2024-09-19

# Computers and Programming

## Principles of Computer Programming

* Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets - and will continue doing so.
* With these changes, the capabilities of computers increase rapidly (storage, speed, graphics, etc.)
* Computer programming languages also change
	+ Better programming language theory leads to new programming techniques
	+ Improved programming language implementations
	+ New languages are created, old ones updated
* There are [hundreds of programming languages](https://www.wikiwand.com/en/List_of_programming_languages), why?
	+ Different tools for different jobs
		- Some languages are better suited for certain jobs
		- For example, Python is best for scripting, Javascript is best for web pages, MySQL is best for databases, etc.
	+ Personal preference and popularity
* This class is about “principles” of computer programming
	+ Common principles behind all languages will not change, even though hardware and languages do
	+ How to organize and structure data
	+ How to express logical conditions and relations
	+ How to solve problems with programs

## Programming Language Concepts

We begin by discussing three categories of languages manipulated by computers. We will be studying and writing programs in *high-level languages*, but understanding their differences and relationships to other languages[[1]](#footnote-22) is of importance to become familiar with them.

* Machine language
	+ Computers are made of electronic circuits
		- Circuits are components connected by wires
		- Some wires carry data - e.g. numbers
		- Some carry control signals - e.g. do an add or a subtract operation
	+ Instructions are settings on these control signals
		- A setting is represented as a 0 or 1
		- A machine language instruction is a group of settings - For example: 1000100111011000
	+ Most CPUs use one of two languages: x86 or ARM
* Assembly language
	+ Easier way for humans to write machine-language instructions
	+ Instead of 1s and 0s, it uses letters and “words” to represent an instruction.
		- Example x86 instruction:
	+ MOV BX, AX
	+ which makes a copy of data stored in a component called AX and places it in one called BX
	+ **Assembler**: Translates assembly language instructions to machine language instructions
		- For example: MOV BX, AX translates into 1000100111011000
		- One assembly instruction = one machine-language instruction
		- x86 assembly produces x86 machine code
	+ Computers can only execute the machine code
* High-level language
	+ Hundreds including C#, C++, Java, Python, etc.
	+ Most programs are written in a high-level language since:
		- More human-readable than assembly language
		- High-level concepts such as processing a collection of items are easier to write and understand
		- Takes less code since each statement might be translated into several assembly instructions
	+ **Compiler**: Translates high-level language to machine code
		- Finds “spelling” errors but not problem-solving errors
		- Incorporates code libraries – commonly used pieces of code previously written such as Math.Sqrt(9)
		- Optimizes high-level instructions – your code may look very different after it has been optimized
		- Compiler is specific to both the source language and the target computer
	+ Compile high-level instructions into machine code then execute since computers can only execute machine code



A Visual Representation of the Relationships Between Languages

A more subtle difference exists between high-level languages. Some (like C) are *compiled* (as we discussed above), some (like Python) are *interpreted*, and some (like C#) are in an in-between called *managed*.

* Compiled vs. Interpreted languages
	+ Not all high-level languages use a compiler - some use an interpreter
	+ **Interpreter**: Lets a computer “execute” high-level code by translating one statement at a time to machine code
	+ Advantage: Less waiting time before you can execute the program (no separate “compile” step)
	+ Disadvantage: Program executes slower since you wait for the high-level statements to be translated then the program is executed
* Managed high-level languages (like C#)
	+ Combine features of compiled and interpreted languages
	+ Compiler translates high-level statements to **intermediate language** instructions, not machine code
		- Intermediate language: Looks like assembly language, but not specific to any CPU
	+ **run-time** executes by *interpreting* the intermediate language instructions - translates one at a time to machine code
		- Faster since translation is partially done already (by compiler), only a simple “last step” is done when executing the program
	+ Advantages of managed languages:
		- In a “non-managed” language, a compiled program only works on one OS + CPU combination (**platform**) because it is machine code
		- Managed-language programs can be reused on a different platform without recompiling - intermediate language is not machine code and not CPU-specific
		- Still need to write an intermediate language interpreter for each platform (so it produces the right machine code), but, for a non-managed language, you must write a compiler for each platform
		- Writing a compiler is more complicated and more work than writing an interpreter thus an interpreter is a quicker (and cheaper) way to put your language on different platforms
		- Intermediate-language interpreter is much faster than a high-level language interpreter, so programs execute faster than an “interpreted language” like Python
	+ This still executes slower than a non-managed language (due to the interpreter), so performance-minded programmers use non-managed compiled languages (e.g. for video games)



A Visual Representation of the Differences Between High-Level Languages

## Software Concepts

* Flow of execution in a program
	+ Program receives input from some source, e.g. keyboard, mouse, data in files
	+ Program uses input to make decisions
	+ Program produces output for the outside world to see, e.g. by displaying images on screen, writing text to console, or saving data in files
* Program interfaces
	+ **GUI** or Graphical User Interface: Input is from clicking mouse in visual elements on screen (buttons, menus, etc.), output is by drawing onto the screen
	+ **CLI** or Command Line Interface: Input is from text typed into “command prompt” or “terminal window,” output is text printed at same terminal window
	+ This class will use CLI because it is simple, portable, easy to work with – no need to learn how to draw images, just read and write text

## Programming Concepts

### Programming workflow



Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs.

The workflow of the programmer will differ a bit depending on if the program is written in a compiled or an intprepreted programming language. From the distance, both looks like what is pictured in the [the flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs](#fig:flowchart1), but some differences remain:

* Compiled language workflow
	+ Writing down specifications
	+ Creating the source code
	+ Running the compiler
	+ Reading the compiler’s output, warning and error messages
	+ Fixing compile errors, if necessary
	+ Executing and testing the program
	+ Debugging the program, if necessary
* Interpreted language workflow
	+ Writing down specifications
	+ Creating the source code
	+ Executing the program in the interpreter
	+ Reading the interpreter’s output, determining if there is a syntax (language) error or the program finished executing
	+ Editing the program to fix syntax errors
	+ Testing the program (once it can execute with no errors)
	+ Debugging the program, if necessary

Interpreted languages have

* **Advantages**: Fewer steps between writing and executing, can be a faster cycle
* **Disadvantages**: All errors happen when you execute the program, no distinction between syntax errors (compile errors) and logic errors (bugs in executing program)

### (Integrated) Development Environment

Programmers can either use a collection of tools to write, compile, debug and execute a program, or use an “all-in-one” solution called an Integrated Development Environment (IDE).

* The [“Unix philosophy”](https://www.wikiwand.com/en/Unix_philosophy) states that a program should do only one task, and do it properly. For programmers, this means that
	+ One program will be needed to edit the source code, a text editor (it can be Geany, notepad, kwrite, emacs, sublime text, vi, etc.),
	+ One program will be needed to compile the source code, a compiler (for C#, it will be either [mono](https://www.wikiwand.com/en/Mono_%28software%29) or [Roslyn](https://www.wikiwand.com/en/Roslyn_%28compiler%29),
	+ Other programs may be needed to debug, execute, or organize larger projects, such as makefile or [MKBundle](https://www.mono-project.com/docs/tools%2Blibraries/tools/mkbundle/).

IDE “bundle” all of those functionality into a single interface, to ease the workflow of the programmer. This means sometimes that programmers have fewer control over their tools, but that it is easier to get started.

* Integrated Development Environment (IDE)
	+ Combines a text editor, compiler, file browser, debugger, and other tools
	+ Helps you organize a programming project
	+ Helps you write, compile, and test code in one place

In particular, Visual Studio is an IDE, and it uses its own vocabulary:

* Solution: An entire software project, including source code, metadata, input data files, etc.
* “Build solution”: Compile all of your code
* “Start without debugging”: Execute the compiled code
* Solution location: The folder (on your computer’s file system) that contains the solution, meaning all your code and the information needed to compile and execute it
1. That will be studied in the course of your study if you continue as a CS major. [↑](#footnote-ref-22)