

**Pokhara University**



# **Programming**

**Prepared By:**

**Shiva Raj Paidel**

**Everest Engineering Collage/BE-Computer**

# Introduction

## Programming Languages:

Programming languages are a unique type of software that allows us to create new software (programs). These languages work in close collaboration with the operating system, enabling programmers to utilize specific features and capabilities of the OS while developing applications.

Programming languages are called "languages" because they share characteristics with human languages, such as syntax rules that resemble grammar. These rules dictate how instructions are written, structured, and interpreted.

Programming languages are broadly categorized based on how programmers create instructions:

## Low-Level Language

A **low-level language** is a programming language that operates closer to the hardware and provides minimal abstraction from it. Writing programs in low-level languages requires an in-depth understanding of the hardware for which the program is being developed. Programs written in low-level languages are typically hardware-specific and are not portable across different machine architectures.

Low-level languages are further divided into two types: **machine language** and **assembly language**.

### Machine Language

Machine language is the most basic programming language, understood directly by the computer. It consists entirely of binary numbers (0s and 1s) that represent instructions. While it is efficient for computers, it is difficult and tedious for humans to write and understand. Early computers were often programmed directly using machine language.

### Assembly Language

Assembly language was developed as an improvement over machine language. Instead of using binary numbers, assembly language uses symbols (known as mnemonics) to represent machine instructions. For example, the binary code 78 for an addition operation could be represented as ADD in assembly language.

Since computers cannot directly understand assembly language, a program called an **assembler** is used to convert it into machine language. Although assembly language programs are still hardware-specific, they offer more flexibility than machine language programs. With minor modifications, these programs can run on different machines, making assembly language more practical and easier to learn.

## High-Level Language

A **high-level language** is a programming language designed to be independent of a particular computer's hardware. High-level languages are closer to human languages, making them easier to read, write, and maintain compared to low-level languages. Each high-level language has its own set of rules, known as **syntax**, that defines how instructions are written.

High-level languages emerged in the 1950s, and there are now many examples, including Ada, Algol, BASIC, COBOL, C, C++, FORTRAN, LISP, Pascal, and Prolog.

Programs written in high-level languages must be translated into machine language for execution. This translation can be done using either a **compiler** or an **interpreter**, depending on the language.

High-level languages are widely used because they simplify programming, reduce development time, and provide portability across different systems.

### Compiler

A **compiler** is a program that translates source code written in a high-level programming language into object code. It analyzes the entire program, reorganizes instructions, and converts them into machine language in a process called **compiling**. The resulting object code can be executed directly by the computer.

Compilers are widely used because they produce efficient programs that run faster compared to those executed by interpreters. However, compiling can be time-consuming, especially for large programs. Each compiler is designed for specific hardware or operating systems. For instance, there are distinct C compilers for PCs and Unix-based systems.

### Interpreter

An **interpreter** translates high-level instructions into an intermediate form and immediately executes them line by line. Unlike compilers, interpreters do not generate object code; they execute the source code directly. This allows interpreters to run programs immediately, making them useful during development for testing small code sections incrementally.

However, programs run by interpreters tend to execute slower than those compiled because the translation occurs during runtime.

### Compile

**Compiling** is the process of transforming a high-level programming language (source code) into machine-readable object code. During this process, the compiler checks the source code for syntax errors and informs the programmer about any violations.

After generating object code, a **linker** is used to combine different program modules and assign memory addresses, creating an executable program.

## Source Code

**Source code** refers to a program written in its original form using a specific programming language. This code is human-readable and serves as the starting point for creating software.

To execute a program, the source code must be translated into machine language through a compiler or interpreter. While source code is accessible and modifiable by programmers, end-users typically receive programs in a compiled, machine-language format, which is not human-readable and cannot be modified.

For example:

- **Source Code:** Written in a programming language like C, Python, or Java.
- **Object Code:** The intermediate form produced by a compiler.
- **Executable Code:** The final machine language version ready for execution.

## Object Code

**Object code** is the output generated by a compiler after translating source code. It is typically similar to a computer's machine language but may need further processing to become executable. A program called a **linker** converts the object code into machine language if it is not already in that form, finalizing the program for execution.

---

## Problem Solving Using Computers

Many general-purpose software packages are available in the market, but sometimes users require custom software tailored to specific tasks. **Problem solving using computers** refers to creating such **application software** to meet specific user needs.

### Steps for Developing Application Software:

1. **Problem Analysis**
2. **Algorithm Development and Flowcharting**
3. **Coding**
4. **Compilation and Execution**
5. **Debugging and Testing**
6. **Program Documentation**

## **2.1 Problem Analysis**

**Problem analysis**, also known as defining the problem, involves understanding the problem thoroughly to ensure an appropriate solution is developed. It includes the following tasks:

- 1. Specifying the Objective of the Program**

Define what the program needs to accomplish. This avoids solving the wrong problem. Small problems may have straightforward objectives, but larger, interconnected problems require detailed analysis and coordination.

- 2. Specifying the Outputs**

Determine the desired results or outputs the program must produce.

- 3. Specifying the Input Requirements**

Identify the data or inputs needed to achieve the outputs.

- 4. Specifying the Processing Requirements**

Define the steps or logic needed to transform inputs into outputs.

- 5. Evaluating the Feasibility of the Program**

Assess whether the program can be developed within the available resources, time, and budget.

- 6. Documenting the Program Analysis**

Record all findings and decisions during the analysis phase to guide the next stages of development.

In addition to specifying inputs and outputs, this stage helps evaluate the overall feasibility and estimate costs, ensuring the solution aligns with the user's requirements and constraints.

## Algorithm Development and Flowchart

Once the problem has been identified and analyzed, the next step is to **design a solution** that meets the specified objectives. This is known as the **program design stage**, which involves creating a clear plan for solving the problem.

### Algorithms

An **algorithm** is a step-by-step, ordered description of instructions that must be followed to solve a given task. Essentially, it represents the logical plan of a program in a plain, verbal or written form.

For example, an algorithm to prepare tea:

1. Start.
2. Fetch water, tea leaves, sugar, and milk.
3. Boil the water.
4. Add tea leaves and sugar to the boiling water.
5. Mix with milk.
6. Serve the tea.
7. Stop.

### Basic Guidelines for Writing Algorithms

#### 1. Use Plain Language

Write instructions in simple, everyday language so they are easy to understand.

#### 2. Avoid Language-Specific Syntax

The algorithm should remain valid for any programming language.

#### 3. Be Explicit

Avoid assumptions; describe each step clearly and completely.

#### 4. Single Entry and Exit Point

Ensure the algorithm has one clear starting point (entry) and one clear ending point (exit).

## Important Features of an Algorithm

An algorithm is designed to solve a specific problem or perform a task. To ensure that an algorithm is well-constructed and effective, it must possess the following essential features:

#### 1. Finiteness

Every algorithm should have a definite end, meaning that it should terminate after a finite number of steps. The process should not run indefinitely.

#### 2. Definiteness

Each step of the algorithm must be clearly defined. The actions in the algorithm should be unambiguous, with no room for interpretation. This ensures that the algorithm can be followed precisely without confusion.

### 3. Inputs

An algorithm can have zero or more inputs. Inputs are the data or values provided to the algorithm, either initially or as the algorithm runs. Inputs are necessary for the algorithm to process and produce results.

### 4. Outputs

An algorithm typically results in one or more outputs. These are the quantities or values that are derived based on the inputs and the algorithm's operations. Outputs represent the solution to the problem defined by the algorithm.

### 5. Effectiveness

All operations in the algorithm must be simple and effective enough to be carried out, even with basic tools such as pen and paper. This ensures that the algorithm can be understood and executed without requiring complex tools or machinery.

## Example Algorithm: Sum of Two Numbers

Here is an example of **A simple algorithm that reads two numbers from the user and displays the resulting sum:**

1. **Start**
2. **Read 2 numbers from the user and store them in variables (let's say A and B).**
3. **Store the sum of A and B in variable C.**
4. **Display the value in C.**
5. **Stop**

This algorithm adheres to all five key features:

- **Finiteness:** It terminates after completing a fixed set of steps.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Inputs:** The algorithm accepts two numbers as input.
- **Outputs:** It produces the sum of the two numbers as the output.
- **Effectiveness:** The steps are simple and can easily be carried out using basic arithmetic operations.

By ensuring these features, an algorithm can be designed to be both functional and efficient in solving a given problem.




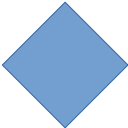


## Flowchart

A **flowchart** is a visual representation of the steps and logic needed to solve a problem or complete a process. It is an essential tool in program design that helps programmers visualize the algorithm and the flow of control in a program. By breaking down complex logic into simpler steps, flowcharts make it easier to understand and implement the solution.

In a flowchart, each step or operation is represented by a specific shape or symbol, with arrows indicating the flow of control between them. This diagrammatic approach simplifies understanding how the program operates and how data is processed.

**Basic Blocks Used for Drawing Flowcharts:**

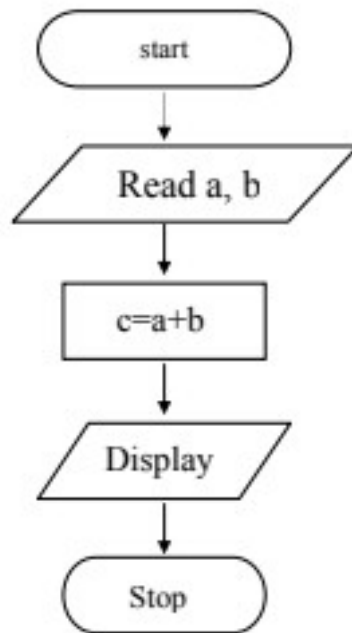
Here’s a table of the common symbols used in flowcharts, along with their descriptions:

Symbol	Name	Purpose
	Start/End	Denotes the start or end of a process or flowchart.
	Process/Action	Represents a process or action step where something is done (e.g., calculation, assignment).
	Input/Output	Used to show input or output operations, such as entering data or displaying results.
	Decision	Represents a decision or branching point where the flow splits based on a condition (Yes/No).
	Flowline	Shows the direction of the process flow from one step to the next.
	Connector	Used to connect parts of the flowchart that are distant from each other or to avoid crossing lines.

This table helps to understand the different flowchart symbols and their roles in representing the sequence of operations or processes within a program.



**For examples: Read 2 numbers form user and display the resulting sum.**



## Coding

**Coding** is the process of writing the actual program based on the design and logic developed in the previous stages. During this phase, the program is written in a programming language, which can be either **high-level** or **low-level** depending on the requirements.

In coding, the goal is to translate the problem-solving logic into a set of instructions that the computer can execute. Good coding practices are essential to ensure the program is maintainable, readable, efficient, and reliable.

## Qualities of a Good Program

### 1. Readability and Understandability:

A good program should be easy to read and understand, even by someone who is not the original programmer. This can be achieved by:

- Using **clear and descriptive variable names**.
- Including **comments** in the code to explain complex logic, algorithms, and the program's flow.

### 2. Efficiency:

An efficient program should make the best use of system resources (such as memory and processing power) and be able to handle larger inputs or tasks without significant performance degradation.

### 3. **Reliability:**

The program should work as expected under all reasonable conditions and consistently produce the correct results.

### 4. **Error Detection and Handling:**

The program should be able to **detect errors** (e.g., invalid input or conditions) and handle them gracefully, without causing the program to crash. It should alert the user or programmer about the issue.

### 5. **Maintainability:**

After the program is deployed, it should be easy to **maintain** and **update**. This includes fixing bugs, adding new features, and ensuring compatibility with future systems or requirements.

## **Key Factors for Writing Efficient Programs**

### 1. **Readability:**

The code should follow consistent patterns that make it easier to follow. This includes proper indentation, organization, and spacing.

### 2. **Descriptive Variable Names:**

Variables should have meaningful names that describe their purpose, which makes the code more understandable.

### 3. **Comments:**

Comments in the code help explain complex sections, making it easier for others (or even the original programmer) to revisit and understand the logic after some time.

## **Compilation and Execution**

Once the program is written, it must be translated into machine language so that the computer can execute it. This translation is done by **compilers** and **interpreters**, which are types of **translators**.

### **Compiled Languages**

In **compiled languages**, a **compiler** translates the entire source code (written by the programmer) into machine language (known as **object code**) before execution. This object code can be saved and run later, making the program execution faster after the initial compilation.

- **Compilation process:** The entire source code is compiled into machine code, which results in a separate file (the object code). The program can be executed immediately or at a later time.
- **Examples** of compiled languages:
  - C
  - C++
  - Fortran
  - COBOL

Advantages of compiled languages:

- Once compiled, the program runs faster because the machine code is already generated.
- The code can be executed independently from the source code.

## Interpreted Languages

In **interpreted languages**, the **interpreter** converts the program statements into machine language one statement at a time as the program is being executed. Unlike compiled languages, no object code is stored, and translation happens in real-time.

- **Execution process:** The interpreter reads and translates each line of code just before execution. This means that translation and execution happen simultaneously.
- **Examples** of interpreted languages:
  - **BASIC**
  - **Python**
  - **JavaScript**

Advantages of interpreted languages:

- Easier to develop, test, and debug, as you don't need to compile the entire code before running it.
- Changes in the program can be tested immediately without waiting for compilation.

## Comparison: Compiled vs. Interpreted Languages

Feature	Compiled Languages	Interpreted Languages
<b>Translation Process</b>	Entire program is translated into machine code before execution.	Translates and executes program one statement at a time.
<b>Object Code</b>	Generates object code that can be saved and executed later.	No object code generated; executed in real-time.
<b>Execution Speed</b>	Faster execution after compilation.	Slower execution, as translation happens during execution.
<b>Development Speed</b>	Slower development due to compilation step.	Faster development since there is no compilation.
<b>Error Detection</b>	Errors are detected during the compilation process.	Errors are detected during execution.

## Which is Better?

- **Compiled languages** are generally better for performance and execution speed, as they convert the code into machine language ahead of time.
- **Interpreted languages** are typically more flexible and easier for rapid development, testing, and debugging, though they tend to execute slower due to the real-time translation process.

Both types of languages have their advantages and can be chosen based on the specific needs of the project.

## The compilation process

The purpose of a compiler is to convert a program written in a high-level programming language (source code) into object code. Programmers write programs in the form of source code, which must undergo several stages before it is transformed into an executable program.

