Levels of Programming Languages

There is only one programming language that any computer can actually understand and execute: its own native binary machine code. This is the lowest possible level of language in which it is possible to write a computer program. All other languages are said to be *high level* or *low level* according to how closely they can be said to resemble machine code.

In this context, a low-level language corresponds closely to machine code, so that a single low-level language instruction translates to a single machine-language instruction. A high-level language instruction typically translates into a series of machine-language instructions.

Low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the central processing unit (CPU) which is the "brain" of any computer. Thus, a program written in a low-level language can be extremely efficient, making optimum use of both computer memory and processing time. However, to write a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. Therefore, low-level programming is typically used only for very small programs, or for segments of code that are highly critical and must run as efficiently as possible.

High-level languages permit faster development of large programs. The final program as executed by the computer is not as efficient, but the savings in programmer time generally far outweigh the inefficiencies of the finished product. This is because the cost of writing a program is nearly constant for each line of code, regardless of the language. Thus, a high-level language where each line of code translates to 10 machine instructions costs only one tenth as much in program development as a low-level language where each line of code represents only a single machine instruction.

In addition to the distinction between high-level and low-level languages, there is a further distinction between *compiler languages* and *interpreter languages*. Let's take a look at the various levels.

Absolute Machine Code

The very lowest possible level at which you can program a computer is in its own native machine code, consisting of strings of 1's and 0's and stored as binary numbers. The main problems with using machine code directly are that it is very easy to make a mistake, and very hard to find it once you realize the mistake has been made.

Assembly Language

Assembly language is nothing more than a symbolic representation of machine code, which also allows symbolic designation of memory locations. Thus, an instruction to add the contents of a memory location to an internal CPU register called the *accumulator* might be **add a number** instead of a string of binary digits (*bits*).

No matter how close assembly language is to machine code, the computer still cannot understand it. The assembly-language program must be translated into machine code by a separate program called an *assembler*. The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. At the same time, it also calculates the required address in memory for each symbolic name of a memory location, and substitutes those addresses for the names. The final result is a machine-language program that can run on its own at any time; the assembler and the assembly-language program are no longer needed. To help distinguish between the "before" and "after" versions of the program, the original assembly-language program is also known as the *source code*, while the final machine-language program is designated the *object code*.

If an assembly-language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program.

Compiler Language

Compiler languages are the high-level equivalent of assembly language. Each instruction in the compiler language can correspond to many machine instructions. Once the program has been written, it is translated to the equivalent machine code by a program called a *compiler*. Once the program has been compiled, the resulting machine code is saved separately, and can be run on its own at any time.

As with assembly-language programs, updating or correcting a compiled program requires that the original (source) program be modified appropriately and then recompiled to form a new machine-language (object) program.

Typically, the compiled machine code is less efficient than the code produced when using assembly language. This means that it runs a bit more slowly and uses a bit more memory than the equivalent assembled program. To offset this drawback, however, we also have the fact that it takes much less time to develop a compiler-language program, so it can be ready to go sooner than the assembly-language program.

Interpreter Language

An interpreter language, like a compiler language, is considered to be high level. However, it operates in a totally different manner from a compiler language. Rather, the interpreter program resides in memory, and directly executes the high-level program without preliminary translation to machine code.

This use of an interpreter program to directly execute the user's program has both advantages and disadvantages. The primary advantage is that you can run the program to test its operation, make a few changes, and run it again directly. There is no need to recompile because no new machine code is ever produced. This can enormously speed up the development and testing process.

On the down side, this arrangement requires that both the interpreter and the user's program reside in memory at the same time. In addition, because the interpreter has to scan the user's program one line at a time and execute internal portions of itself in response, execution of an interpreted program is much slower than for a compiled program.