Chapter 1 Notes

Introduction to Computer Programs

A computer program is a sequence of instructions. These instructions are written in a programming language, designed to assist in writing instructions for computers. When a computer is performing the actions specified by the program, we say it is *running* or *executing* the program.

Programs are executed on the computer's hardware, which consists of the physical components of the computer. The hardware includes:

- **CPU**: Executes instructions.
- **Memory**: Where the program is loaded before execution.
- **Interactive Devices**: Allow users to interact with the computer.
- **Storage**: Retains information even when the computer is turned off.

Software refers to the programs executed on the hardware. Programs interact not only with hardware but also with other software on the system.

The term **platform** describes the compatible set of hardware and software that provides the environment for software to run. For example:

- A PC running Windows on an x86-family computer is a platform.

The platform provides useful services for programs. A program that can run on different platforms is **portable**, and modifying a program to run on another platform is called **porting**.

Machine Language

A computer's CPU processes instructions in **machine language** (machine code). The set of all possible machine language instructions is called the **instruction set**. An example of a machine language instruction is:

10110000 01100001

Each instruction is interpreted by the CPU as a command, such as comparing numbers or writing to memory. Writing machine language instructions is time-consuming.

Each instruction is composed of binary digits (*bits*), and the number of bits varies between architectures (e.g., 32-bit instructions, variable-length instructions for x86).

Each family of compatible CPUs (e.g., x86, ARM64) has its own machine language, and these are not compatible between families. A CPU family is called an **Instruction Set Architecture (ISA)**.

Assembly Language

Since machine language is hard for humans to read and understand, **assembly language** was invented. In assembly, each instruction is represented by a short abbreviation. For example, the previous machine instruction could be written in x86 assembly as:

```
mov al, 061h
```

This makes assembly easier to read than machine language. Programs written in assembly are fast and resource-efficient. An **assembler** converts assembly language into machine code. Every ISA has its own assembly language.

Low-Level Languages

Machine language and assembly are considered **low-level languages** because they provide no abstraction. However, they are not portable, as they are tailored to a specific ISA. Writing these programs requires detailed knowledge of the architecture, making them difficult to understand and lengthy. The primary benefit of low-level languages is their speed.

High-Level Languages

High-level languages are portable and easier to understand. There are two ways high-level languages are translated into machine code:

1. **Compilation**: A compiler translates source code into machine code. For example, a C++ compiler converts C++ source code into machine code, often producing an executable file.

2. **Interpretation**: An interpreter directly executes the instructions in the source code without compiling it into machine code. This method is slower but more flexible.

Compilation Process

The process of compiling high-level code is as follows:

$$\text{High-Level Code} \rightarrow \text{Compiler} \rightarrow \text{Executable} \rightarrow \text{Hardware} \rightarrow \text{Program Results}$$

A key advantage of compilers is that they analyze and optimize code, often producing faster executables than manually written machine code.

Interpreted Languages

An interpreter runs the program's source code directly. For example, Python compiles source code into bytecode, which is then interpreted and executed. While interpreters are slower than compilers, they allow for flexibility.

Benefits of High-Level Languages

High-level languages provide several advantages:

- **Abstraction from Architecture**: For example, in `b = 3;`, the programmer doesn't need to know where `b` is stored in memory, only that it exists.
- **Portability**: A C++ program can compile anywhere with a C++ compiler. Programs that run on multiple platforms are called **cross-platform**.
- **Ease of Use**: High-level languages are easier to read, write, and learn. They resemble natural language and mathematical notation.
- **Built-in Capabilities**: High-level languages include features to simplify common programming tasks.

However, some factors can inhibit portability:

- Operating systems offering platform-specific features.
- Third-party libraries available only on certain platforms.
- Compiler-specific extensions.
- C++ features with operating system-defined behavior.

C++ is considered a **mid-level language** because it provides a balance between low-level control and high-level abstraction.