# CSCI 135
# Overview

Subash Shankar

# What Is Computer Science?

One view – The science of computing (not necessarily computers), with four major components:

1. **Computability:** Characterize the set of problems that can be solved using some [abstract or real] machine (theory courses)
2. **Languages:** Express a problem in a precise form so that it can be solved on a target machine (programming courses)
3. **Systems:** Develop hardware and software systems that can execute programming language instructions on a real machine (architecture, operating systems, compilers courses; computer engineering)
4. **Applications:** Express **algorithms** for efficient solution of problems; **software engineering** for reading/writing/testing/maintaining software, ...

This course concentrates on languages, though touching on all aspects.

# What is Programming?

The expression of a problem in a programming language, in a way that the code is maintainable AND runs efficiently.
**Software Engineering** is the science of developing AND maintaining software (reality: $\frac{2}{3}$ of software lifecycle is spent in maintaining code, not coding). Some characteristics of well-engineered programs include:

- Readability/Writeability/etc.: Well structured and modularized, use of white space, etc.
- Documentation: Header comments for each module, *meaningful* inline comments for code (don't repeat code in English) especially for tricky parts
- Correctness: most commonly involves identification of good test cases
- ...

Don't forget, programs are often millions of lines of code, and have to be maintained for years!

# Paradigms

❓ How do you convert an [often incomplete, ambiguous, and self-contradictory] English specification into a computable form?

**Programming paradigm**: a way of thinking about or expressing a problem typically containing many languages (Ex: imperative, functional, logic).

C/C++ (and Java) fall under the **imperative** (also called procedural) paradigm. *i.e.*, the point of a statement is to update the value of a variable (which maps to a location in memory). This is called a *side effect*.

C++ adds **object-oriented constructs** to C, placing it in the {imperative,OO} paradigm (like Java). Object-orientation allows you to organize programs as a set of entities (called objects), with restrictions on object interaction that enable good software engineering practices.

Python attempts to mix multiple paradigms.

❓ What is good and bad about this?

# Some Languages

## Low-Level Language:

Close to the machine. The lowest level is **machine language** (binary data) or **assembly language** (the human-readable mnemonic form of the binary data).

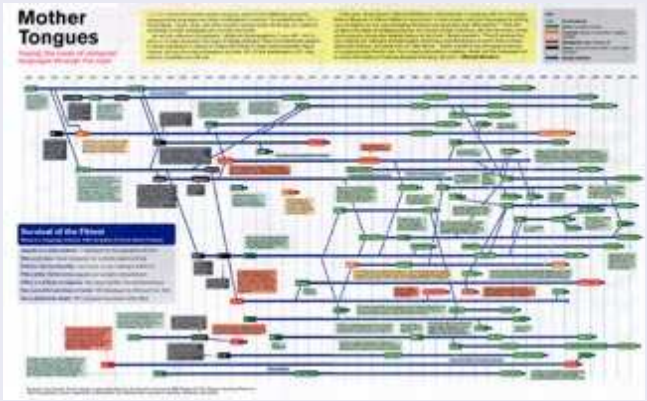
# Some Languages

## Low-Level Language:

Close to the machine. The lowest level is **machine language** (binary data) or **assembly language** (the human-readable mnemonic form of the binary data).

## High-Level Language:

Convenient to code in, but may not be close to the machine




All these languages are equivalent in expressive power!
(But problems may be easier to express in some languages, and C was designed to be close to machine)

# Partial History of Languages

The original Big 4:

1954 Fortran (Formula Translator)

1958 LISP (List Processing Language)

1958 Algol (Algorithm Language)

1959 COBOL (Common Business Oriented Language)

Assembly language needed for efficient and low-level code (*e.g.*, operating systems, embedded systems)

# Partial History of Languages

The original Big 4:

1954 Fortran (Formula Translator)

1958 LISP (List Processing Language)

1958 Algol (Algorithm Language)

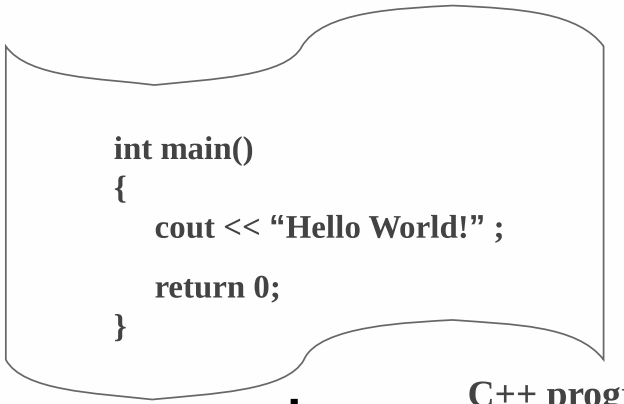
1959 COBOL (Common Business Oriented Language)

Assembly language needed for efficient and low-level code (*e.g.*, operating systems, embedded systems)

1970s Kernighan and Richie (K&R) develop an operating system (Unix) and a language (C) targeted at efficient and low-level applications.

1983 Stroustrup creates C++, a superset of C with object-oriented constructs ⇒ All C programs are also C++ programs
⇒ Good C++ programmers need to understand all C constructs well while also avoiding some C constructs that detract from the OO philosophy (when possible).

# Translating Between Levels




C++ program source file

A **compiler** converts a high-level (C++) program into machine language.

**Compile time.**

C++ Compiler

1001101001
1001001010

**Machine Language Program**

Operating System

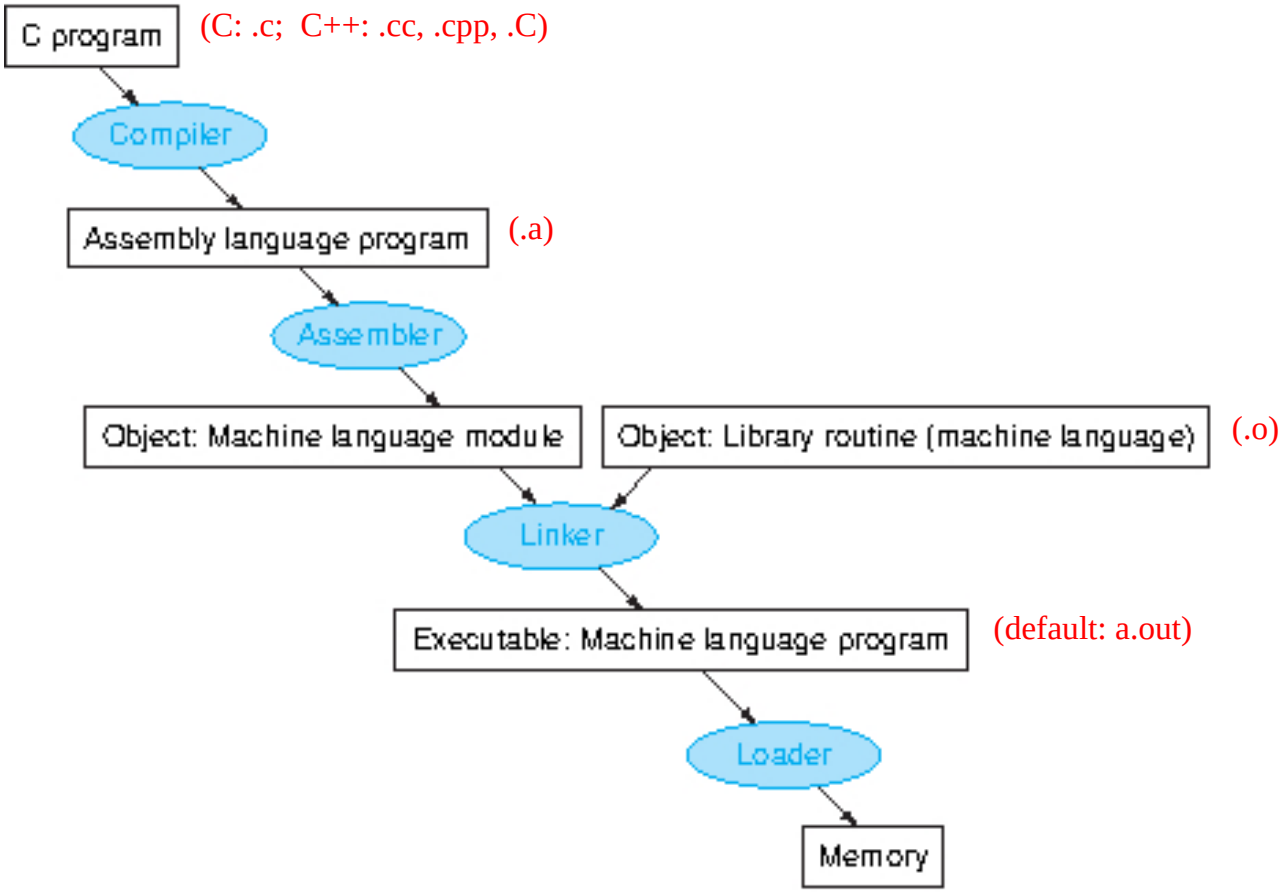
CPU

**CPU "*runs*" or "*executes*" the program.**

**chug ;**
**chug ;**
**chug ;**

**Run time.**

# Translating, More Precisely




(figure taken from Hennessy & Patterson, Computer Org. & Design, Elsevier)

# Alternate Translation Mechanisms

- Compilers: convert entire program to assembly before running
- Interpreters: convert each line of program to assembly when needed (reconvert if executing again)
- Virtual Machines: Compile program to intermediate form (p-code, called byte code in Java) virtual machine (*e.g.*, JVM); use interpreter to execute p-code.

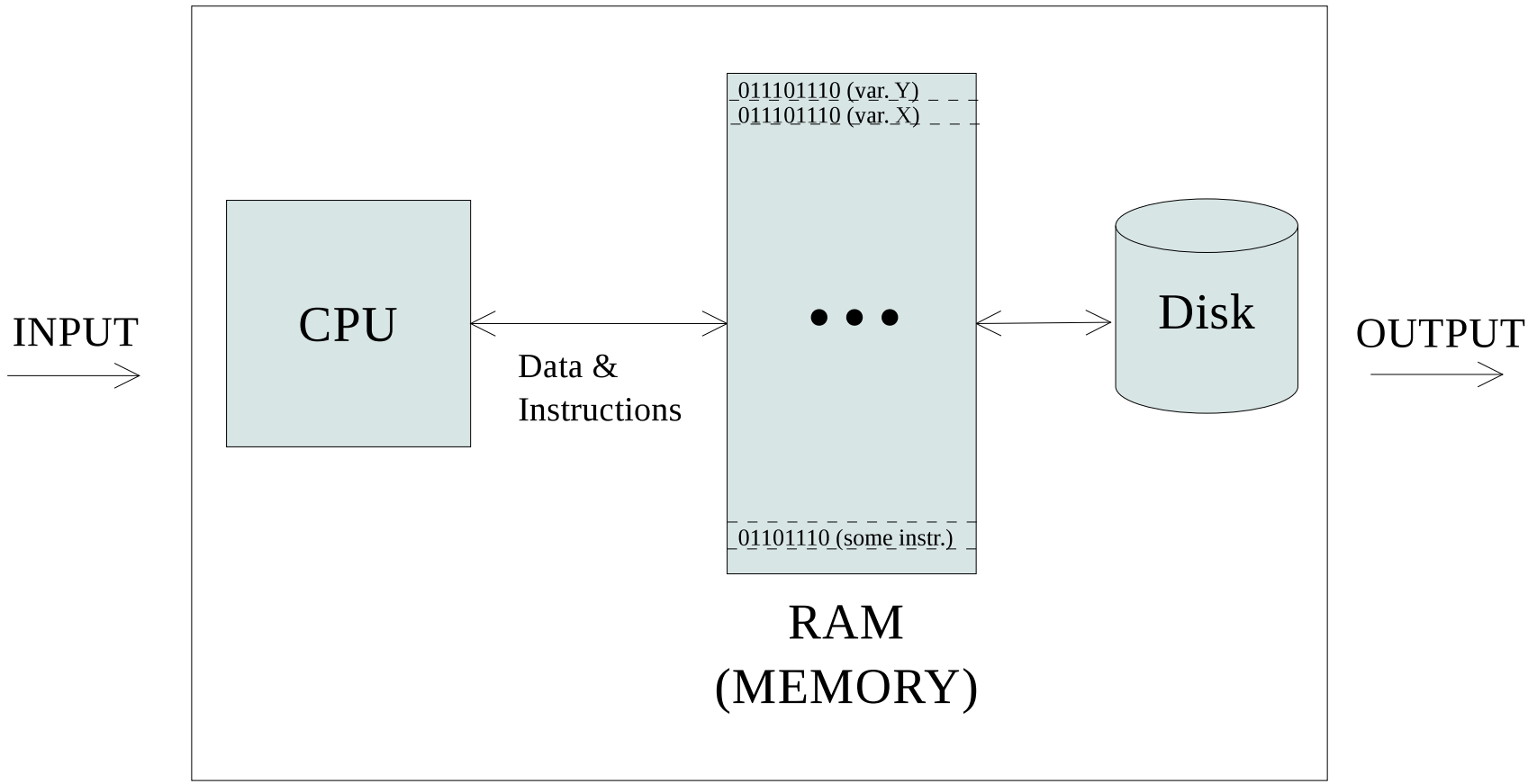
Compilers (vs. interpreters):

- Much faster execution!
- Slower to start running (need to translate whole program first) $\Rightarrow$ separate/incremental compilation
- Object-source mapping more difficult to maintain by debugging tools.

? Is P-code the best or worst of both worlds

# Executing it all

Recall: all programs and data (variables) are now just 0s/1s in RAM.




The **Operating System** oversees allocation of hardware resources (*e.g.*, RAM) for safe and efficient program execution.

# HLL Level

But wait! High-level languages differ on how high-level they are! Very High Level (vs. Lowish High Level) $\Rightarrow$:

- Minimal knowledge of machine needed
- Difficult to tweak for performance (but easier for efficient compilation?)
- No need to worry about which variables are stored in memory, how many bits they are, what their types are (*i.e.*, how to interpret those 0s and 1s), memory management, etc.
- Easier to program, especially for non-CSers
- Not appropriate for applications needing machine control (*e.g.*, embedded systems, operating systems).

# HLL Level

But wait! High-level languages differ on how high-level they are!
Very High Level (vs. Lowish High Level) $\Rightarrow$:

- Minimal knowledge of machine needed
- Difficult to tweak for performance (but easier for efficient compilation?)
- No need to worry about which variables are stored in memory, how many bits they are, what their types are (*i.e.*, how to interpret those 0s and 1s), memory management, etc.
- Easier to program, especially for non-CSers
- Not appropriate for applications needing machine control (*e.g.*, embedded systems, operating systems).

HLLs ranked from low to high:
C$\rightarrow$ Java $\rightarrow$ Python $\rightarrow$ functional/logic languages