University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

High-Level Languages

Programming Means Translating a Task into Instructions

Programming means **translating**
- **from a task specification** (in human language)
- **into instructions** (from an ISA).

As you already know,
- some of this process can be automated
- (done by computers),
- such as turning assembly language into bits.

Few Programmers Write Instructions (Assembly Code)

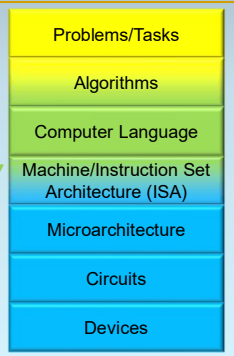
In ECE120, you learned
how to design a computer.

But computer **instructions are quite simple** (add two numbers, copy some bits).

Not many programmers use them directly.

Challenge: Semantic Gap Between Human and Computer

The difficulty is the **semantic gap** between human expression and computational capabilities.



There has been substantial effort to bridge this gap for more than 60 years.

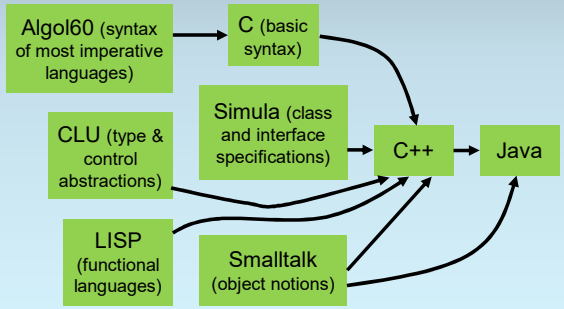
Most Programs Are Written in High-Level Languages

FORTRAN (FORmula TRANSlator)
- was introduced in 1954
- to help scientists express equations in a more natural way.

Since then,
- thousands of languages have been invented,
- with tens of them widely used commercially.

Most programs are written in these languages.

Language Evolution is Convoluted



Some Languages Can Be Compiled to Instructions

Languages can be **compiled** or **interpreted**.

Compiled languages include
FORTRAN, Pascal, C, and C++.

Code written in compiled languages
- is **translated to assembly language**
- by a program called a **compiler**.
- After assembly, the code **runs directly on a computer**.

Interpreted Languages Require an Interpreter to Execute

Interpreted languages include
Perl, Python, Javascript, and Java.

Code written in interpreted languages
- is **used as input to** another program (called **an interpreter**)
- **that executes the code** written in the interpreted language.

JIT Compilation Focuses on the Code Being Used

In some cases,
- **interpreted languages** may be
- **partially compiled** to instructions
- **when executed**.

Usually, only the most frequently used parts of the program are compiled.

This approach is called **Just In Time (JIT) compilation**, and is often used in Java Virtual Machines (JVMs).

A Brief History of C

The **C programming language** was
- developed by Dennis Ritchie in 1972
- to simplify the task of writing Unix.

C has a transparent mapping to typical ISAs:
- easy to understand the mapping
- easy to teach a computer:
  C compiler (a program) converts a
  C program into instructions

C was first standardized in 1989 by ANSI.

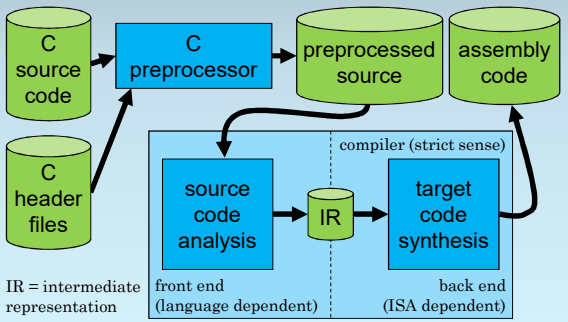
Our Class Starts with C. Here's Why.

As mentioned,
- C is **easy to translate** (to LC-3, for example)
- so you can **understand exactly what a compiler does**.

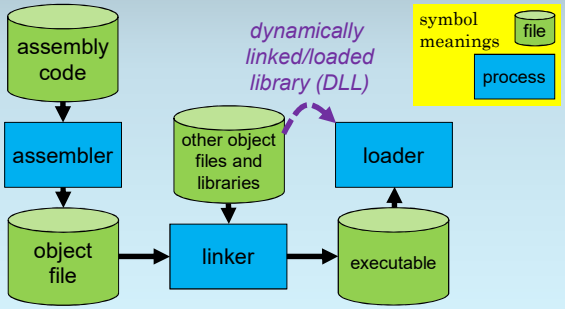
C **syntax is similar to** that of **many useful languages**.

To write C++ well, **you must** be able to **write** the **C** part well.

Overview of the C Compilation Process



Process Same as Before with Assembly Code



A Compiler Turns Preprocessed Source into Assembly

But doesn't the compiler turn C code into an executable?

Actually, no.

As shown in the diagram, a compiler
- **turns preprocessed source code**
- (with header files incorporated,
- and macros expanded)
- **into assembly code**.

A Compiler Can Also Invoke Other Programs

A **compiler can also execute** (by default, but optionally)
- a **preprocessor**,
- an **assembler**, and
- a **linker**.

What if you don't want all of the steps?*
- Use -E to obtain preprocessed output.
- Use -S to obtain assembly code.
- Use -c to obtain an object file (.o).

*These are the `gcc` options.

Too Many Possible Combinations of Language and ISA

Why are compilers built in two parts?

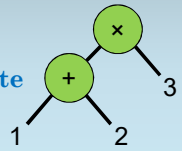
Imagine developing a compiler…
- languages: C, C++, Pascal, Java, and more
- ISAs: x86, ARM, PowerPC, Power, and more

Do you develop a separate compiler
- **for every language/ISA combination?**
- 10 languages, 10 ISAs → 100 compilers!

No.

Front End and Back End Operate Independently

Instead,
- **front end** converts language (such as C) to an **intermediate representation (IR)**, such as … trees!



- (IR can be optimized.)
- **back end** converts IR to assembly code.*

(10 + 10) / 2 = 10 compilers to write (<< 100)!

*Take CS426 (421 for front-end, with other stuff).

A Modern Example

Chris Lattner (UIUC CS Ph.D., 2005)
- developed LLVM compiler framework
- with Vikram Adve's group as a grad student,
- and continued to work on it within Apple.

In 2010, he
- started to develop the Swift programming language,
- using the LLVM compiler (IR and back end) as a starting point.

One Benefit of High-Level Languages: Managing Variables

What good are high-level languages?

Remember deciding (in examples and MPs)
- what information to store, and
- where to put it
- (which register, or which memory location)?

In high-level languages,
- **programmer specifies symbolic name** (like a label in assembly) **and**
- **data type**.

Compiler decides where to put each variable.

High-Level Languages Support Complex Data Types

The benefit generalizes to include…
- **structures** (such as events in MP2), and
- **arrays** (event list in MP2), and
- **pointers** (in the schedule in MP2).*

Compiler
- **knows how each maps into memory**,
- and **manages access for you** by name.

*We'll see how later in our class.

High-Level Languages Provide Operators and Libraries

High-level languages also provide
- **a set of operators**
  - that is not (too) dependent on the ISA
  - so you do not need to write right shift, OR, XOR, and so forth.
- **standard libraries** for
  - I/O,
  - math,
  - graphics,
  - threads,
  - and many other things.

How to Learn C Programming in ECE220

In lecture, you will **learn from examples**.

Exact rules of syntax are left to you.

To be good at programming,
you need practice
- **reading code** (examples in lecture / online),
- **writing code** (MPs), and
- **testing code** (MPs, one focused on testing)

Ask lots of questions!

Learn to Program by Reading Code

You can learn a lot by reading code

- How to **express types of problems**.
- How to **properly use application programming interfaces** (APIs) for networking, mathematics, graphics, sound, animation, user interfaces, and so forth.
- How to **make code easy to read** (style).

It's Often Necessary to Read Code to Understand It

We try to make you write plenty of comments.

When we give you code for class assignments, it will be well-commented (DISCLAIMER: THIS IS NOT A WARRANTY!)

In the real world…
- You will be lucky to find comments.
- You will be really lucky to find comments in a language that you understand.

Learn to Test Your Code

How do you know that your program works?

There's only one correct answer: test it!*

Brooks' Rule of Thumb
- 1/3 planning and design
- 1/6 writing the program
- **1/2 testing**

Just because your program compiles does not mean that your program works!

*Becoming a good tester will take years. Don't worry if it seems tough.

A Starting Point: Every Statement Must be Executed

How can we test our program?

Let's start with something simple.

Let's say that we have **a statement that is never executed by tests**.

Does the statement work correctly?

How can we know? We have no tests!

So, no, it **does not work correctly**.

At a minimum, we **must execute every statement** (called **full code coverage**).

