Chapter 2: Logic and Languages

Notes

- "Bombing" or "Crashing" is when unexpected results occur in a program. This may be the program halting and not responding, causing the system to stop responding, or simply corrupting input, output, or data.

Computers are Smarter

In computery stuff, we have a hard drive and usually disks and printers. The data on the hard drive stays there even after we shut off power to it. We don't need to concern ourselves with how, just familiarize yourself with the concept. Likewise a printed piece of paper that contains data is permanent because it requires nothing except to be untampered. The only way to lose data on a hard drive or printed paper is to actively *try* to harm them.

In our machine, this simple program would most likely be fixed to the hardware itself, and unchangable. As machines gain complexity, their programs must be updated to extend their functionality or fix old functionality. Personal computers use these kinds of **software programs**.

Software programs are still instructions, but they exist on some kind of storage rather than on the machine's logical hardware itself. These are not already simply "known" by the machine, they must be loaded and *run*. In the world of computing you know, these are executable files on your personal computer.

When a **software program** is run, it is read by the machine using it's brain, the CPU (microprocessor). It follows the instructions of the program exactly. It stores what the program wants, retrieves what the program wants, outputs what the program wants, and feeds the program requested input. The CPU rarely denies requests and is therefore a powerful, but dangerous thing.

So, there is another platform on which much of this logic can be run: the **operating system**. The **operating system** is a software program or programs that can run other programs that are written specifically for it. When these higher level programs need access to the hardware of the machine, they will ask the operating system rather than talking to the hardware directly through the CPU. This is safer and because the **operating system** has a lot of logic already, higher-level programs can use that logic without writing more of their own.

What I will be teaching you to write are software programs to run on modern operating systems running on modern computers. By modern, I mean within the last couple decades or so … I won't be picky if you're using *this* (link to old Tandy from Art Bell here!) computer.

Computers Don't Speak Love

Everyone now and again hears or mentions the magical phrase "universal language". Let me tell you, computers don't speak it. In fact, if you thought there was a battle of the languages here in the real world, wait until you get into the cyber realm! People already familiar with that aspect of debate know what I'm talking about. What you'll find is that many people will defend their favorite language to the death; worse than the crappy music they listen to!

But what am I talking about with languages? At any of several levels in the computer's logic there exists a language. The language is the *format of the logical instructions that comprise a program*. In personal computers, the very base language is that of the CPU itself and is known as "machine language". In fact, your computer itself knows nothing other than this dismal language. The program with machine language is it's simplified beyond human comprehension. Every instruction is broken down to very, *very* simple things so more of them can be read into the CPU at once. It doesn't have to do any *parsing* (i.e. breaking the data into it's logical and useful parts and then processing it) on them at all. It would be the equivalent of instinct in animals. They just *know* how to do some things, just like the CPU *knows* how to read these instructions brainlessly and endlessly.

In the dark ages of computing, humans were slaves to computers and were forced to *write* in machine language. Then came the first of the translated languages: assembler. It was hardly a step up from machine language, but I can actually put it in writing. For example, to place the number 5 in one of the CPU's registers (i.e. "brain cells") you could write:

```
Mov ax, 5
```

Since computers only really understand their machine language, programs written in these languages had to be translated. No human would want that job, let me tell you, so the instructions were fed into a translator program whose instructions were already in machine language. For assembler the process of translating is known as “assembling” and the program used to do it simply as “assembler”.

Assembler is known as a low-level language because it is exactly like machine language except mildly comprehensible. Later on, more translated languages popped up but they were “higher-level” because they were even more readable and less tied to the computer’s machine language. This sparked the idea of portability … you could take instructions from Machine A and put them on Machine B and compile them there into Machine B’s machine language. These early high-level languages (such as “COBOL”) were good points in history but they had their short comings (they were the first, so this was inevitable). For one thing, portability was really a myth. You could take instructions from Machine A, then put them on Machine B, then pay someone to modify them for all the special “syntax” Machine B used, then have it compiled, then pay for repairs when it didn’t work, etc. And they seemed to think that broken English was a great architecture for a “human readable” programming language.

I used some “new” terms there. One was “syntax”. It simply means the rules of a language (or “features”, but those are inclusive to rules). The latter was “programming language”. A high-level language is known as a “programming language” for the simple reason that you write in this foreign language for the purpose of programming.

The higher-level translated languages of which I have been speaking and we will focus on are called “compiled languages”. C and C++ are compiled languages. The term you figured was coming is “compiler”: the program used to translate compiled languages into machine language.

The fundamental thing you must realize here is that computers cannot *read* or *understand* C++ without it first being translated. Another way a computer read instructions from a high-level language is through “interpretation” rather the “compilation”. The term “interpretation” is used by languages whose instructions are read by a translator *when* the program is to be run.

Viva La Existance

The instructions for any language exist in files. For high-level languages these files are known as "source" files as they contain "source code". The terms "source" and "source code" are synonymous … I suppose someone just got tired of saying "code" and dropped it and no one really has noticed since. Source is actually just a way of saying "instructions". When I say "Here my C++ source", what I mean is "Here are my instructions written in C++". For C++, these files usually have one of the following extensions: "c", "cpp", "cxx", "h", or "hpp". I'll get into what each of them typically means later.

Most machine language exists in files as well. These are typically referred to as "executable" files. They don't have a larger salary than other files, but the machine *does* understand them natively. These files can be executed or "run" (*not* killed) natively by the machine. On DOS/Windows machines, these files have the extension "exe".

Your C++ source files are *not* executable … though you would like them to be. You cannot simply run them because the computer doesn't understand their language; it only knows the blasphemous machine language. If you try to "run" a C++ source file on Microsoft Windows, by say "double-clicking" it (a popular choice for many people today – double clicking that is) … it will either open the file *using* a program (like Microsoft Visual C++, TextPad, etc.) or ask you what you want to do with it. But it cannot *run* the file; it isn't executable. Remember you must compile (translate) your C++ source files, and therefore your C++ instructions, into machine code and an executable file. More succinctly put, you must translate your source file into an executable file.

Notice I said source *files* (plural). This is an understandability issue. Usually programmers will keep their program's source code in multiple source files. This makes it easier to manage and change. These files will all get compiled (remember that means translated) into a single file of machine code. That is, of course, if they are for a single project. You see, not all machine code is in a single file either. When you keep those instructions in separate files you only have to change those files to change that set of instructions. Did you think Windows was stored in a single executable somewhere on your system that you could doink with? No, it's stored in many, *many* files.

Not all executable files can be run directly. Some of them are sucked in when other files are run. For example, DLL's (or SO's if you're on a Unices platform) are executable files but you cannot run them directly.

They need additional information to run that you cannot give them; other programs must.

Other Things That Run … or Crawl

Now, if you're fairly adept at operating a computer you might be wondering about those other programs that you can "run". Things like command scripts, batch files, etc. These are instructions, no doubt about it, and they are *not* machine language because usually you can actually read (and edit) them with a text editor. The instructions in these files is still *translated* just like C++ instructions are *compiled* and Assembler instructions are *assembled*. The translation for these types of files is known as "interpretation". Yeah, it's a bit silly but it's true. The interpreter (program that translates interpreted languages) has all the machine language at it's disposal that the interpreted language will ever use. It reads the instructions, *interprets* them, executes some machine language that equates to the instruction, and then moves on to the next one. This all happens in "real time" and no executable file is ever generated.

These languages are also sometimes called "scripting" languages. They include the ever popular JavaScript, VisualBasic Script (VBA), QBasic, and batch files.

The typical advantage of interpreted languages is that they're easy to understand, quick to write (since the interpreter usually knows a lot so the programmer has to know less), and it is easier to write extremely "dynamic" (ever-changing) things. The major disadvantage is speed, followed by rules and portability. Since you are insulated from the machine itself by the interpreter, you can make less mistakes but in being further away your program is slower. High performance programs are not typically written in script (a short-name for an interpreted language). Many utilities, however, are and for good reason.

Since this tutorial is meant to school you in C++, a compiled language, I will not spend any more time on interpreted languages.

Interpreted Compiled Instructions

Yes, there is such a thing as instructions that are compiled *into* an interpreted language. On very popular example is Java. The Java source is written in a human-readable format. It is then compiled into byte-code which is instructions for a "virtual machine" (fancy name for an interpreter). The advantage of this is that the interpreter is much faster because the format it must understand is *much* simpler (almost machine-code like). The disadvantages are still basically the same as an interpreted language; except now you have to worry about some other things.

Again, I will speak no more of these byte-code languages as C++ is not one of them. If you're interested in more information, see here: …