0.2 — Introduction to programming languages

Modern computers are incredibly fast, and getting faster all the time. However, computers also have some significant constraints: they only natively understand a limited set of commands, and must be told exactly what to do.

A **computer program** (also commonly called an **application**) is a set of instructions that the computer can perform in order to perform some task. The process of creating a program is called **programming**. Programmers typically create programs by producing **source code** (commonly shortened to **code**), which is a list of commands typed into one or more text files.

The collection of physical computer parts that make up a computer and execute programs is called the **hardware**. When a computer program is loaded into memory and the hardware sequentially executes each instruction, this is called **running** or **executing** the program.

Machine Language

A computer's CPU is incapable of speaking C++. The limited set of instructions that a CPU can understand directly is called **machine code** (or **machine language** or an **instruction set**).

Here is a sample machine language instruction: `10110000 01100001`

Back when computers were first invented, programmers had to write programs directly in machine language, which was a very difficult and time-consuming thing to do.

How these instructions are organized is beyond the scope of this introduction, but it is interesting to note two things. First, each instruction is composed of a sequence of 1s and 0s. Each individual 0 or 1 is called a **binary digit**, or **bit** for short. The number of bits that make up a single command varies -- for example, some CPUs process instructions that are always 32 bits long, whereas some other CPUs (such as the x86/x64 family, which you may be using) have instructions that can be a variable length.

Second, each set of binary digits is interpreted by the CPU into a command to do a very specific job, such as *compare these two numbers*, or *put this number in that memory location*. However, because different CPUs have different instruction sets, instructions that were written for one CPU type could not be used on a CPU that didn't share the same instruction set. This meant programs generally weren't **portable** (usable without major rework) to different types of system, and had to be written all over again.

Assembly Language

Because machine language is so hard for humans to read and understand, assembly language was invented. In an assembly language, each instruction is identified by a short abbreviation (rather than a set of bits), and names and other numbers can be used.

Here is the same instruction as above in assembly language: `mov al, 061h`

This makes assembly much easier to read and write than machine language. However, the CPU can not understand assembly language directly. Instead, the assembly program must be translated into machine language before it can be executed by the computer. This is done by using a program called an **assembler**. Programs written in assembly languages tend to be very fast, and assembly is still used today when speed is critical.

However, assembly still has some downsides. First, assembly languages still require a lot of instructions to do even simple tasks. While the individual instructions themselves are somewhat human readable, understanding what an entire program is doing can be challenging (it's a bit like trying to understand a sentence by looking at each letter individually). Second, assembly language still isn't very portable -- a program written in assembly for one CPU will likely not work on hardware that uses a different instruction set, and would have to be rewritten or extensively modified.

High-level Languages

To address the readability and portability concerns, new programming languages such as C, C++, Pascal (and later, languages such as Java, Javascript, and Perl) were developed. These languages are called **high level languages**, as they are designed to allow the programmer to write programs without having to be as concerned about what kind of computer the program will be run on.

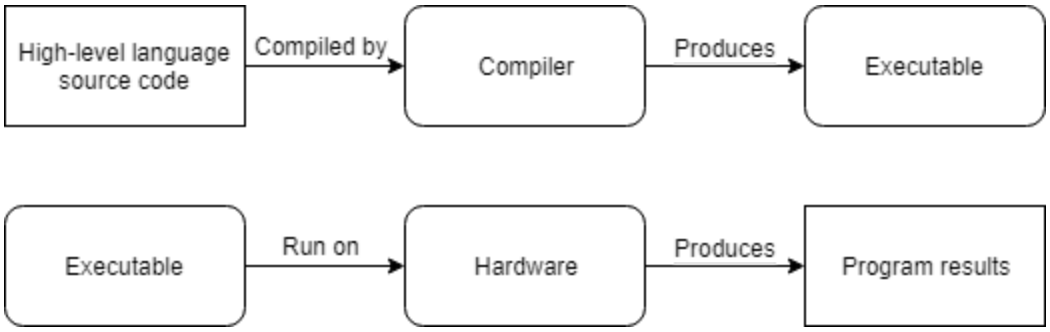
Here is the same instruction as above in C/C++: `a = 97;`

Much like assembly programs, programs written in high level languages must be translated into a format the computer can understand before they can be run. There are two primary ways this is done: compiling and interpreting.

A **compiler** is a program (or collection of programs) that reads source code (typically written in a high-level language) and translates it into some other language (typically a low-level language, such as assembly or machine language, etc…). Most often, these low-level language files are then combined into an executable file (containing machine language instructions) that can be run or distributed to others. Notably, running the executable file does not require the compiler to be installed.

In the beginning, compilers were primitive and produced slow, unoptimized code. However, over the years, compilers have become very good at producing fast, optimized code, and in some cases can do a better job than humans can in assembly language!

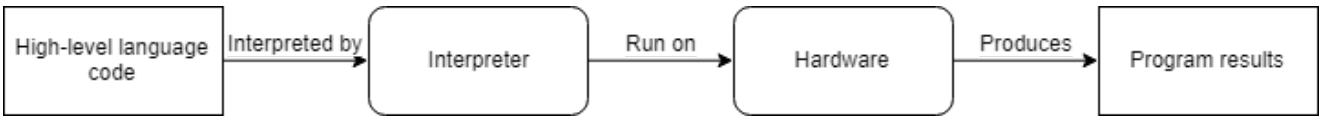
Here is a simplified representation of the compiling process:



Since C++ programs are generally compiled, we'll explore C++ compilers in more detail shortly.

An **interpreter** is a program that directly executes the instructions in the source code without requiring them to be compiled into an executable first. Interpreters tend to be more flexible than compilers, but are less efficient when running programs because the interpreting process needs to be done every time the program is run. This also means the interpreter must be installed on every machine where an interpreted program will be run.

Here is a simplified representation of the interpretation process:



Optional reading

A good comparison of the advantages of compilers vs interpreters can be found here.

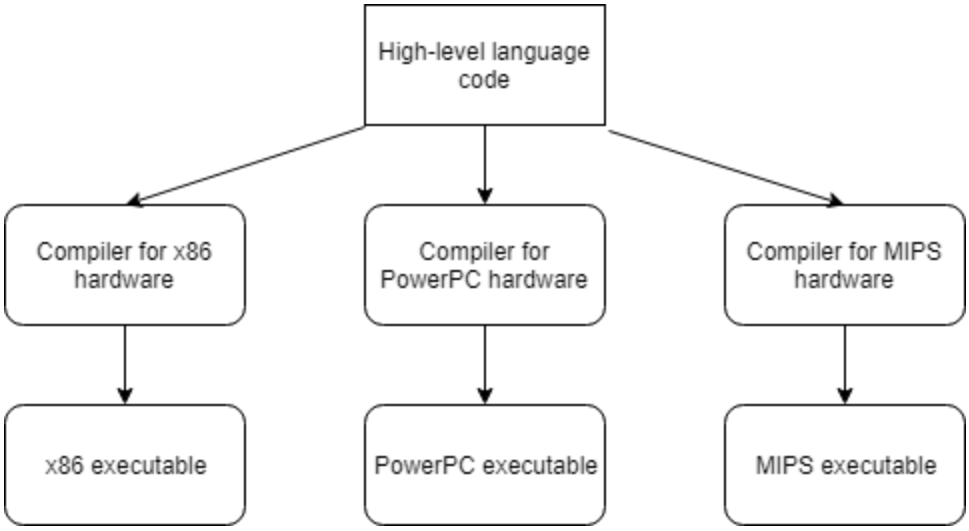
Another advantage of compiled programs is that distributing a compiled program does not require distributing the source code. In a non-open-source environment, this is important for IP protection purposes.

Most languages can be compiled or interpreted. Traditionally languages like C, C++, and Pascal are compiled, whereas "scripting" languages like Perl and Javascript tend to be interpreted. Some languages, like Java, use a mix of the two.

High level languages have many desirable properties.

First, high level languages are much easier to read and write because the commands are closer to natural language that we use every day. Second, high level languages require fewer instructions to perform the same task as lower level languages, making programs more concise and easier to understand. In C++ you can do something like `a = b * 2 + 5;` in one line. In assembly language, this would take 5 or 6 different instructions.

Third, programs can be compiled (or interpreted) for many different systems, and you don't have to change the program to run on different CPUs (you just recompile for that CPU). As an example:



There are two general exceptions to portability.

The first is that many operating systems, such as Microsoft Windows, contain platform-specific capabilities that you can use in your code. These can make it much easier to write a program for a specific operating system, but at the expense of portability. In these tutorials, we will avoid any platform specific code.

The second is that some compilers also support compiler-specific extensions -- if you use these, your programs won't be able to be compiled by other compilers that don't support the same extensions without modification. We'll talk more about these later, once you've installed a compiler.

Rules, Best practices, and warnings

As we proceed through these tutorials, we'll highlight many important points under the following three categories:

Rule

Rules are instructions that you *must* do, as required by the language. Failure to abide by a rule will generally result in your program not working.

Best practice

Best practices are things that you *should* do, because that way of doing things is either conventional (idiomatic) or recommended. That is, either everybody does it that way (and if you do otherwise, you'll be doing something people don't expect), or it is superior to the alternatives.

Warning

Warnings are things that you *should not* do, because they will generally lead to unexpected results.