

UNIT 1

Programming Language

(10 Hours) — C Programming (CACS151), Year I / Sem II

1.1 Introduction to Programming Language

A programming language is a formal, notation-based language consisting of a set of instructions that a programmer writes to communicate with a computer and direct it to perform specific tasks. It provides a structured way of expressing algorithms so that they can be understood by both humans (for readability and maintenance) and machines (for execution).

Every programming language has its own syntax (the set of rules that define valid combinations of symbols) and semantics (the meaning associated with those combinations). A program written in a programming language is called source code, which must eventually be converted into machine-understandable instructions before it can be executed.

Example — A Simple Instruction Set

A programmer wants the computer to add two numbers and display the result. In C, this intent is expressed using precise syntax: `int a = 5, b = 10, sum; sum = a + b; printf("%d", sum);`

The compiler translates this human-readable code into machine instructions the CPU can execute.

Note: Programming languages bridge the gap between human logical thinking and machine-level binary execution.

1.2 Types of Programming Language

Programming languages are broadly classified according to their closeness to human language versus machine hardware. This classification is often visualized as a spectrum, ranging from low-level languages (closer to hardware) to high-level languages (closer to human language).

1.2.1 Machine Language

Machine language is the most basic, low-level language consisting entirely of binary digits (0s and 1s). It is the only language directly understood by the CPU without any translation. It is machine-dependent, extremely fast to execute, but very difficult for humans to write, read, or debug.

1.2.2 Assembly Language

Assembly language uses mnemonic codes (such as MOV, ADD, SUB, JMP) instead of raw binary, making it slightly more human-readable than machine language. It is still hardware-specific and requires an assembler to convert it into machine code.

1.2.3 High-Level Language

High-level languages (such as C, C++, Java, Python) use English-like keywords and mathematical notations, making them portable across machines and much easier to write and maintain. They require a compiler or interpreter to be translated into machine code.

Feature	Machine Language	Assembly Language	High-Level Language
Readability	Very difficult	Moderate (mnemonics)	Easy (English-like)
Machine dependency	Fully dependent	Fully dependent	Portable / independent
Execution speed	Fastest	Very fast	Slower (needs translation)
Example	01001010...	MOV AX, BX	int x = 5;
Translator needed	None	Assembler	Compiler / Interpreter

1.3 Language Processor

A language processor (or language translator) is system software that converts a program written in one language — usually a high-level or assembly language — into a form the computer can execute, typically machine code. The three main types are the Compiler, the Interpreter, and the Assembler.

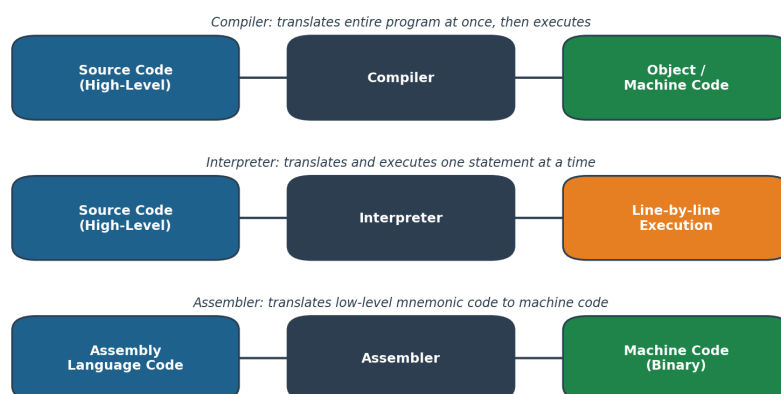


Fig 1.1 — Compiler, Interpreter, and Assembler translation paths

1.3.1 Compiler

A compiler scans and translates the entire source program into machine code (object code) in a single pass before execution begins. It reports all syntax errors together at the end of compilation. Languages like C and C++ use compilers, which generally produce faster-running executables since translation happens only once.

1.3.2 Interpreter

An interpreter translates and executes the source program line by line, without producing a separate object file. Execution stops as soon as an error is encountered on a particular line. Languages such as Python and early BASIC implementations use interpreters, trading some execution speed for easier debugging and interactivity.

1.3.3 Assembler

An assembler is a specialized translator that converts assembly language (mnemonic instructions) into machine code. Since assembly language has a near one-to-one correspondence with machine instructions, assemblers perform a comparatively simple translation.

Aspect	Compiler	Interpreter
Translation unit	Entire program at once	One statement at a time
Error reporting	All errors listed after compilation	Stops at first error
Execution speed	Faster (pre-translated)	Slower (translated on the fly)
Output	Produces object / executable file	No separate object file
Example languages	C, C++, Java (partly)	Python, JavaScript, PHP

1.4 Program Errors

Errors are mistakes in a program that prevent correct compilation or execution. Understanding the categories of errors helps a programmer debug efficiently. The three principal types are syntax errors, runtime errors, and logical errors.

1.4.1 Syntax Errors

Syntax errors occur when the rules (grammar) of the programming language are violated, such as a missing semicolon, unmatched braces, or misspelled keywords. These are detected by the compiler or interpreter before the program runs.

Example — Syntax Error

```
int x = 5 (missing semicolon)
```

The compiler will report: expected ';' before '}' token.

1.4.2 Runtime Errors

Runtime errors occur while the program is executing, even though it compiled successfully. Common causes include division by zero, accessing invalid memory, or array index out of bounds.

Example — Runtime Error

```
int result = 10 / 0;
```

The program compiles without issue but crashes or produces undefined behavior when run.

1.4.3 Logical Errors

Logical errors occur when a program runs and produces output without crashing, but the output is incorrect because the underlying algorithm or logic is flawed. These are the hardest to detect since no error message is generated.

Example — Logical Error

Intending to calculate the average of three numbers but writing:

```
average = a + b + c / 3; instead of average = (a + b + c) / 3;
```

Due to operator precedence, only 'c' is divided by 3, producing a wrong result silently.

Error Type	Detected When	Typical Cause
Syntax Error	During compilation	Grammar / punctuation violation
Runtime Error	During execution	Invalid operation (e.g. divide by zero)
Logical Error	After execution (wrong output)	Flawed algorithm or reasoning

1.5 Features of a Good Program

A well-written program is judged not only by whether it produces correct output, but also by qualities that make it reliable, maintainable, and efficient over its lifetime.

- **Correctness** — produces the expected, accurate output for all valid inputs.

- Readability — uses meaningful variable names, consistent indentation, and comments.
- Efficiency — uses minimal time and memory resources (optimal time and space complexity).
- Portability — can run on different platforms/machines with little or no modification.
- Robustness — handles unexpected or invalid input gracefully without crashing.
- Modularity — organized into independent functions or modules for reuse and testing.
- Maintainability — easy to modify, extend, or debug in the future.
- User-friendliness — provides clear prompts, instructions, and error messages.

1.6 Different Programming Paradigms

A programming paradigm is a fundamental style or approach used to structure and write programs. Different paradigms offer different ways of thinking about problems and organizing solutions.

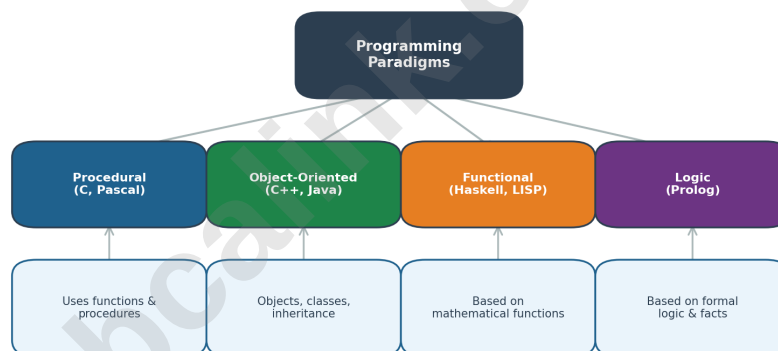


Fig 1.2 — Major programming paradigms

Paradigm	Core Idea	Example Languages
Procedural	Sequence of instructions grouped into procedures/functions	C, Pascal, FORTRAN
Object-Oriented	Organizes code around objects that bundle data and behaviour	C++, Java, Python
Functional	Computation as evaluation of mathematical functions; avoids state	Haskell, LISP, Erlang

Paradigm	Core Idea	Example Languages
Logic	Facts and rules; the system infers answers via logical reasoning	Prolog
Structured	Uses control structures (sequence, selection, iteration), avoids GOTO	C, Ada

Note: C primarily follows the procedural (and structured) programming paradigm, organizing code as a sequence of function calls.

1.7 Software Development Model

A software development model (or Software Development Life Cycle model) is a structured approach used by teams to plan, design, build, test, and maintain software systematically.

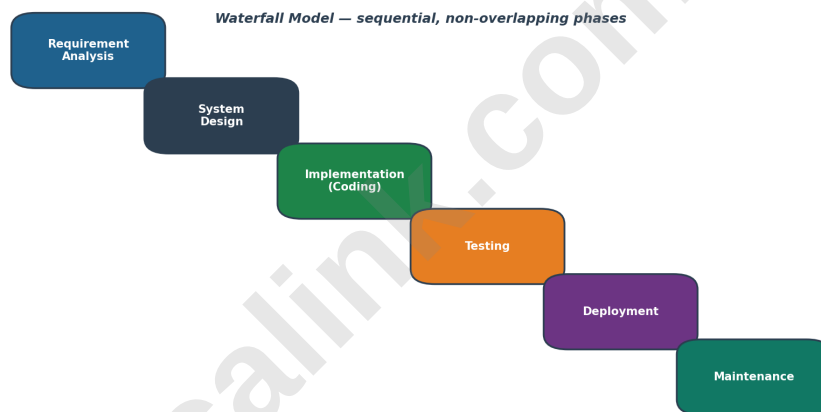


Fig 1.3 — Waterfall software development model

1.7.1 Waterfall Model

The Waterfall model is the earliest and simplest SDLC model, following a linear, sequential flow where each phase must be completed before the next begins: Requirement Analysis, System Design, Implementation, Testing, Deployment, and Maintenance. It works best for projects with clear, unchanging requirements.

1.7.2 Other Common Models

- Spiral Model — combines iterative development with systematic risk analysis, repeating through planning, risk analysis, engineering, and evaluation phases.
- Iterative / Incremental Model — the system is built and delivered in small increments, each adding functionality to the previous version.

- Agile Model — emphasizes flexibility, collaboration, and delivering working software in short cycles called sprints.
- V-Model (Verification and Validation) — extends the waterfall model by pairing each development phase with a corresponding testing phase.

Model	Best Suited For	Key Characteristic
Waterfall	Stable, well-understood requirements	Sequential, no overlap
Spiral	Large, high-risk projects	Repeated risk analysis cycles
Iterative	Projects needing early partial delivery	Builds in increments
Agile	Rapidly changing requirements	Short sprints, continuous feedback

1.8 Program Development Life Cycle (PDLC)

The Program Development Life Cycle describes the sequence of steps a programmer follows to convert a problem statement into a working, tested program.



Fig 1.4 — Program Development Life Cycle

- 1. Problem Analysis — understand the requirements, inputs, outputs, and constraints of the problem.
- 2. Algorithm / Flowchart Design — plan the logical steps or visual flow to solve the problem.
- 3. Coding — translate the algorithm into a specific programming language's syntax.
- 4. Compilation & Debugging — translate the source code and fix any syntax errors.
- 5. Testing — run the program with sample and edge-case data to verify correctness.

- 6. Documentation & Maintenance — record how the program works and update it as needed over time.

Note: The PDLC is cyclical in practice — testing often reveals issues that send the programmer back to earlier stages.

1.9 System Design Tools

Before writing actual code, programmers use design tools to plan the logic of a solution in a language-independent way. The most common tools are algorithms, flowcharts, pseudocode, and decision tables.

1.9.1 Algorithm

An algorithm is a finite, ordered set of well-defined steps written in plain language (or structured English) that solves a specific problem.

Example — Algorithm to find the largest of two numbers

Step 1: Start

Step 2: Read two numbers A and B

Step 3: If $A > B$, print A as the largest; otherwise print B as the largest

Step 4: Stop

1.9.2 Flowchart

A flowchart is a graphical/pictorial representation of an algorithm using standardized geometric symbols (oval for start/stop, parallelogram for input/output, rectangle for process, diamond for decision) connected by arrows showing the flow of control.

1.9.3 Pseudocode

Pseudocode is an informal, high-level description of a program's logic that uses structured English combined with programming-like constructs (IF, WHILE, FOR), without following the strict syntax of any particular language.

1.9.4 Decision Table

A decision table is a tabular tool used to represent complex conditional logic, listing all possible combinations of conditions and the corresponding actions to take for each combination.

Tool	Nature	Best Used For
Algorithm	Textual, step-by-step	General problem-solving logic
Flowchart	Graphical, symbol-based	Visualizing control flow
Pseudocode	Semi-formal, English + code-like	Bridging algorithm and real code
Decision Table	Tabular	Complex multi-condition logic

Key Takeaways

- ✓ A programming language lets humans express instructions that a computer can eventually execute.
- ✓ Languages range from low-level (machine, assembly) to high-level (C, Java, Python).
- ✓ Language processors — compilers, interpreters, and assemblers — translate source code to machine code.
- ✓ Errors fall into three categories: syntax, runtime, and logical.
- ✓ A good program is correct, readable, efficient, portable, robust, modular, and maintainable.
- ✓ Programming paradigms (procedural, OOP, functional, logic) define different ways of structuring code.
- ✓ Software development models (Waterfall, Spiral, Iterative, Agile) guide the overall project lifecycle.
- ✓ The Program Development Life Cycle takes a problem from analysis through coding, testing, and documentation.
- ✓ System design tools — algorithms, flowcharts, pseudocode, decision tables — plan logic before coding.

— End of Unit 1 —