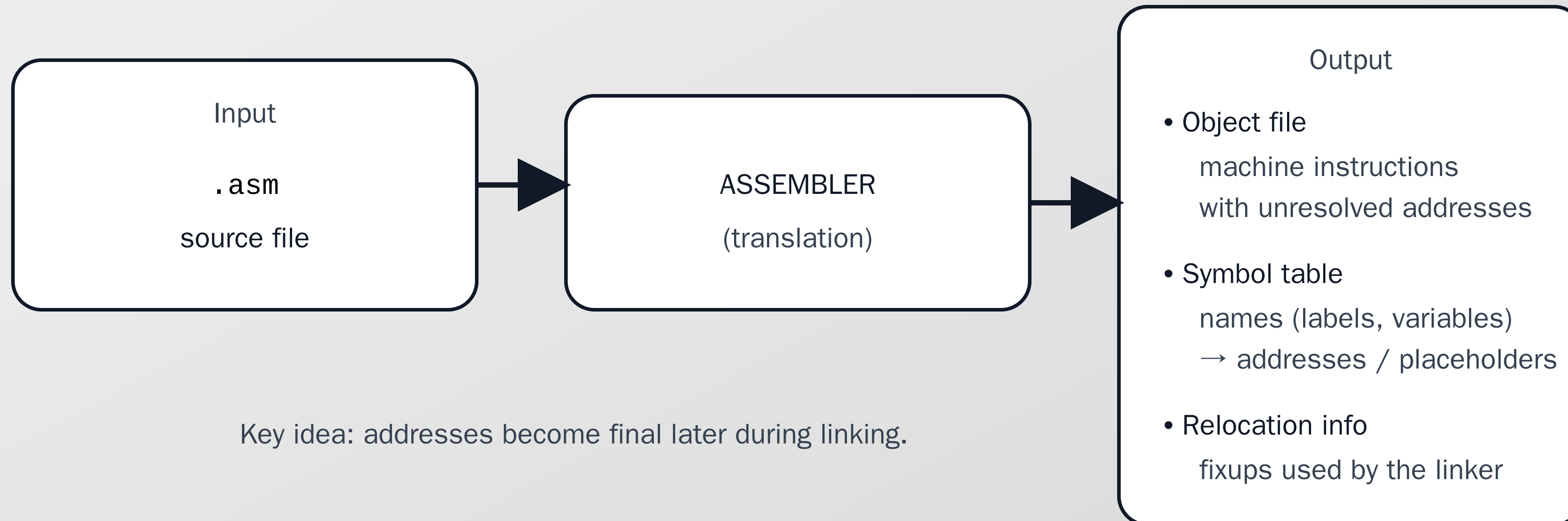
## Assembly language: reminder

- Symbolic representation, Eg: ADD AX, BX
- Close to hardware
- Architecture-specific

## Why assembly still matters

- Hardware control
- Predictable performance
- Boot code, interrupts

# Assembler: inputs and outputs



# Assembler: inputs and outputs

## Input

- `.asm` source file

## Output

- **Object file** – machine instructions with unresolved addresses
- **Symbol table** – mapping of names (labels, variables) to addresses or placeholders
- **Relocation info** – data used by the linker to adjust addresses later

# What is inside an object file?

- Machine instructions (not final addresses)
- Symbol table (names → placeholders)
- Relocation entries
- Optional debug information

## What assembler really does

- Mnemonic → opcode
- Labels → addresses
- Directives → layout rules

## Think Deeper 🤔

- Why not assign final addresses immediately?
- Why generate relocation info?

# Any questions?

## Section 3 summary

- Assembly bridges human and machine
- Assembler produces object code
- Object code is incomplete by design

# Section 4 : Linker

## What we will see

- Why linking exists
- Symbol resolution
- Libraries

# Why linking is needed

- Large programs
- Reuse code
- Separate compilation

## Linker responsibilities

- Combine object files
- Resolve external symbols
- Assign final addresses

## Symbols explained

- **Symbol**: A named reference to a function or variable whose address is resolved by the linker
- Function names
- Global variables
- Entry points

## Static vs shared libraries (idea)

- **Static libraries**: Code is copied into the executable at link time
- **Shared libraries**: Code is loaded at runtime and shared across programs

## Static vs Shared: trade-offs

| Static linking | Shared linking |
| --- | --- |
| Bigger executable | Smaller executable |
| No runtime dependency | Needs shared libraries |
| Predictable behavior | Flexible updates |
| Used in embedded systems | Used in desktops/servers |

# Real-life analogy

## Publishing a book

- Chapters written separately
- Editor resolves references
- Final book printed

## Think Deeper 🤔

- What happens if two files define same symbol?
- Why does linker error feel “late”?

# Any questions?

## Section 4 summary

- Linker creates a complete program
- Resolves names and locations
- Enables modular development

# Section 5 : Loader

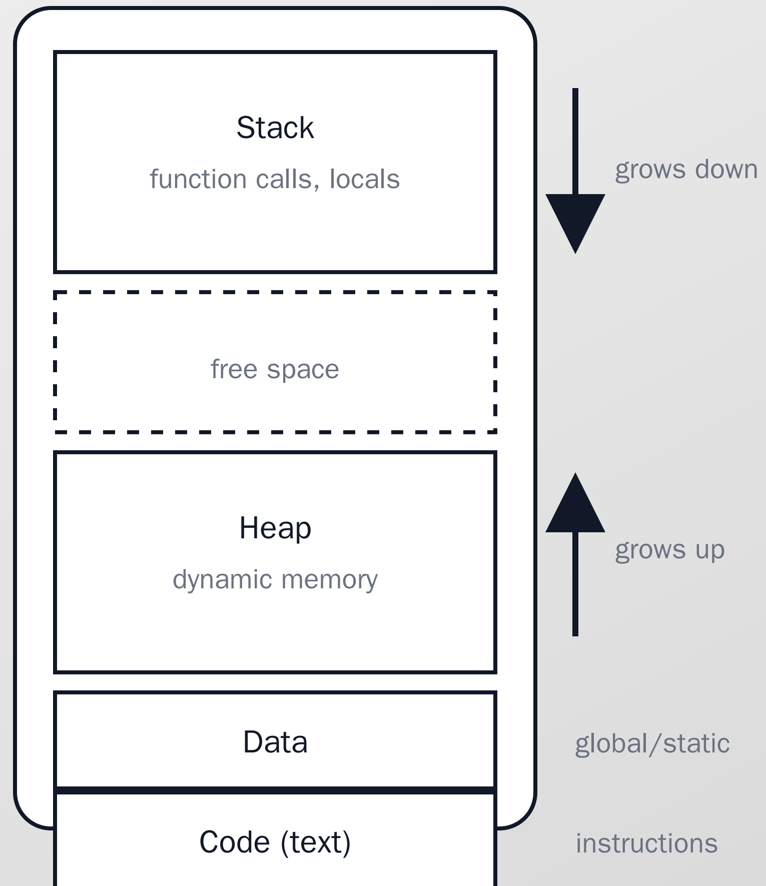
## What we will see

- What is the loader’s job when a program is prepared to run?
- What is the difference between a program and a process?
- How is the operating system involved in getting a program running?

## Loader's role

- Load executable into memory
- Setup **stack** – memory for function calls and local variables
- Setup **heap** – memory for dynamic allocation
- Start execution

## Process memory layout (simplified)



# Process memory layout (simplified)

- **Code (text)** – instructions
- **Data** – global/static variables
- **Heap** – dynamic memory (grows up)
- **Stack** – function calls, locals (grows down)

# Think Deeper

- Why is that stack grows down and heap up?
- The stack grows down and the heap grows up so both can expand dynamically and safely share the same address space. Most OSes + CPUs standardized this layout.

## Program vs Process

- Program = file on disk
- Process = running instance

# Loader errors

- Missing libraries
- Address conflicts
- Permission issues

# Real-life analogy

**Loader = stage setup**

- Script exists
- Actors placed
- Lights on
- Play begins

# Any questions?

## Section 5 summary

- Loader turns program into process
- Memory layout matters
- OS closely involved

## Who does what?

- **Compiler** → correctness + translation
- **Linker** → completeness
- **Loader** → execution setup
- **OS** → control and protection

# Section 6 : Compiler

## What we will see

- Why compilers exist
- Compilation stages
- Optimization idea

# Where compiler fits in the pipeline

> Important note on ordering

- In real execution:
  **C source → Compiler → Assembly → Assembler → Object → Linker**
- We studied **assembler first** to understand:
  - Object files
  - Symbols
  - Linking

This is a learning choice, not execution order.

## Why high-level languages?

- Productivity
- Portability
- Maintainability

## Compiler pipeline (simplified)

1. **Parsing** – check grammar and structure
2. **Analysis** – understand meaning, types, dependencies
3. **Code generation** – produce machine or assembly code
4. **Optimization** – improve performance without changing behavior

## Compiler and assembler connection

- Compiler may output:
    - **Assembly code** → goes to the assembler
    - **Object file directly** → skips assembler stage
- Either way, the result feeds into the **linker**

## Compiler output options

- Assembly
- Object file
- Intermediate code

## Optimization: key idea

- Same logic
- Fewer instructions
- Faster execution

## Optimization: the trade-off

- Same program meaning
- Different instruction order
- Faster execution
- Harder debugging

# Real-life analogy

**Compiler = experienced chef**

- Same recipe
- Fewer steps
- Less waste

## Think Deeper 🤔

- Can optimization change behavior?
- Why debugging optimized code is harder?

# Any questions?

## Section 6 summary

- Compiler bridges human logic and machine
- Multiple internal stages
- Optimization is powerful and risky

# Section 7 : Debugger

## What we will see

- Purpose of debugging tools
- Running Code (Execution) inspection
- Typical usage

## Debugger capabilities

- **Breakpoints** – pause execution at chosen points
- Step execution
- Inspect memory and registers

# What the compiler does to make debugging possible

- It keeps extra information that connects your source code to the running program
  - so the debugger knows which line and variable you are looking at
- It does not remove or heavily change parts of your code
  - variables stay visible instead of disappearing
- It keeps the program closer to how you wrote it
  - stepping through code feels natural and predictable

## Why debuggers matter

- Bugs are logical, not just syntax
- Print statements don’t scale

# Real-life analogy

**Debugger = flight recorder**

- Records state
- Helps after failure
- Enables learning

# Any questions?

## Section 7 summary

- Debuggers reveal program behavior
- Essential for real systems
- Part of normal workflow

# Section 8 : Mixing Assembly and C

## What we will see

- Why mix languages
- Practical cases
- Toolchain support

## Why mix C and assembly?

- Performance-critical code
- Hardware access
- Special instructions

## How mixing works

- Assembly produces object files
- C produces object files
- Linker combines both

# Inline assembly (idea)

- **Inline assembly**: Assembly instructions written inside a high-level language program
- Used carefully
- Compiler-dependent

## Think Deeper 🤔

- Why OS kernels use assembly?
- Why application code mostly avoids it?

# Any questions?

## Section 8 summary

- Language mixing is intentional
- Toolchain enables it
- Used sparingly but critically

# Section 9 : Operating System

## What we will see

- OS responsibilities
- OS as coordinator
- Relation to earlier tools

## OS: central coordinator

- CPU scheduling
- Memory management
- I/O handling

## OS enables tools

- Editor – file access, editing file
- Compiler – memory and process creation
- Assembler – disk and I/O services
- Linker – file management
- Loader – process startup

# Loader vs OS kernel

- **Loader**: prepares process image
  - A process image is everything a program needs in memory while it is running.
- **Kernel**: scheduling, memory protection, syscalls
  - A kernel is the core and foundational part of an operating system. It is a bridge between software applications and the hardware of a computer.
  - System calls are the way a program asks the operating system to do something for it. For example: read from file or keyboard etc.
- Clear boundary, shared responsibility

## Multitasking view

- Many processes
- Shared hardware
- Fair allocation

# Real-life analogy

**OS = orchestra conductor**

- Many instruments
- One performance
- Timed coordination

# Any questions?

## Summary

- Software is a pipeline
- Many specialized tools
- Each tool solves one problem
- Clear boundaries enable scale
    - Each tool does one well-defined job
    - Teams can work independently
    - Tools can improve or be replaced without breaking others

## Summary

- Design choices matter
- Design trade-offs are everywhere
- OS coordinates execution

## Reflect

- Which tool do you want to explore deeper and why?
- Where do you think most real bugs occur?

# Thank you!

Any questions?

Thejesh GN