369 TESLA | KALVIYOGI

UNIT VI

PROGRAMMING WITH C++

Visual notes
Real compiled programs
3D infographics
TRB exam theory




```
class Student {
int roll;
public:
void show();
};
template <typename T>
T maxOf(T a, T b);
try { throw 1; }
catch (int e) { }
ifstream fin("a.txt");
virtual void draw() = 0;
cout << "Hi";
```



Kalviyogi Nagarajan

EDUNIVRA PVT LTD

369 TESLA | KALVIYOGI

UNIT VI

PROGRAMMING WITH C++

Complete Visual Notes with Real Compiled Programs, Output Windows, 3D Infographics and TRB Exam Theory

Kalviyogi Nagarajan

EDUNIVRA PVT LTD

Programming with C++ — Unit VI
Kalviyogi Nagarajan · 369 TESLA | KALVIYOGI

Published by EDUNIVRA PVT LTD.

This edition is a redesigned and edited version of the author's Unit VI notes (Language Design and Translation Issues · Elementary Data Types · Object-Oriented Programming · Basics of C++ · Functions · Class and Objects · Files and Event Handling). Every program screen in this book was compiled as ISO C++17 with Clang and run; the output windows show the real output of the compiled program, and text typed at the keyboard is shown in yellow. Technical statements were checked and corrected where needed, and the figures are redrawn as vector 3D illustrations.

For the TRB Computer Instructor examination, every chapter ends with **Exam Theory Notes**: key facts plus predict-the-output items whose results were verified by compiling and running them. The appendices add quick-reference tables and a last-day revision.

Sizes such as `sizeof(int)` and `sizeof(long)` depend on the compiler and platform; the values shown are from 64-bit Windows (where `long` is 4 bytes; on 64-bit Linux it is 8).

Contents

Language Design and Translation Issues 7

Chapter 1 · Programming Language Concepts 8
- 1.1 Meaning of Programming Language 8
- 1.2 Need for Programming Languages 9
- 1.3 Characteristics of a Good Programming Language 10
- 1.4 Language Design Goals 11
- 1.5 Readability, Writability, Reliability and Efficiency 12

Chapter 2 · Basic Language Concepts 14
- 2.1 Syntax and Semantics 14
- 2.2 Tokens, Lexemes and Patterns 15
- 2.3 Identifiers and Keywords 16
- 2.4 Variables and Constants 17
- 2.5 Expressions and Statements 18
- 2.6 Control Structures 19
- 2.7 Data Abstraction 21
- 2.8 Procedural Abstraction 22

Chapter 3 · Programming Paradigms 24
- 3.1 Imperative Programming Paradigm 24
- 3.2 Object-Oriented Programming Paradigm 25
- 3.3 Functional Programming Paradigm 26
- 3.4 Logic Programming Paradigm 27
- 3.5 Event-Driven Programming Paradigm 28
- 3.6 Concurrent Programming Paradigm 30
- 3.7 Comparative Study of Paradigms 31

Chapter 4 · Programming Models 32
- 4.1 Von Neumann Model 32
- 4.2 Declarative Model 33
- 4.3 Object-Based Model 34
- 4.4 Dataflow Model 35
- 4.5 Concurrent and Parallel Models 37
- 4.6 Model-Based Language Selection 38

Chapter 5 · Programming Environments 39
- 5.1 Programming Environment 39
- 5.2 Editors and Integrated Development Environments 40
- 5.3 Compilers and Interpreters 41
- 5.4 Debuggers and Profilers 42
- 5.5 Linkers and Loaders 43
- 5.6 Runtime Environment 44
- 5.7 Development Tools and Libraries 45

Chapter 6 · Virtual Computers 47
- 6.1 Concept of Virtual Computer 47
- 6.2 Machine Language and Assembly Language 48
- 6.3 High-Level Language and Virtual Machine 49
- 6.4 Abstract Machine 50
- 6.5 Java Virtual Machine 51
- 6.6 Role of Virtual Computers in Portability 52

6.7 Advantages and Limitations of Virtual Computers 53

Chapter 7 · Binding and Binding Times 54

7.1 Meaning of Binding 54
7.2 Types of Binding 55
7.3 Static Binding 56
7.4 Dynamic Binding 57
7.5 Binding Time Concepts 59
7.6 Language Design Time 60
7.7 Language Implementation Time 61
7.8 Compile Time 62
7.9 Link Time 64
7.10 Load Time 65
7.11 Run Time 66
7.12 Importance of Binding in Language Translation 68

Chapter 8 · Syntax of Programming Languages 70

8.1 Definition of Syntax 70
8.2 Syntax Rules 71
8.3 Grammar and Production Rules 73
8.4 Context-Free Grammar 74
8.5 Backus–Naur Form 75
8.6 Extended Backus–Naur Form 77
8.7 Syntax Trees 78
8.8 Parse Trees 80
8.9 Ambiguity in Grammar 81
8.10 Syntax Errors 82

Chapter 9 · Language Translation 85

9.1 Need for Language Translation 85
9.2 Source Program and Target Program 86
9.3 Compiler 87
9.4 Interpreter 89
9.5 Assembler 90
9.6 Preprocessor 92
9.7 Cross Compiler 93
9.8 Translator Design Issues 95
9.9 Comparison between Compiler and Interpreter 96

Chapter 10 · Stages in Translation 98

10.1 Lexical Analysis 99
10.2 Syntax Analysis 101
10.3 Semantic Analysis 103
10.4 Intermediate Code Generation 104
10.5 Code Optimization 106
10.6 Code Generation 107
10.7 Symbol Table Management 108
10.8 Error Detection and Recovery 110
10.9 Front End and Back End of Compiler 112

Chapter 11 · Formal Translation Models 114

11.1 Introduction to Formal Models 115
11.2 Finite Automata 116
11.3 Regular Expressions 119
11.4 Context-Free Grammar 121

11.5 Pushdown Automata 124
11.6 Turing Machine 126
11.7 Role of Automata in Translation 129
11.8 Formal Specification of Syntax 131
11.9 Formal Specification of Semantics 132

PART I · UNIT VI SYLLABUS 1

Language Design and Translation Issues

Chapter 1 Programming Language Concepts

Programming language concepts · paradigms and models · programming environments · virtual computers and binding times · syntax · stages in translation · formal transition models.

CHAPTER · UNIT VI · LANGUAGE DESIGN

1 Programming Language Concepts

OPENING SITUATION

In the college laboratory two students solve the same problem: one in C++, one in Python. Both programs give the same answer, yet one is shorter, one runs faster, and one is easier to read a month later. This chapter explains what a programming language is, why we need one, and the qualities (readability, writability, reliability, efficiency) that we use to judge a language.

IN THIS CHAPTER

- 1.1 Meaning of Programming Language
- 1.2 Need for Programming Languages
- 1.3 Characteristics of a Good Programming Language
- 1.4 Language Design Goals
- 1.5 Readability, Writability, Reliability and Efficiency

YOU NEED

- Everyday use of a computer.
- No programming experience is assumed.

1.1 Meaning of Programming Language

DEFINITION 1.1 — PROGRAMMING LANGUAGE

A **programming language** is a formal system of **symbols, rules and syntax** used to write instructions that a computer can carry out after **translation**. It lets humans express algorithms, logic and solutions in a form a computer can execute.

1 Key idea

- A bridge between human thinking and computer execution.
- Programs are written as **source code**.
- A **translator** (compiler or interpreter) turns it into machine instructions.
- The computer executes them and produces results.

2 Why it matters

- **Communication** — precise instructions to computers.
- **Abstraction** — hides machine details.
- **Portability** — runs on other systems after translation.
- **Reusability** and **productivity** — faster, more reliable software.

Table 1.1 Core elements of a programming language

Element	Meaning	C++ example
Syntax	Rules that define the structure (form) of statements	`int x = 5;` needs the `;`
Semantics	The meaning of a valid statement	`x = x + 1` increases x by one
Symbols	Keywords, identifiers, operators, constants, punctuation	`int`, `total`, `+`, `10`, `{ }`
Rules	How symbols combine to form correct programs	a variable is declared before use
Execution	Running the translated program to perform the task	compile, link, run `a.exe`

The smallest complete C++ program prints one line. The editor window on the left is the source file; the dark window on the right is what the computer really printed when the compiled program ran.

C++ hello.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, C++!" << endl;
    return 0;
}
```

▶ Output

```
Hello, C++!
```

Screen 1.1 Source code (left) becomes machine code through the compiler; the running program prints the message (right).

Human language	Programming language
• Communication between people	• Communication with computers
• Often ambiguous, flexible grammar	• Strict and unambiguous, formal rule-based grammar
• Understood by humans; no execution by itself	• Understood by machines after translation; executed to produce output

REAL-LIFE VIEW

A recipe is a program for a cook: exact steps in a fixed order. A programming language is the "recipe language" for a computer — every step must be exact, because the computer cannot guess. Mobile apps, banking systems, games and AI tools are all written this way. Typical examples: **C** (system programming), **C++** (system + object-oriented), **Python** (high-level, easy), **Java** (platform independent), **SQL** (querying databases).

★ **A programming language is a formal system of symbols, rules and syntax for writing instructions that are translated and executed by a computer.**

1.2 Need for Programming Languages

Programming languages are needed to **bridge the gap** between humans and computers: they let us express ideas and instruct computers in a simple, efficient and reliable way.

✗ Why not machine code?

- Only 0s and 1s — hard for humans to read.
- **Machine dependent** — changes with the hardware.
- Difficult to write, debug and maintain; slow and error-prone.

✓ Major reasons

- **Ease of communication** and **abstraction** (what to do, not how).
- **Reduced complexity**, **portability**, **maintainability**.
- **Faster development**, **fewer errors**, **reusability** (functions, libraries, modules).

Table 1.2 Levels of languages compared

Feature	Machine language	Assembly language	High-level language
Ease of writing	Very difficult	Difficult	Easy
Readability	Very low (0s and 1s)	Low (mnemonics)	High (English-like)
Portability	None (hardware dependent)	Low (machine dependent)	High (mostly portable)
Development time	Very long	Long	Short
Error rate	Very high	High	Low
Maintenance	Very difficult	Difficult	Easy
Example	`10110000 01100001`	`MOV AL, 61h`	`total = a + b;`

D Benefits to developers

Simpler syntax; focus on logic, not hardware; easier debugging and testing; code reuse; more productivity.

O Benefits to organisations

Lower development and maintenance cost; faster time to market; better quality; easy to scale; competitive advantage.

EXAM POINT

Languages provide abstraction and human-friendly syntax; they reduce complexity and improve portability and maintainability. **No single language is best for every application** — C for systems, Python for data science, SQL for databases.

★ **Programming languages make it easy, efficient and reliable for humans to instruct computers.**

1.3 Characteristics of a Good Programming Language

A good language helps developers express solutions **clearly, correctly and efficiently**: it reduces errors, saves time, and makes programs easy to write, test and maintain.

Table 1.3 Desirable features

Characteristic	What it means	Why it matters
Simplicity	Easy to learn and use; few basic constructs	Shorter learning curve, fewer errors
Clarity	Clear, readable syntax	Code is easy to understand
Consistency (orthogonality)	The same rules used everywhere	Predictable, less confusing
Expressiveness	Ideas can be expressed naturally	Write more with less code
Reliability	Correct and predictable results	Builds trust in software
Efficiency	Good use of time and memory	Better performance
Portability	Runs on different platforms	Easy to move and deploy
Maintainability	Easy to test, fix and update	Lower cost over time
Security	Protects data, prevents misuse	Safe from attacks and failures

Readability and simplicity

Simple syntax, good naming and indentation reduce cognitive load and errors.

Reliability and security

Strong type checking and error handling detect bugs early and block invalid input.

Portability and maintainability

Runs on many OS and devices; modular design makes code easy to test and extend.

★ **A good language is simple, clear, consistent, reliable, efficient, portable, maintainable and secure.**

1.4 Language Design Goals

DEFINITION 1.2 — DESIGN GOALS

Language design goals are the qualities a language aims to achieve. They guide designers' decisions about features, syntax, semantics and implementation. The major goals: **readability, writability, reliability, efficiency, simplicity, portability, maintainability, security**.

Trade-offs in design

- **Power vs simplicity** — more power may add complexity.
- **Flexibility vs safety** — flexible features (pointers) may add risk.
- **Speed vs readability** — highly optimised code is harder to read.
- **Rich features vs learnability** — many features overwhelm beginners.
- **Performance vs portability** — platform-specific tuning reduces portability.

Real languages, different priorities

- **Python** — readability, simplicity, productivity.
- **Java** — portability, reliability, security.
- **C** — efficiency, control, portability.
- **Rust** — safety, reliability, performance.
- **C++** — efficiency *and* abstraction (classes, templates).

Table 1.4 How languages score on design goals (✓✓✓ strong · ✓✓ good · ✓ basic)

Goal	Python	Java	C	Rust
Readability	✓✓✓	✓✓	✓	✓✓
Writability	✓✓✓	✓✓	✓	✓✓
Reliability	✓✓	✓✓✓	✓	✓✓✓
Efficiency	✓✓	✓✓	✓✓✓	✓✓✓
Portability	✓✓✓	✓✓✓	✓✓	✓✓✓
Security	✓✓	✓✓	✓	✓✓✓
Simplicity	✓✓✓	✓✓	✓	✓✓

REAL-LIFE VIEW

No goal is absolute. For **embedded systems** (a washing-machine controller) efficiency and safety dominate; for **web applications** readability and maintainability matter more.

★ **Design goals define the qualities a language should achieve; designers balance them through trade-offs.**

1.5 Readability, Writability, Reliability and Efficiency

These four are the main **evaluation criteria** for comparing programming languages.

1 Readability

How easily code can be **understood**. Helped by meaningful names, consistent style, simple syntax. Lowers learning time and maintenance cost.

2 Writability

How easily programs can be **created and modified**. Depends on **simplicity, orthogonality** and **expressive power**.

3 Reliability

Programs behave **correctly and predictably**. Needs **type checking, exception handling**, safe memory use (bounds checking, garbage collection).

4 Efficiency

Good use of **time, memory** and other resources: fast execution, low overhead, optimised compilation.

The two programs below compute the same value. The compiler does not care about names, but a reader does: readability is a property for *humans*.

C++ readability.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int a=40,b=3;cout<<a*b<<endl;      // hard to read
    int pricePerBook = 40, books = 3;      // easy to read
    int totalCost = pricePerBook * books;
    cout << "Total cost = " << totalCost << endl;
    return 0;
}
```

▶ Output

```
120
Total cost = 120
```

Screen 1.2 Same result, very different readability: meaningful names and spacing make the second version self-explaining.

Reliability comes partly from **type checking**. C++ checks types at compile time, so this mistake is caught before the program can ever run:

C++ typecheck.cpp

```cpp
#include <string>
int main() {
    int marks = 90;
    std::string name = "Asha";
    marks = name;          // a string cannot be stored in an int
    return 0;
}
```

✕ Compiler — build failed

```
typecheck.cpp:5:13: error: assigning to 'int' from incompatible type 'std::string' (aka
'basic_string<char>')
1 error generated.
```

Screen 1.3 The compiler rejects the assignment: static type checking is one source of reliability.

Table 1.5 The four criteria at a glance

Criterion	Meaning	Key factors	Effect on software quality
Readability	Ease of understanding code	Meaningful names, simple syntax, clarity	Better understandability, easy maintenance
Writability	Ease of writing and modifying	Simplicity, orthogonality, expressive power	Higher productivity, fewer errors
Reliability	Correct, predictable behaviour	Type checking, error handling, safe memory	Fewer defects, trust
Efficiency	Effective use of time and resources	Execution speed, memory use, low overhead	Better performance, scalability

COMMON MISTAKE 1.1 — "READABILITY AND WRITABILITY ARE THE SAME"

They often conflict. A very short, clever expression may be easy to *write* but hard to *read*. Readability is judged by the reader; writability by the writer.

★ **Readability = easy to understand; writability = easy to write; reliability = trustworthy; efficiency = performs well.**

Chapter 1 summary

- A programming language = symbols + rules + syntax, translated and executed by a computer.
- Syntax is form; semantics is meaning.
- Machine code is binary and hardware dependent; high-level languages are readable and portable.
- Good language: simple, clear, consistent, reliable, efficient, portable, maintainable, secure.
- Design involves trade-offs (power vs simplicity, flexibility vs safety, speed vs readability).
- Four evaluation criteria: readability, writability, reliability, efficiency.

Next: Chapter 2 — the basic building blocks every language shares: syntax, tokens, identifiers, variables, statements and abstraction.

CHAPTER · UNIT VI · LANGUAGE DESIGN

2 Basic Language Concepts

OPENING SITUATION

A student's program will not compile because of one missing semicolon; another compiles perfectly but prints the wrong answer. The first is a *syntax* problem, the second a *semantic* one. This chapter covers the building blocks shared by all languages: syntax and semantics, tokens, identifiers and keywords, variables and constants, expressions and statements, control structures and abstraction.

IN THIS CHAPTER

- 2.1 Syntax and Semantics
- 2.2 Tokens, Lexemes and Patterns
- 2.3 Identifiers and Keywords
- 2.4 Variables and Constants
- 2.5 Expressions and Statements
- 2.6 Control Structures
- 2.7 Data Abstraction
- 2.8 Procedural Abstraction

YOU NEED

- Chapter 1: what a programming language is.

2.1 Syntax and Semantics

DEFINITION 2.1 — SYNTAX AND SEMANTICS

Syntax is the grammatical structure (form) of statements in a language. **Semantics** is the meaning or effect of those statements when executed.

Syntax — how code is written

- Follows the grammar rules of the language.
- Checked by the **compiler / interpreter**.
- Error → **syntax error**: the program will not compile or run.
- Example: missing `;`, misspelt keyword.

Semantics — what code does

- Ensures the program performs the intended operation.
- Wrong semantics shows up **during execution**.
- Error → **logical / semantic error**: runs but gives wrong output.
- Example: wrong condition, wrong formula.

Both programs below have correct *syntax*. The second one has a semantic error: `if (x = 0)` **assigns** 0 to x (the value 0 is false), instead of **comparing** with `==`.

C++ semantics.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    if (x == 0) cout << "Zero\n";
    else        cout << "Not zero, x = " << x << "\n";
    if (x = 0)  cout << "Zero\n";          // = assigns!
    else        cout << "Not zero, x = " << x << "\n";
    return 0;
}
```

▶ Output

```
Not zero, x = 5
Not zero, x = 0
```

Screen 2.1 Correct syntax, wrong semantics: the second test silently changed x to 0 and took the `else` branch.

C++ syntax.cpp

```cpp
#include <iostream>
int main() {
    int x = 10
    std::cout << x;
}
```

✕ Compiler — build failed

```
syntax.cpp:3:15: error: expected ';' at end of declaration
1 error generated.
```

Screen 2.2 A syntax error: the compiler stops at the missing semicolon and no program is produced.

EVERYDAY ANALOGY

Syntax is grammar: "Book the read I" is grammatically wrong. **Semantics is meaning**: "The book drank the river" is grammatical but meaningless. A program needs both correct syntax and correct semantics to be useful.

★ **Syntax is the structure of a program and semantics is its meaning — both must be correct.**

2.2 Tokens, Lexemes and Patterns

Lexical analysis, the compiler's first phase, breaks source code into meaningful units described by three concepts.

T Token

A **category or class** of lexemes: identifier, keyword, operator, constant, delimiter.

L Lexeme

The **actual character sequence** in the source that matches a token — a concrete instance. In class IDENTIFIER the lexeme may be `count`.

P Pattern

The **rule** (usually a regular expression) describing the form of lexemes for a token: identifier → `[A-Za-z_][A-Za-z0-9_]*`.

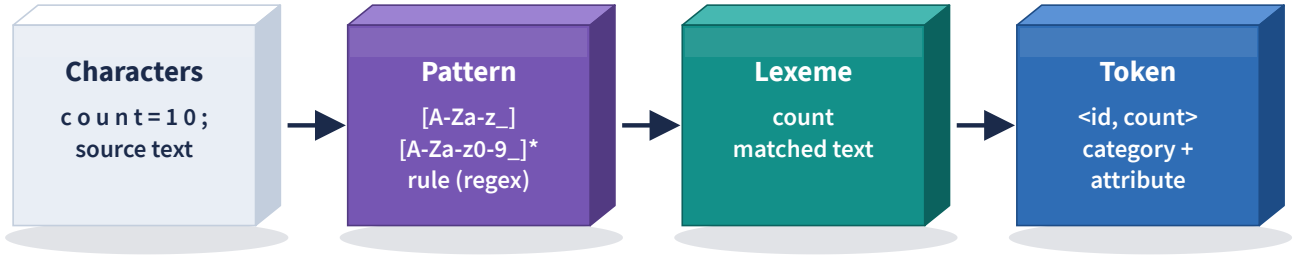


Figure 2.1 The lexical analyser reads characters, matches a pattern, and emits a token whose attribute is the lexeme.

Table 2.1 Tokens, lexemes and patterns

Token (category)	Lexeme (example)	Pattern (rule)
Keyword	`while`	`while`
Identifier	`count`	`[A-Za-z_][A-Za-z0-9_]*`
Relational operator	`>=`	`< | <= | == | != | > | >=`
Integer constant	`123`	`[0-9]+`
Delimiter	`;`	`; , ( ) { }`

The statement `sum = a + 10;` contains **6 tokens**: `sum` (identifier), `=` (operator), `a` (identifier), `+` (operator), `10` (constant), `;` (delimiter). Spaces, tabs and comments are removed and never become tokens.

EXAM POINT

Token = category · **Lexeme** = actual text · **Pattern** = rule. Lexical analysis uses patterns to find lexemes and assign tokens; the lexeme may be passed as an attribute value to later phases.

★ **Tokens are categories, lexemes are actual character sequences, and patterns are the rules that define valid lexemes.**

2.3 Identifiers and Keywords

Identifiers

User-defined names for variables, functions, classes and objects: `totalMarks`, `computeSum`, `studentCount`. Created by the programmer.

Keywords

Reserved words with a predefined meaning, part of the language syntax; cannot be identifiers: `while if return int float class for break else switch`. Created by the language designer.

Rules for identifiers (C++)

- Start with a letter (A–Z, a–z) or underscore `_`; never a digit.
- Only letters, digits and underscores — no spaces or special characters.
- Cannot be a keyword.
- **Case-sensitive**: `Total` and `total` differ.
- Choose meaningful, descriptive names.

Valid or invalid?

Valid: `totalMarks` `computeSum` `_count` `student2` `my_value`

Invalid: `2sum` (digit first) · `total marks` (space) · `my-value` (hyphen) · `@count` (symbol) · `while`, `return` (keywords)

C++ identifiers.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int total = 10, Total = 20;   // case-sensitive: two variables
    int student2 = 3, _count = 4;
    cout << total << " " << Total << " " << student2 + _count << endl;
    int 2sum = 5;                 // invalid: starts with a digit
    return 0;
}
```

✕ Compiler — build failed

```
identifiers.cpp:7:9: error: expected unqualified-id
1 error generated.
```

Screen 2.3 `total` and `Total` are different names, but `2sum` stops compilation.

COMMON MISTAKE 2.1 — "ANY NAME STARTING WITH _ IS FINE"

It compiles, but in C++ names that begin with an underscore followed by a capital letter (`_Count`) or contain a double underscore (`my__var`) are **reserved for the compiler and library**. Avoid them in your own code.

★ **Identifiers are user-defined names; keywords are reserved words with a fixed meaning.**

2.4 Variables and Constants

DEFINITION 2.2 — VARIABLE AND CONSTANT

A **variable** is a named memory location whose value **can change** during execution. A **constant** is a named memory location whose value **remains fixed**. Both have a **name, type and value**.

Table 2.2 Variable versus constant

Aspect	Variable	Constant
Can change?	Yes	No
Declaration	`type name = value;`	`const type name = value;`
Purpose	Data that changes during execution	Values that must stay the same
Example	`age = 18` , daily temperature, user score	`PI = 3.14159` , days in a week (7)

C++ varconst.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    int count = 5;              float temperature = 36.6f;
    char grade = 'A';           string name = "Tesla";
    bool isValid = true;
    const float PI = 3.14159f;  const int DAYS_IN_WEEK = 7;
    count = count + 1;          // a variable can change
    cout << count << " " << temperature << " " << grade << " "
         << name << " " << isValid << " " << PI * 2 << " " << DAYS_IN_WEEK << endl;
    return 0;
}
```

▶ Output

```
6 36.6 A Tesla 1 6.28318 7
```

Screen 2.4 Common types: `int`, `float`, `char`, `string`, `bool` (printed as 1 for `true`). `const` values are used like variables but cannot be changed.

C++ const_error.cpp

```cpp
int main() {
    const int DAYS_IN_WEEK = 7;
    DAYS_IN_WEEK = 8;       // not allowed
}
```

✕ Compiler — build failed

```
const_error.cpp:3:18: error: cannot assign to variable 'DAYS_IN_WEEK' with const-qualified type
'const int'
1 error generated.
```

Screen 2.5 Constants prevent accidental modification: the compiler refuses to change a `const`.

★ **Variables hold changeable data; constants hold fixed data needed by programs.**

2.5 Expressions and Statements

DEFINITION 2.3 — EXPRESSION AND STATEMENT

An **expression** is a combination of operands and operators that **produces a value** (`x + y * 2`). A **statement** is a complete instruction **executed** by the program (`total = a + b;`).

E Types of expressions

- **Arithmetic**: `a + b * 2`
- **Relational**: `x > y`
- **Logical**: `(a > 0) && (b < 10)`
- **Assignment**: `x = y + 5`
- **Conditional** (ternary): `(x > 0) ? x : -x`

S Types of statements

- **Declaration**: `int x;`
- **Assignment**: `x = 10;`
- **Input/output**: `cin >> x; cout << x;`
- **Selection**: `if (x > 0) { … }`
- **Looping**: `for (i = 0; i < n; i++) { … }`
- **Return**: `return sum;`

Precedence decides which operator is evaluated first; **associativity** decides the order for operators of equal precedence; parentheses override both. So `a + b * c` means `a + (b * c)`.

C++ expr.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int a = 2, b = 3, c = 4, x = -7;
    cout << a + b * c << endl;          // 2 + (3*4)
    cout << (a + b) * c << endl;        // parentheses first
    cout << (a > b) << " " << (a < b && c > 0) << endl;
    cout << ((x > 0) ? x : -x) << endl;  // absolute value
    return 0;
}
```

Output

```
14
20
0 1
7
```

Screen 2.6 Arithmetic, relational, logical and conditional expressions; relational and logical results print as 1 (true) or 0 (false).

COMMON MISTAKE 2.2 — "EVERY STATEMENT ENDS WITH A SEMICOLON"

Simple statements do (`x = y + 2;`), but a **compound statement** (block) `{ … }` and the control statements `if`, `for`, `while` ending in a block do not need one. Also, `x > 10;` is a legal but useless *expression statement*, not an assignment.

EXAM POINT

Expressions produce values; statements perform actions. Every expression can be used inside a statement, but not every statement produces a value.

★ **Expressions compute values; statements tell the computer what action to perform.**

2.6 Control Structures

DEFINITION 2.4 — CONTROL STRUCTURES

Control structures determine the order in which instructions execute: **sequence** (one after another), **selection** (choice by a condition) and **iteration** (repetition while a condition holds).

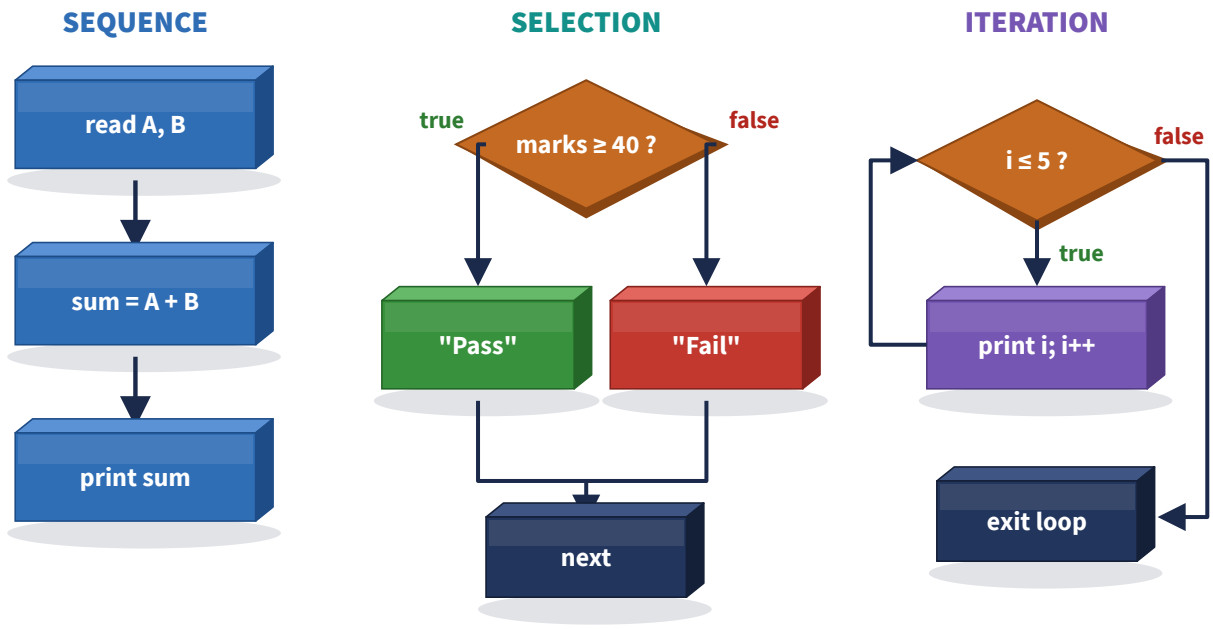


Figure 2.2 The three basic control structures. Any algorithm can be built from these three (the structured-programming theorem of Böhm and Jacopini).

Table 2.3 Selection, iteration and jump statements in C++

Statement	When to use	How it works
`if` / `if-else`	One or two paths	Runs a block if the condition is true (else the other block)
nested `if`	Multi-level decisions	`if` inside another `if`
`switch`	Many options on one value	Jumps to the matching `case`
`while`	Repetitions unknown; check first	Condition → block → repeat; may run 0 times (entry-controlled)
`do-while`	Block must run at least once	Block → condition → repeat (exit-controlled)
`for`	Count known / counter-controlled	init → condition → block → update → repeat
`break`	Leave a loop or switch	Exits immediately
`continue`	Skip one iteration	Jumps to the next iteration
`return`	Leave a function	Exits, optionally returning a value
`goto`	Rarely	Jumps to a labelled statement (reduces readability)

C++ control.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int a = 4, b = 6;
    int sum = a + b;                    // sequence
    if (sum % 2 == 0) cout << sum << " is even\n";   // selection
    else              cout << sum << " is odd\n";
    for (int i = 1; i <= 5; i++) {      // iteration
        if (i == 3) continue;           // skip 3
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Output

```
10 is even
1 2 4 5
```

Screen 2.7 Sequence, selection and iteration in one program; `continue` skips the value 3.

EVERYDAY ANALOGY

Driving to college: **sequence** — follow the road; **selection** — turn left or right at a signal; **iteration** — keep moving until you reach the destination.

★ **Control structures organise decisions and repetition so programs solve problems step by step.**

2.7 Data Abstraction

DEFINITION 2.5 — DATA ABSTRACTION AND ADT

Data abstraction focuses on **what** data represents and **what operations** are allowed on it, while hiding **how** it is stored and implemented. An **Abstract Data Type (ADT)** is a model of a data type defined by its set of values and its operations, without specifying the implementation. Examples: stack, queue, list, set, dictionary.

Table 2.4 Abstract view versus implementation view

Feature	Abstract view	Implementation view
Focus	What the data does	How the data is stored and managed
Concern	Operations and behaviour	Data structures and algorithms
Example	Stack: push, pop, peek	Array-based stack, linked-list stack
User role	Uses the operations	Develops and maintains the details

S Stack

`push(item)`, `pop()`, `peek()`, `isEmpty()` — last in, first out.

Q Queue

`enqueue(item)`, `dequeue()`, `front()`, `isEmpty()` — first in, first out.

L List

`insert(item, pos)`, `remove(pos)`, `search(item)`, `get(pos)`, `size()`.

In C++ a **class** is the tool for data abstraction: users call the public operations; the array inside is private and could be replaced by a linked list without changing user code.

C++ stack_adt.cpp

```cpp
#include <iostream>
using namespace std;
class Stack {                    // the ADT: operations are public
    int data[10]; int top = -1;  // representation is hidden
public:
    void push(int x) { data[++top] = x; }
    int  pop()       { return data[top--]; }
    bool isEmpty()   { return top == -1; }
};
int main() {
    Stack s;
    s.push(10); s.push(20); s.push(30);
    while (!s.isEmpty()) cout << s.pop() << " ";
    cout << endl;
    return 0;
}
```

▶ Output

```
30 20 10
```

Screen 2.8 The user of `Stack` sees only push, pop and isEmpty; values come out in last-in-first-out order.

Benefits

Information hiding, **security** (data protected from misuse), **easier testing**, **flexibility** (implementation can change without affecting users), modularity, reuse.

Real-life view

An ATM shows "Check Balance" and "Send Money". You never see how data is stored, validated or processed — you use the operations only.

★ **Data abstraction hides representation details and exposes only meaningful data behaviour.**

2.8 Procedural Abstraction

DEFINITION 2.6 — PROCEDURAL ABSTRACTION

Procedural abstraction means using a **named procedure or function** to represent a sequence of steps. Instead of writing the same detailed steps repeatedly, we call it by name and hide the internal steps.

Main components

1. **Name** — used to call it.
2. **Parameters** — input values.
3. **Local variables** — temporary data inside.
4. **Body** — the statements.
5. **Return value** — sent back (functions).

Procedure vs function

- **Procedure**: performs a task, no return value — `displayMenu()`.
- **Function**: computes and returns a value — `calculateSum(a, b)`.
- In C++ both are functions; a "procedure" is a function with return type `void`.

C++ procedures.cpp

```cpp
#include <iostream>
using namespace std;
int calculateSum(int a, int b) { return a + b; }      // function
void displayMenu() {                                   // procedure
    cout << "1. Add\n2. Subtract\n3. Exit\n";
}
double area(double radius) { return 3.14 * radius * radius; }
int main() {
    displayMenu();
    cout << calculateSum(5, 7) << endl;
    cout << area(4) << endl;
    return 0;
}
```

▶ Output

```
1. Add
2. Subtract
3. Exit
12
50.24
```

Screen 2.9 One `void` procedure and two value-returning functions; the caller uses each by name only.

EXAM POINT

Advantages: **modularity, maintainability, easier debugging, teamwork, reusability**. Functions return values; procedures do not. Real life: a washing machine's "Wool" mode runs a predefined sequence of steps.

★ **Procedural abstraction packages a sequence of actions into reusable named units.**

Chapter 2 summary

- Syntax = form (compiler checks); semantics = meaning (wrong → wrong output).
- Token = category, lexeme = actual text, pattern = rule (regular expression).
- Identifiers: letter/underscore first, no spaces, not a keyword, case-sensitive.
- Variable = changeable named location; constant (`const`) = fixed.
- Expression produces a value; statement performs an action.
- Control structures: sequence, selection, iteration (+ jump statements).
- Data abstraction / ADT: what, not how (stack, queue, list).
- Procedural abstraction: named functions; procedure = `void` function.

Next: Chapter 3 — programming paradigms: imperative, object-oriented, functional, logic, event-driven and concurrent.

CHAPTER · UNIT VI · LANGUAGE DESIGN

3 Programming Paradigms

OPENING SITUATION

The task "find the total of all marks" can be written as a loop that keeps adding (imperative), as a `Student` object that knows its marks (object-oriented), as one call to a summing function (functional), or as a rule that a solver answers (logic). Each is a different **paradigm**: a different way of thinking about programs. C++ is a **multi-paradigm** language, so every paradigm below can be shown in C++.

IN THIS CHAPTER

- 3.1 Imperative Programming Paradigm
- 3.2 Object-Oriented Programming Paradigm
- 3.3 Functional Programming Paradigm
- 3.4 Logic Programming Paradigm
- 3.5 Event-Driven Programming Paradigm
- 3.6 Concurrent Programming Paradigm
- 3.7 Comparative Study of Paradigms

YOU NEED

- Chapter 2: variables, statements, control structures.

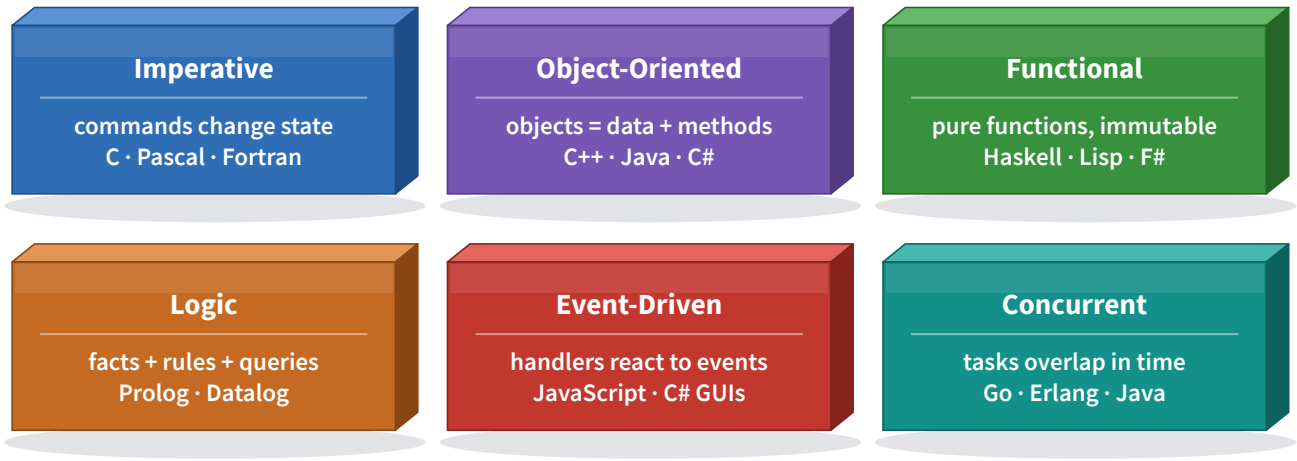


Figure 3.1 The six programming paradigms of this chapter. Imperative and object-oriented are *how*-oriented; functional and logic are *declarative* (*what*-oriented).

3.1 Imperative Programming Paradigm

DEFINITION 3.1 — IMPERATIVE PARADIGM

The **imperative paradigm** describes computation as a **sequence of commands that change the program state**. The program tells the computer exactly *how* to do a task, step by step.

1 Core idea and features

- Variables store data that can change (**mutable data**).
- **Assignment statements** perform operations.
- **Sequence, selection, iteration** with explicit control flow.
- Structure: **Input → Processing → Output**.

2 Strengths and limits

- ✓ Simple mental model; efficient; close to machine behaviour (von Neumann).
- ✗ Large programs become hard to maintain.
- ✗ Too much mutable state causes bugs.
- Languages: **C, Pascal, Fortran, BASIC**, scripting languages.

C++ imperative.cpp

```cpp
#include <iostream>
using namespace std;
int main() {
    int marks[] = {78, 91, 65, 88};
    int total = 0;                    // state
    for (int i = 0; i < 4; i++)       // command after command
        total = total + marks[i];     // the state changes each time
    cout << "Total = " << total << endl;
    return 0;
}
```

▶ Output

```
Total = 322
```

Screen 3.1 Imperative style: the variable `total` is updated step by step until the answer is reached.

★ **Imperative programming tells the computer exactly how to do a task through ordered commands.**

3.2 Object-Oriented Programming Paradigm

DEFINITION 3.2 — OBJECT-ORIENTED PARADIGM

Object-Oriented Programming (OOP) organises a program around **objects** that combine **data (attributes)** and **behaviour (methods)** into one unit. It models real-world entities, making software modular, reusable and easier to maintain.

C Class

Blueprint for objects. Logical; defining a class creates no object.

O Object

Instance of a class with its own state; memory allocated when it is created.

E Encapsulation

Bundles data with methods and restricts direct access.

I Inheritance

New classes reuse existing ones.

P Polymorphism

One interface, many implementations.

A Abstraction

Shows essential features, hides details.

C++ oop_car.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;
class Car {                          // class = blueprint
public:
    string model; int speed = 0;     // attributes
    void accelerate(int d) { speed += d; }      // methods
    void show() { cout << model << " at " << speed << " km/h\n"; }
};
int main() {
    Car myCar, yourCar;              // objects = instances
    myCar.model = "Nexon";  yourCar.model = "Swift";
    myCar.accelerate(60);            // message passing
    yourCar.accelerate(40); yourCar.accelerate(10);
    myCar.show(); yourCar.show();
    return 0;
}
```

▶ Output

```
Nexon at 60 km/h
Swift at 50 km/h
```

Screen 3.2 One class, two objects, each with its own state; objects act when they receive messages (method calls).

EXAM POINT

The **four pillars of OOP**: Encapsulation, Inheritance, Polymorphism, Abstraction. OOP languages: **Java, C++, C#, Python, Ruby**. Steps: define a class → create objects → each object has its own state → objects communicate by messages.

★ **OOP builds software by modelling real-world entities as interacting objects.**

3.3 Functional Programming Paradigm

DEFINITION 3.3 — FUNCTIONAL PARADIGM

Functional programming treats computation as the **evaluation of functions**. It emphasises **immutability** (data does not change; new values are created) and avoids changing state.

Table 3.1 Key functional concepts

Concept	Meaning	Example
Pure function	Same input always gives the same output; no side effects	`f(x) = x + 1`
Higher-order function	Takes functions as arguments or returns a function	`map(f, list)`, `filter(p, list)`
Lambda	Anonymous small function	C++: `[](int x){ return x*x; }`
Recursion	A function calls itself (instead of loops)	`fact(n) = n * fact(n-1)`
Map / filter / reduce	Transform, select, combine collections	C++: `transform`, `copy_if`, `accumulate`
First-class functions	Functions are values: stored, passed, returned	`auto sq = [](int x){…};`

C++ functional.cpp

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
using namespace std;
int fact(int n) { return n <= 1 ? 1 : n * fact(n - 1); }   // recursion
int main() {
    const vector<int> v = {1, 2, 3, 4, 5};                  // never modified
    vector<int> sq(v.size());
    transform(v.begin(), v.end(), sq.begin(), [](int x) { return x * x; }); // map
    int total = accumulate(sq.begin(), sq.end(), 0);        // reduce
    cout << "squares sum = " << total << ", 5! = " << fact(5) << endl;
    return 0;
}
```

▶ Output

```
squares sum = 55, 5! = 120
```

Screen 3.3 Functional style in C++: a lambda passed to the higher-order function `transform`, a reduction with `accumulate`, and recursion.

Benefits

Easier reasoning (pure functions), fewer side effects and bugs, safe for parallelism, concise code, ideal for data transformation.

Limitations

Recursion overhead; learning curve; not ideal for problems with heavy state changes.

Languages and uses

Haskell, Lisp, Scheme, F#, Erlang, Scala; Python and C++ support functional features. Used in ETL (extract–transform–load), log processing, finance.

EXAM POINT

Functional programming is **declarative** (not imperative), treats functions as **first-class citizens**, emphasises **immutability** and avoids changing state.

★ **Functional programming focuses on what to compute through functions rather than changing state step by step.**

3.4 Logic Programming Paradigm

DEFINITION 3.4 — LOGIC PARADIGM

Logic programming expresses problems as **facts, rules and queries**. The system uses **inference** to derive answers by applying rules to known facts.

1 Core elements

- **Facts** — basic truths: `parent(alice, bob).`
- **Rules** — implications that derive new facts.
- **Queries** — questions posed to the system.
- **Unification** — matching terms so a rule applies.
- **Backtracking** — trying alternatives to find all solutions.

2 Features and uses

- **Declarative**: describe what is true, not how to compute.
- Automatic search for solutions by inference.
- Languages: **Prolog**, Datalog, Mercury.
- Uses: AI, expert systems, theorem proving, natural language processing, database querying.



family.pl — PROLOG

```prolog
parent(alice, bob).
parent(bob, carol).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
?- grandparent(alice, Y).          % answer: Y = carol
```

C++ is not a logic language, but the same inference can be imitated by searching the facts — which shows how much work Prolog does automatically:

C++ logic_sim.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;
struct Fact { string parent, child; };
int main() {
    vector<Fact> kb = {{"alice", "bob"}, {"bob", "carol"}, {"bob",
"dan"}};
    // rule: grandparent(X, Y) if parent(X, Z) and parent(Z, Y)
    for (auto f1 : kb)
        for (auto f2 : kb)
            if (f1.parent == "alice" && f1.child == f2.parent)
                cout << "Y = " << f2.child << endl;
    return 0;
}
```

▶ Output

```
Y = carol
Y = dan
```

Screen 3.4 Searching facts for the query `grandparent(alice, Y)` : both solutions are found, as Prolog's backtracking would find them.

★ **Logic programming solves problems by applying facts and rules to answer queries.**

3.5 Event-Driven Programming Paradigm

DEFINITION 3.5 — EVENT-DRIVEN PARADIGM

In **event-driven programming** the flow of the program is determined by **events** — user actions, device inputs or system messages — and code is executed **in response** to those events.

Table 3.2 Main elements of event-driven programs

Element	Meaning	Example
Event	Something that happens in the system	Button clicked, key pressed
Event source	Object that generates the event	Button, timer, sensor
Event handler	Code that runs in response	`onClick()`, `updateData()`
Listener / callback	Component registered to be notified	`addEventListener()`, `subscribe()`
Event queue / loop	Stores pending events; the loop waits and dispatches them	queue of pending events

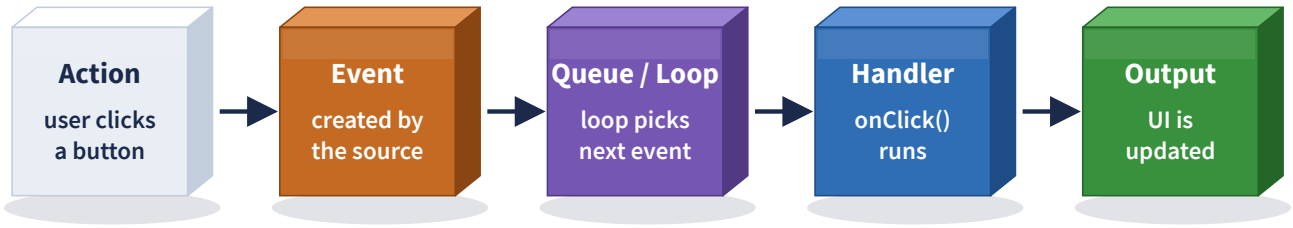


Figure 3.2 How an event-driven program works: the event loop takes each event from the queue and runs its handler, then waits for the next one.

C++ events.cpp

```cpp
#include <iostream>
#include <functional>
#include <map>
#include <queue>
#include <string>
using namespace std;
int main() {
    map<string, function<void()>> handlers;            // registered listeners
    handlers["click"] = [] { cout << "Button clicked: saving form\n"; };
    handlers["key"]   = [] { cout << "Key pressed: updating text\n"; };
    queue<string> events;                              // event queue
    events.push("key"); events.push("click"); events.push("key");
    while (!events.empty()) {                           // event loop
        handlers[events.front()]();                     // dispatch to handler
        events.pop();
    }
    return 0;
}
```

Output

```
Key pressed: updating text
Button clicked: saving form
Key pressed: updating text
```

Screen 3.5 A miniature event loop: handlers are registered as callbacks, and each queued event is dispatched to its handler in turn.

REAL-LIFE VIEW

A **doorbell**: someone presses it (event), the bell unit (source) detects it and triggers the response (handler) — the bell rings. Typical events: button click, key press, mouse movement, timer, network or sensor data. Used in **GUI apps, web pages, games and IoT**; it keeps applications responsive.

★ Event-driven programming reacts to events and runs the appropriate handler to update the program or interface.

3.6 Concurrent Programming Paradigm

DEFINITION 3.6 — CONCURRENT PARADIGM

In **concurrent programming** multiple tasks make progress during **overlapping time periods**. They may run truly in parallel on several processors, or appear to through **time-slicing** (context switching) on one processor.

1 Key concepts

- **Thread** — lightweight unit of execution inside a process.
- **Process** — independent program in execution with its own memory space.
- **Shared memory / message passing** — ways to communicate.
- **Mutex / semaphore** — control access to shared resources (critical section).
- **Scheduling** — the system decides which task runs.

! Common challenges

- **Race condition** — tasks access shared data at once → inconsistent results.
- **Deadlock** — tasks wait forever for each other's resources.
- **Starvation** — a task never gets to run.
- **Nondeterminism** — result depends on timing.

C++ threads.cpp

```cpp
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
long total = 0;
mutex m;                                  // protects total
void addRange(int from, int to) {
    long part = 0;
    for (int i = from; i <= to; i++) part += i;
    lock_guard<mutex> lock(m);            // critical section
    total += part;
}
int main() {
    thread t1(addRange, 1, 500), t2(addRange, 501, 1000);  // run together
    t1.join(); t2.join();                 // wait for both
    cout << "Sum 1..1000 = " << total << endl;
    return 0;
}
```

▶ Output

```
Sum 1..1000 = 500500
```

Screen 3.6 Two threads add halves of the range at the same time; the `mutex` makes the update of the shared variable safe, so the result is always 500500.

REAL-LIFE VIEW

A **kitchen**: chef, cook and cleaner work at the same time, share utensils and the stove, and coordinate to avoid conflicts. In an office, typing a report, printing and attending a call happen together. Benefits: responsiveness, better use of CPU and I/O, scalability on multi-core systems.

★ Concurrency lets multiple activities progress together while coordinating access to resources.

3.7 Comparative Study of Paradigms

A **programming paradigm** is a way of organising and thinking about programs: a set of concepts and rules that guide how we design, write and structure code.

Table 3.3 Paradigms compared

Paradigm	Main focus	Program style	Languages	Best for
Imperative	How to do (step by step); state change	Sequence of commands	C, Pascal, Python	Systems, low-level tasks
Object-oriented	Modelling real-world entities	Objects and classes interacting	Java, C++, C#, Python	Large business applications, simulations
Functional	Computation as evaluation; no state change	Functions and expressions	Haskell, Scala, F#, Lisp	Data transformation, maths, analytics
Logic	Facts, rules, logical inference	Rules and queries	Prolog, Datalog	Expert systems, AI, knowledge bases
Event-driven	Responding to events	Event handlers and callbacks	JavaScript, C#, Python (Tkinter)	GUIs, games, real-time apps
Concurrent	Simultaneous tasks and communication	Processes / threads running together	Go, Erlang, Java	Servers, web apps, parallel computing

MEMORY TRICK — ONE WORD PER PARADIGM

Imperative = **commands** · OOP = **objects** · Functional = **functions** · Logic = **rules** · Event-driven = **events** · Concurrent = **simultaneous tasks**.

COMMON MISTAKE 3.1 — "C++ IS A PURELY OBJECT-ORIENTED LANGUAGE"

C++ is **multi-paradigm**: it supports procedural (imperative), object-oriented, generic (templates) and functional (lambdas) styles. A C++ program need not contain any class, and `main()` is always outside every class. Languages such as Smalltalk are called *pure* object-oriented.

★ Different paradigms offer different ways to solve problems; the best choice depends on the task.

Chapter 3 summary

- Imperative: commands change state (C, Pascal, Fortran).
- OOP: objects = data + methods; four pillars (Java, C++, C#).
- Functional: pure functions, immutability, recursion, higher-order functions (Haskell, Lisp).
- Logic: facts, rules, queries, unification, backtracking (Prolog).
- Event-driven: event → queue/loop → handler (GUIs, web).
- Concurrent: threads/processes; race condition, deadlock, starvation; mutex.
- Functional and logic are declarative; imperative and OOP are procedural in style.
- C++ is multi-paradigm.

Next: Chapter 4 — computational models behind languages: von Neumann, declarative, object-based, dataflow, concurrent and parallel.

CHAPTER · UNIT VI · LANGUAGE DESIGN

4 Programming Models

OPENING SITUATION

Behind every paradigm sits a **computational model** — a picture of how computation happens. A C++ loop follows the von Neumann machine on the desk; an SQL query follows the declarative model; a spreadsheet recalculating cells follows the dataflow model. Knowing the model tells us which language fits a problem.

IN THIS CHAPTER

- 4.1 Von Neumann Model
- 4.2 Declarative Model
- 4.3 Object-Based Model
- 4.4 Dataflow Model
- 4.5 Concurrent and Parallel Models
- 4.6 Model-Based Language Selection

YOU NEED

- Chapter 3: programming paradigms.

4.1 Von Neumann Model

DEFINITION 4.1 — VON NEUMANN (STORED-PROGRAM) MODEL

The **von Neumann model** is a **stored-program architecture** in which **both instructions and data are stored in the same memory**. The CPU fetches instructions from memory, executes them, and stores results back in memory.

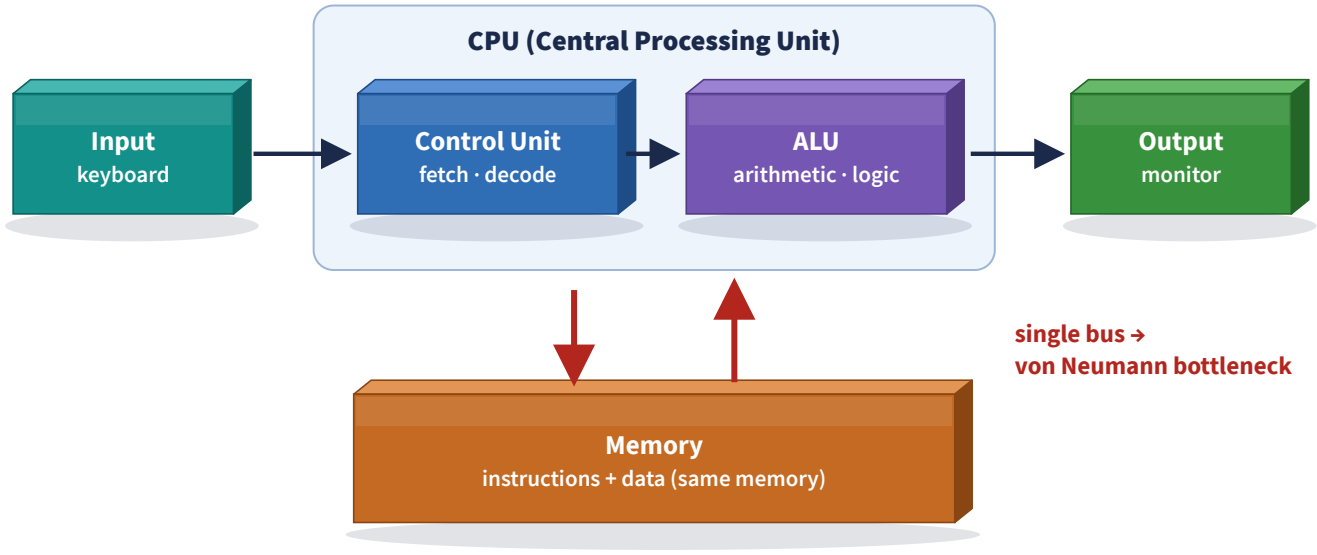


Figure 4.1 Von Neumann architecture: one memory holds both the program and the data, and one path (bus) connects it to the CPU — the source of the *von Neumann bottleneck*.

Table 4.1 Components of the von Neumann machine

Component	Function	Example role
Memory	Stores instructions and data	RAM, ROM
Control Unit (CU)	Fetches and decodes instructions; issues control signals	Instruction decoder, sequencer
ALU (Arithmetic Logic Unit)	Arithmetic and logical operations	Adder, comparator
Input unit	Takes data and instructions from outside	Keyboard, sensor
Output unit	Delivers results to the user or devices	Monitor, printer



Key features

- Single memory for instructions and data.
- Instructions executed **sequentially**, one at a time.
- Centralised control by the CU; simple, general-purpose.

Limitations

- **Von Neumann bottleneck**: one memory and one bus — instruction fetch and data access compete.
- Limits speed and throughput; not ideal for highly parallel or real-time work.

EXAM POINT

Stored-program concept: instructions and data share the same memory. Instruction cycle: **Fetch → Decode → Execute → Store**. Imperative languages (C, C++) mirror this model: variables = memory cells, assignment = store. The **Harvard architecture** differs: it has **separate** memories (and buses) for instructions and data.

★ **The von Neumann model stores programs and data in memory and executes instructions step by step using the CU and ALU.**

4.2 Declarative Model

DEFINITION 4.2 — DECLARATIVE MODEL

The **declarative model** specifies **what** result is wanted, not the step-by-step procedure (**how**). The programmer describes relationships, constraints or queries; the system decides the most efficient way to produce the result.

Table 4.2 Main forms of declarative programming

Form	Explanation	Example
Logic-based	States facts and rules; system infers new information	Prolog: `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`
Functional	Expressions and functions; no state change	Haskell: `sum (map (^2) (filter even [1..n]))`
Query-based	Specify what data is needed, not how to retrieve it	SQL: `SELECT name FROM Students WHERE age > 18 AND city = 'Chennai';`

C++ is imperative, but its standard algorithms let us write in a declarative spirit. Both parts below count the passing marks:

C++ declarative.cpp

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    int m[] = {35, 72, 40, 18, 90};
    int c = 0;                                  // HOW: loop + counter
    for (int i = 0; i < 5; i++) if (m[i] >= 40) c++;
    cout << "imperative: " << c << endl;
    cout << "declarative: "                     // WHAT: count if pass
         << count_if(m, m + 5, [](int x) { return x >= 40; }) << endl;
    return 0;
}
```

Output

```
imperative: 3
declarative: 3
```

Screen 4.1 The imperative version spells out the loop; `count_if` only states the condition and leaves the iteration to the library.

REAL-LIFE VIEW

In a cab app you state **what** you want (destination); the app finds the route (**how**). Benefits: readability, high abstraction, easier reasoning, concise code, and optimisation by the system (an SQL engine chooses its own query plan).

★ **Declarative programming states the goal; the system works out the process.**

4.3 Object-Based Model

DEFINITION 4.3 — OBJECT-BASED MODEL

The **object-based model** organises programs around **objects**, each encapsulating its own **state** (data) and **behaviour** (operations), interacting with other objects through **messages**.

1 Core elements

- **Object** — self-contained unit with state and behaviour.
- **State** — attributes; **Behaviour** — methods.
- **Identity** — distinguishes one object from another.
- **Message passing** — method calls between objects.

2 Object-based vs object-oriented

- Object-based: objects + encapsulation; **may not support inheritance**; looser structure.
- Object-oriented: adds **classes, inheritance, polymorphism** and hierarchies.
- Formula: **object-oriented = object-based + inheritance** (+ polymorphism).

C++ object_based.cpp

```cpp
#include <iostream>
using namespace std;
class BankAccount {                    // no inheritance used
    double balance = 0;                // state (hidden)
public:
    void deposit(double a)  { balance += a; }      // behaviour
    bool withdraw(double a) { if (a > balance) return false; balance -= a; return true; }
    double check() const    { return balance; }
};
int main() {
    BankAccount asha, ravi;            // two objects, two identities
    asha.deposit(5000); ravi.deposit(800);
    cout << boolalpha << ravi.withdraw(1000) << " " << asha.withdraw(1000) << endl;
    cout << asha.check() << " " << ravi.check() << endl;
    return 0;
}
```

▶ Output

```
false true
4000 800
```

Screen 4.2 Object-based design: each account keeps its own balance, and objects respond to messages; Ravi's withdrawal is refused (`false`).

EXAM POINT

Object-based languages: **Ada 83**, Visual Basic 6, early JavaScript — they have objects but no (class) inheritance. Examples of objects: Car (speed, fuel; start(), brake()), Bank account (balance; deposit(), withdraw()), Student record (name, marks; calculateGrade()), UI button (label; click()). Benefits: modularity, abstraction, reuse, maintainability.

★ **The object-based model builds software from independent objects that encapsulate data and operations and interact by messages.**

4.4 Dataflow Model

DEFINITION 4.4 — DATAFLOW MODEL

The **dataflow model** represents computation as **data flowing through a network of interconnected operations (nodes)**. An operation is triggered ("**fires**") as soon as all its required input data (**tokens**) are available; its results flow on as new tokens.

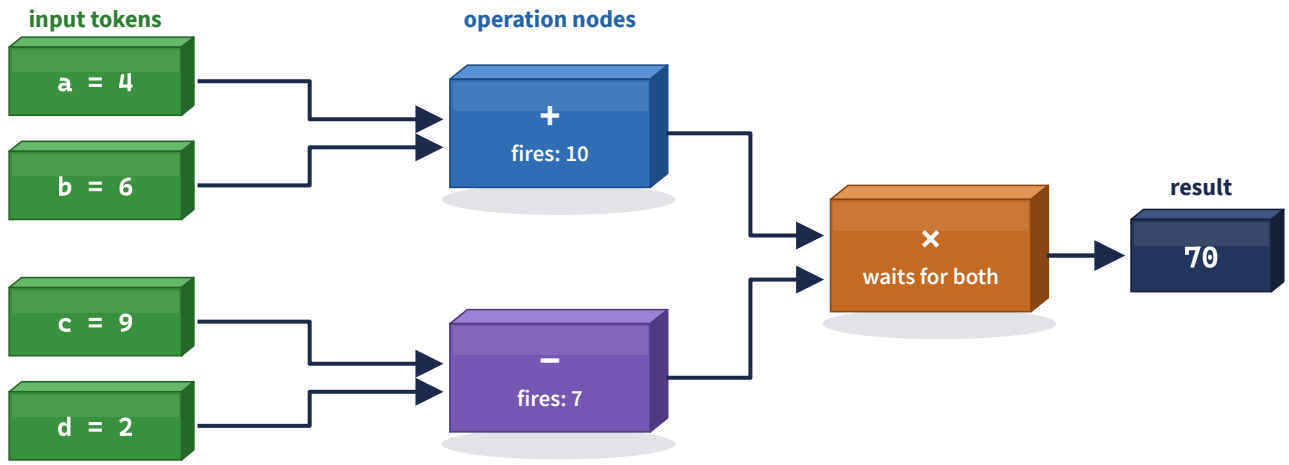


Figure 4.2 Dataflow graph for `(a + b) * (c - d)`. The `+` and `-` nodes have their inputs at once, so they can fire in parallel; `×` fires only when both results arrive.

Table 4.3 Elements of a dataflow network

Element	Description
Node / operator	Performs a computation (add, filter, transform)
Data token	A unit of data flowing through the network
Arc / edge	Directed connection carrying tokens between nodes
Dependency	A node waits for all its required input tokens
Output	Result tokens sent to downstream nodes

C++ dataflow.cpp

```cpp
#include <iostream>
#include <future>
using namespace std;
int main() {
    int a = 4, b = 6, c = 9, d = 2;
    // the two nodes have their tokens already: they may fire in parallel
    future<int> sum  = async(launch::async, [=] { return a + b; });
    future<int> diff = async(launch::async, [=] { return c - d; });
    // the multiply node fires only when both input tokens arrive
    cout << "(a+b)*(c-d) = " << sum.get() * diff.get() << endl;
    return 0;
}
```

Output

```
(a+b)*(c-d) = 70
```

Screen 4.3 Dataflow imitated with futures: execution order is driven by data availability, not by a fixed sequence of statements.

REAL-LIFE VIEW

Water flows from the tap (source tokens) through pipes (arcs) and valves (operations) into a bucket (output). A spreadsheet works the same way: change one cell and every dependent cell recalculates. Uses: signal processing, streaming data, pipelines, real-time systems; it exposes parallelism naturally (no fixed global sequence).

★ **In the dataflow model, data availability drives execution and results move through a network of operations.**

4.5 Concurrent and Parallel Models

Concurrent model

- Many tasks make progress during **overlapping time periods**; they may run, wait or switch.
- Can run on a **single core** by time-slicing.
- Goal: manage many tasks. Example: a web server handling many requests.

Parallel model

- Many tasks execute **at the same instant** on multiple processors or cores.
- **Requires** multi-core or multi-processor hardware.
- Goal: speed-up. Example: matrix multiplication on several cores.

MEMORY TRICK

Concurrency is about *dealing* with many things at once; parallelism is about *doing* many things at once. Every parallel system is concurrent, but not every concurrent system is parallel.

1 Core idea

- **Tasks** — work units; **processors/cores** execute them.
- **Coordination** — assigning and communicating.
- **Synchronisation** — correct access to shared data (sync and merge points).
- **Speed-up** — faster results using many processors.

+ Uses and benefits

- Web servers, scientific simulation, matrix and numerical computing, embedded control, big-data analytics.
- Throughput, responsiveness, performance, scalability.

C++ hardware.cpp

```cpp
#include <iostream>
#include <thread>
using namespace std;
int main() {
    unsigned cores = thread::hardware_concurrency();
    cout << (cores > 1 ? "Parallel execution possible: more than one core\n"
                       : "Single core: concurrency by time-slicing only\n");
    return 0;
}
```

▶ Output

```
Parallel execution possible: more than one core
```

Screen 4.4 `thread::hardware_concurrency()` reports how many threads the hardware can truly run in parallel.

REAL-LIFE VIEW

Concurrency: many cooks share one stove, taking turns — many dishes progress, but not all are cooking at the same instant. **Parallelism**: each cook has a separate station, so dishes are cooked simultaneously and finish faster.

★ **Concurrency manages many tasks; parallelism runs tasks simultaneously on many processors for faster results.**

4.6 Model-Based Language Selection

DEFINITION 4.5 — MODEL-BASED LANGUAGE SELECTION

Model-based language selection chooses a programming language by matching the problem's **computational model** with the language that best supports it. Goal: the right language for the right model → correct, efficient, maintainable solutions.

Table 4.4 Computational model → language

Model	Description	Typical languages	Use cases
Von Neumann / imperative	Sequential execution with state and control flow	C, C++	System software, OS components, performance-critical code
Declarative	What to achieve, not how	SQL, Prolog, Haskell	Database queries, logic reasoning, constraint solving
Object-based	Objects and their interactions	Java, C#, Python	Enterprise apps, simulations, large systems
Dataflow	Data moves through a network of operations	LabVIEW, streaming tools	Signal processing, pipelines, real-time data
Concurrent / parallel	Multiple tasks execute simultaneously	Go, Erlang, CUDA, OpenMP	Web services, distributed systems, GPU computing

? Selection factors

Problem type, **performance needs** (speed, memory), **concurrency**, **maintainability**, **ecosystem** (libraries, tools, community), **team skill**.

✓ Decision process

1. Identify the problem and requirements.
2. Choose the model that fits best.
3. Evaluate languages supporting it.
4. Prototype with the shortlist.
5. Finalise on trade-offs.

Typical scenarios: web applications → concurrent (Go, Java, Python); scientific computing → dataflow/parallel (Python, C++, CUDA); AI and reasoning → declarative (Prolog, Haskell, Python); embedded systems → imperative (C, C++); GUI applications → object-based (Java, C#, Python).

REAL-LIFE VIEW

Choosing a language is like choosing a vehicle: a racing bike for speed, a dump truck for heavy loads, a tractor for rough fields. A mismatch between model and language leads to complex, inefficient solutions.

★ **The best language choice comes from matching the computational model to the task's requirements.**

Chapter 4 summary

- Von Neumann: stored program — instructions and data in one memory.
- Instruction cycle: fetch → decode → execute → store; von Neumann bottleneck.
- Harvard architecture: separate instruction and data memories.
- Declarative: what, not how (SQL, Prolog, Haskell).
- Object-based = objects + encapsulation; object-oriented adds inheritance.
- Dataflow: nodes fire when input tokens arrive; natural parallelism.
- Concurrency = dealing with many things; parallelism = doing many at once.
- Select a language by matching the problem's computational model.

Next: Chapter 5 — programming environments: editors, IDEs, compilers, debuggers, linkers, loaders and run-time support.

CHAPTER · UNIT VI · LANGUAGE DESIGN

5 Programming Environments

OPENING SITUATION

When a student presses **Build and Run** in Code::Blocks or Visual Studio, at least five programs work in turn: an editor saved the file, a compiler translated it, a linker joined it with the C++ library, a loader placed it in memory, and the run-time system supports it until it ends. Together these tools form the **programming environment**.

IN THIS CHAPTER

- 5.1 Programming Environment
- 5.2 Editors and Integrated Development Environments
- 5.3 Compilers and Interpreters
- 5.4 Debuggers and Profilers
- 5.5 Linkers and Loaders
- 5.6 Runtime Environment
- 5.7 Development Tools and Libraries

YOU NEED

- Chapter 1: translation of source code.

5.1 Programming Environment

DEFINITION 5.1 — PROGRAMMING ENVIRONMENT

A **programming environment** is the complete set of software tools and services that help developers **write, edit, translate, test, debug and run** programs efficiently. It provides an integrated workspace that simplifies development and improves productivity.

E Editor
Write and edit source code.

T Translator
Compiler/interpreter: converts code to machine language.

D Debugger
Finds and fixes errors.

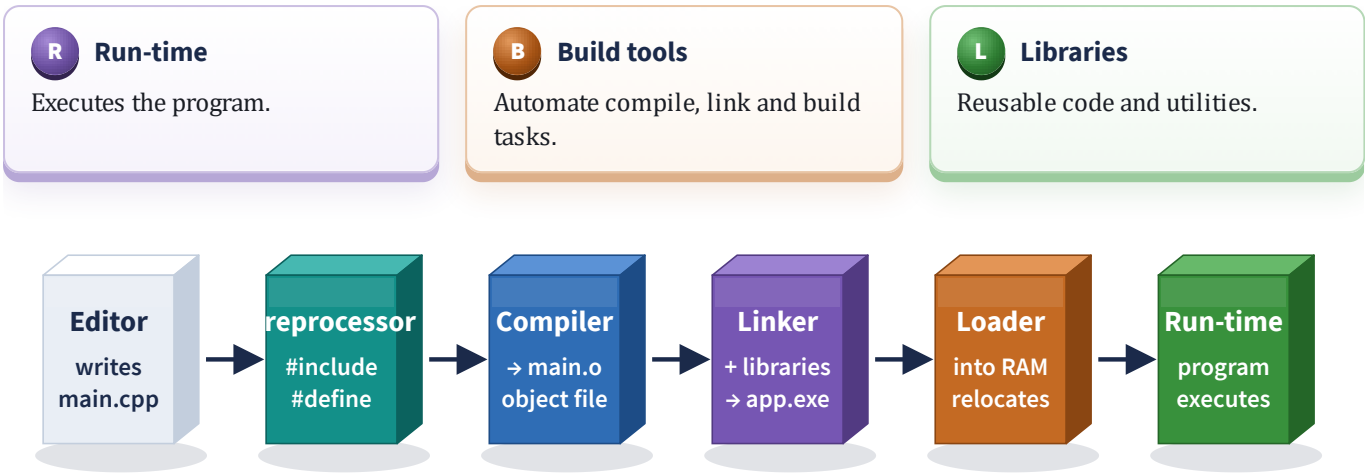


Figure 5.1 The C++ tool chain: each tool of the environment hands its output to the next one, from source file to running program.

Table 5.1 Language, environment and platform

Aspect	Programming language	Programming environment	Execution platform
Definition	Formal rules and syntax for writing programs	Tools and services for program development	System or hardware where the program runs
Purpose	Defines how to write the program	Helps create, test and manage it	Executes it and provides resources
Examples	C, C++, Python, Java	IDE (VS Code, IntelliJ), compiler, debugger, build tools	Windows, Linux, macOS, Android, embedded systems

How it works: write code (editor) → translate/build (compiler) → test/debug (debugger) → run (run-time) → maintain/update (version control). Key features: integrated tools, productivity aids (code completion, templates, refactoring), code management, testing support, automation, collaboration. Benefits: easier development, fewer errors, faster delivery, maintainability, better learning.

REAL-LIFE VIEW

A programming environment is a complete **workshop**: the editor is the workbench, the compiler is the machine that shapes the product, the debugger is the quality checker, and the run-time is where the product is used.

★ **A programming environment is an integrated set of tools to write, test, debug and run programs from start to finish.**

5.2 Editors and Integrated Development Environments

DEFINITION 5.2 — EDITOR AND IDE

An **editor** is a simple text-based tool to write and edit source code. An **IDE (Integrated Development Environment)** is a complete workspace combining an editor with tools for coding, building, testing and debugging.

Table 5.2 Editor versus IDE

Aspect	Editor	IDE
Scope	Writing and editing code	Complete workspace with many tools
Tools	Basic editing, syntax highlighting	Editor, compiler/interpreter, debugger, terminal, project explorer, build tools
Ease of use	Lightweight and simple	Feature-rich; may have a learning curve
Examples	Notepad++, Sublime Text, VS Code (editor mode), Vim	Visual Studio, Code::Blocks, Dev-C++, Eclipse, PyCharm, IntelliJ IDEA
Best use	Quick edits, small scripts, learning	Large projects, teamwork, professional work

IDE features: syntax highlighting, auto-completion, project explorer, integrated terminal, extensions/plug-ins, build/run support. Workflow: open project → write code → get suggestions → run/build → test and refine. Popular C++ IDEs: **Visual Studio, Code::Blocks, Dev-C++, CLion, Eclipse CDT**; the classic DOS-era one is **Turbo C++**.

REAL-LIFE VIEW

An editor is a notebook where you write your thoughts; an IDE is a smart workstation that helps you write, organise, test and run them.

★ **Editors help you write code; IDEs give a complete environment to write, build, test and debug.**

5.3 Compilers and Interpreters

DEFINITION 5.3 — COMPILER AND INTERPRETER

A **compiler** translates the **entire** source program into machine code **before** execution, producing an executable. An **interpreter** translates and executes the source **statement by statement**, without producing a separate machine-code file.

Table 5.3 Compiler versus interpreter

Aspect	Compiler	Interpreter
Translation	Whole program at once	One statement at a time
Speed	Faster execution after compilation	Slower (translation during the run)
Error reporting	Reports all errors after compiling the program	Stops at the first error encountered
Output	Executable file (machine code), e.g. `.exe`	No separate output file
Memory at run time	Generally less	Generally more (interpreter stays in memory)
Portability	Executable is platform dependent	Source is portable wherever the interpreter exists
Examples	C, C++, Go, Rust (GCC, Clang, MSVC)	Python, JavaScript, PHP

C Compile when…

maximum performance is needed, the program runs many times, strong error checking is required.

I Interpret when…

quick testing and debugging are needed, portability matters, programs are small or dynamic.

COMMON MISTAKE 5.1 — "JAVA AND PYTHON ARE PURELY INTERPRETED"

Java is compiled by `javac` to **bytecode**, which the JVM interprets *and* JIT-compiles. **CPython** also compiles source to bytecode first and then interprets the bytecode. Such languages are best called **hybrid** (compile + interpret). JavaScript engines such as V8 (Chrome) and SpiderMonkey (Firefox) likewise JIT-compile.

REAL-LIFE VIEW

A **compiler** is a translator who translates the whole book first; an **interpreter** translates sentence by sentence while the speaker talks.

★ **A compiler translates the whole program once and runs it fast; an interpreter translates and runs it statement by statement.**

5.4 Debuggers and Profilers

Debugger — correctness

- Finds and fixes errors (bugs) by inspecting execution.
- **Breakpoints**, **step into / step over**, **variable watch**, **stack trace**.
- Used during development and testing.
- GDB, LLDB, Visual Studio Debugger, Chrome DevTools.

Profiler — performance

- Measures and analyses performance to find **bottlenecks**.
- Execution time, CPU usage, memory usage, function-call statistics, memory leaks.
- Used during optimisation and tuning.
- gprof, Valgrind, Visual Studio Profiler, cProfile (Python).

Before using a debugger, programmers often add an **assertion**: a check that stops the program at the exact point where an assumption fails.

C++ assert.cpp

```cpp
#include <iostream>
#include <cassert>
using namespace std;
double average(int total, int count) {
    assert(count > 0);                // debugging check
    return (double)total / count;
}
int main() {
    cout << average(250, 5) << endl;
    cout << average(250, 0) << endl;   // bug: count is zero
    return 0;
}
```

▶ Output

```
50
Assertion failed: count > 0, file assert.cpp, line 5
```

Screen 5.1 The first call works; the second stops with a message naming the file, line and failed condition — exactly where a debugger breakpoint would be set.

How it works: run → detect bug or slowdown → inspect behaviour → fix/optimise → retest and verify.

REAL-LIFE VIEW

A debugger is a **mechanic** who finds faults and fixes them; a profiler is a **performance meter** that shows where time and resources are spent most.

★ **Debuggers fix bugs by inspecting execution; profilers improve performance by analysing resource use.**

5.5 Linkers and Loaders

DEFINITION 5.4 — LINKER AND LOADER

A **linker** combines object files and libraries, **resolves external references** (symbols) and produces an **executable file**. A **loader** places the executable into main memory, performs **relocation** and final preparation, and transfers control to the program's entry point.

Table 5.4 Linker versus loader

Aspect	Linker	Loader
Stage	Before execution (link time)	At execution (load time)
Function	Combines, resolves symbols, creates the executable	Places the executable in memory and prepares it to run
Input	Object files (`.o` , `.obj`), libraries (`.lib` , `.a`)	Executable file, system settings
Output	Executable (`.exe` , `a.out`)	Program in memory, ready to run
Role	Builds the program	Runs (starts) the program

▶ Command Prompt — separate compile and link

```
> g++ -c main.cpp            (compile only: creates main.o)
> g++ -c util.cpp            (compile only: creates util.o)
> g++ main.o util.o -o app   (link: creates app.exe)
> app                        (the loader starts the program)
```

If a function is declared and called but never defined, the **compiler** is satisfied — the error appears only when the **linker** cannot resolve the name:

C++ link_error.cpp

```cpp
#include <iostream>
int square(int x);                 // declaration only
int main() {
    std::cout << square(5);        // compiles, but who defines square?
    return 0;
}
```

✕ Linker — build failed

```
lld-link: error: undefined symbol: square(int)
```

Screen 5.2 A link-time error (“undefined symbol”): the program compiled, but the linker could not resolve the external reference `square` .

S Static vs dynamic linking

Static: library code copied into the executable at link time (bigger file, self-contained). **Dynamic**: libraries (`.dll`, `.so`) linked at load/run time; **shared libraries** save memory and disk.

R Relocation

Adjusting addresses so the program can run wherever it is placed in memory. The file produced by the linker is the **executable image**.

REAL-LIFE VIEW

The linker is an **assembler of machines** who gathers all parts (modules, libraries) and builds the final machine; the loader is the **operator** who places it in the factory (memory), connects everything and starts it.

★ **Linkers build the executable by combining and resolving program parts; loaders place it in memory and start it.**

5.6 Runtime Environment

DEFINITION 5.5 — RUNTIME ENVIRONMENT

A **runtime environment** is the system that supports a program while it executes: the resources, services and infrastructure needed to run it correctly until it finishes.

1 Main components

- **Memory management** (stack, heap, garbage collection where present).
- **Run-time libraries**.
- **Virtual machine** for bytecode/managed code.
- **I/O services**, **exception handling**, **security**.

2 Run time vs build time

- **Run time**: while the program runs — JVM, .NET CLR, Python runtime, browser JavaScript runtime, C++ runtime library.
- **Build time**: before it runs — compiler, linker, assembler, Make, IDEs.

C++ has a small runtime library. Below, the runtime detects an invalid index and handles the error through the exception mechanism:

C++ runtime.cpp

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;
int main() {
    vector<int> v = {10, 20, 30};
    try {
        cout << v.at(1) << endl;     // fine
        cout << v.at(7) << endl;     // checked at run time
    } catch (const out_of_range& e) {
        cout << "Runtime error caught: out_of_range" << endl;
    }
    return 0;
}
```

▶ Output

```
20
Runtime error caught: out_of_range
```

Screen 5.3 Run-time services: bounds checking by `at()` and the exception-handling support of the C++ runtime.

REAL-LIFE VIEW

A runtime environment is the **stage and backstage crew** of a theatre: the actor (program) performs, while the crew provides lights, sound, props and safety and handles problems behind the scenes. Benefits: platform support, memory safety, portability (code runs anywhere the runtime exists), managed execution.

★ **A runtime environment provides the resources and services to execute a program safely until it finishes.**

5.7 Development Tools and Libraries

DEFINITION 5.6 — DEVELOPMENT TOOLS AND LIBRARIES

Development tools are applications that help developers write, build, test, debug and manage code. **Libraries** are collections of pre-written, reusable code for common tasks, used directly by the program.

Table 5.5 Kinds of tools, with examples

Type	Purpose	Examples
Version control	Track and manage code changes	Git
Build tools	Automate compilation and packaging	Make, CMake, Maven, Gradle
Testing tools	Check quality and correctness	Google Test, JUnit, pytest
Package managers	Manage dependencies	vcpkg, Conan, npm, pip
Documentation tools	Create documentation	Doxygen
Libraries / frameworks	Pre-built code for common features	C++ Standard Library (STL), Java Standard Library, Qt, React

Development tools

- Support the **process**: building, testing, managing, deploying.
- Used **indirectly**; set up and configured by the developer.

Libraries

- Support the **code**: provide functionality.
- Used **directly**: `#include` and call their functions/classes.

C++ library.cpp

```cpp
#include <iostream>
#include <algorithm>   // library: ready-made sort
#include <string>
using namespace std;
int main() {
    string names[] = {"Ravi", "Asha", "Meena", "Bala"};
    sort(names, names + 4);              // no need to write a sort
    for (const string& n : names) cout << n << " ";
    cout << endl;
    return 0;
}
```

▶ Output

```
Asha Bala Meena Ravi
```

Screen 5.4 Library reuse: one call to the standard `sort` replaces a hand-written sorting routine.

Workflow supported: plan → code → build → test → collaborate → deploy. Benefits: reuse, speed, consistency, teamwork, maintainability. Tools are a well-organised toolbox; libraries are ready-made parts.

★ **Development tools streamline the workflow; libraries provide reusable code — together they build better software faster.**

Chapter 5 summary

- Environment = editor, translator, debugger, run-time, build tools, libraries.
- Editor = writing only; IDE = editor + compiler + debugger + terminal + project tools.
- Compiler: whole program, executable, fast; interpreter: statement by statement.
- Java and Python are hybrid: compiled to bytecode, then interpreted/JIT.
- Debugger = correctness (breakpoints, stepping, watch, stack trace); profiler = performance.
- Linker: object files + libraries → executable (link time); loader: executable → memory (load time).
- Static vs dynamic linking; relocation; shared libraries.
- Runtime environment: memory, libraries, VM, I/O, exceptions (JVM, CLR).

Next: Chapter 6 — virtual computers: machine language, assembly, high-level languages, abstract machines and the Java Virtual Machine.

CHAPTER · UNIT VI · LANGUAGE DESIGN

6 Virtual Computers

OPENING SITUATION

A C++ programmer never sees transistors, voltages or even machine instructions. They see a "C++ computer" that understands `int`, `for` and `class`. That computer does not exist in hardware: it is a **virtual computer** built in layers of software — compiler, operating system, run-time library — on top of the real machine.

IN THIS CHAPTER

- 6.1 Concept of Virtual Computer
- 6.2 Machine Language and Assembly Language
- 6.3 High-Level Language and Virtual Machine
- 6.4 Abstract Machine
- 6.5 Java Virtual Machine
- 6.6 Role of Virtual Computers in Portability
- 6.7 Advantages and Limitations of Virtual Computers

YOU NEED

- Chapter 4: the von Neumann model. Chapter 5: compilers and run-time.

6.1 Concept of Virtual Computer

DEFINITION 6.1 — VIRTUAL COMPUTER

A **virtual computer** is an abstract or software-based representation of a computing system that provides an environment in which programs can run **as if they were on a real machine**. A software layer sits between user programs and the actual hardware.

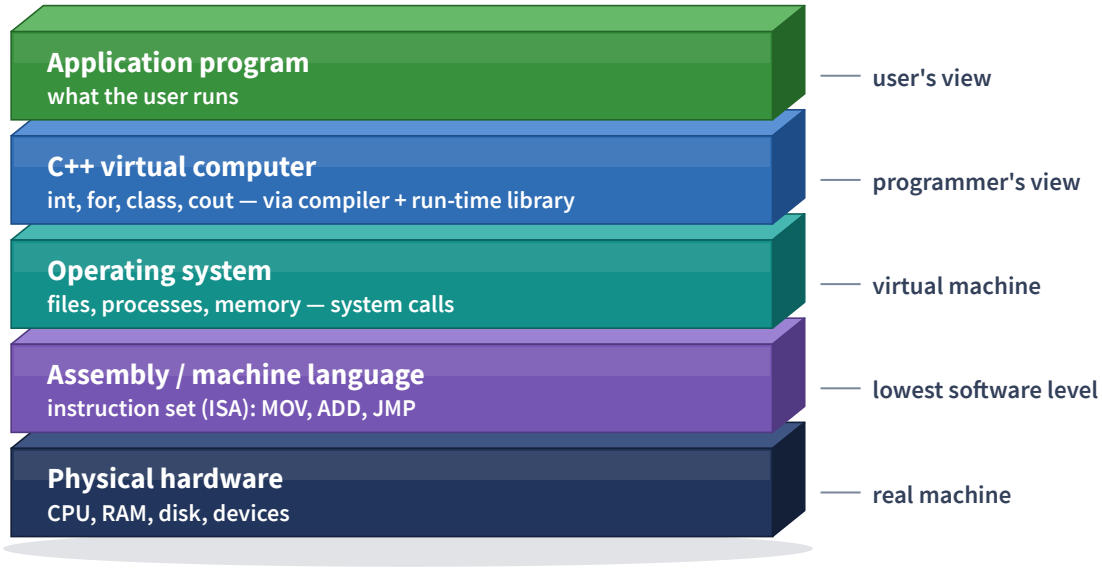
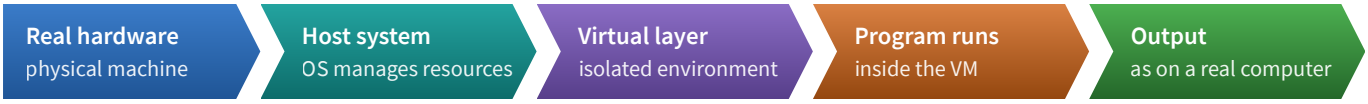


Figure 6.1 Layers of virtual computers. Each layer hides the one below it and offers a simpler "machine" to the layer above.

Table 6.1 Key concepts

Concept	Meaning
Physical machine	The actual hardware on which software runs
Virtual computer	A software-created system that mimics a real computer
Host system	The real machine that provides resources to virtual computers
Guest environment	The virtual environment inside the virtual computer where programs run
Abstraction layer	The software layer that hides hardware complexity and manages resources



Main features

- **Abstraction** — hides hardware complexity.
- **Isolation** — each virtual system runs independently.
- **Resource sharing** — CPU, memory, storage shared efficiently.
- **Flexibility** — many virtual systems at once; **repeatability**.

Benefits

- Safe space to learn and test; failures do not affect the host.
- Portability across hardware.
- Efficient resource use; a consistent, predictable execution model.

REAL-LIFE VIEW

A **flight simulator** behaves like a real aircraft cockpit: pilots practise safely without using the real plane. A virtual computer gives programs the same kind of realistic but controlled environment.

★ **A virtual computer is a software-defined computing environment that behaves like a real machine for running programs.**

6.2 Machine Language and Assembly Language

DEFINITION 6.2 — MACHINE AND ASSEMBLY LANGUAGE

Machine language is a low-level language of **binary instructions (0s and 1s)** executed directly by the CPU. **Assembly language** is a low-level **symbolic** language that uses **mnemonics** and labels for machine instructions; an **assembler** translates it into machine code.

Below is one real C++ function, the assembly the compiler produced for it (x86-64, 64-bit Windows: the arguments arrive in registers `ecx` and `edx`), and the machine-code bytes the assembler made from that assembly:

C++ add.cpp

```cpp
int add(int a, int b)
{
    return a + b;
}
```

add.s ASSEMBLY

```
lea eax, [rcx+rdx]
ret
```

add.o MACHINE CODE

```
10001101 00000100 00010001
11000011
```

Screen 6.1 One statement becomes two assembly instructions — `lea` computes `eax = a + b`, `ret` returns it — and four bytes of machine code (hexadecimal `8D 04 11` and `C3`). Each assembly line corresponds one-to-one with a machine instruction.

Table 6.2 Machine language versus assembly language

Aspect	Machine language	Assembly language
Representation	Binary (0s and 1s)	Mnemonics and symbols (`MOV`, `ADD`, `LOAD R1`)
Readability	Very hard	Easier for humans
Execution	Directly by the CPU	After assembly into machine code
Translator	None needed	Assembler required
Generation	First generation (1GL)	Second generation (2GL)

Key features

- Direct control over hardware; very fast, minimal overhead.
- Efficient use of memory and resources.
- **Hardware-specific, not portable**; difficult to write, debug, maintain.

Uses

- Firmware for microcontrollers and devices.
- Device drivers and system-level software.
- Embedded and IoT systems; boot loaders and start-up code.
- Performance-critical, time-sensitive routines.

★ **Machine language is the CPU's binary language; assembly language is its human-readable symbolic form, translated by an assembler.**

6.3 High-Level Language and Virtual Machine

DEFINITION 6.3 — HIGH-LEVEL LANGUAGE AND VIRTUAL MACHINE

High-level languages (HLL) — C++, Java, Python, C# — are programmer-friendly languages with human-readable syntax. A **virtual machine (VM)** is a software environment that executes translated code (**bytecode** or intermediate code) in a managed way.

Table 6.3 HLL versus VM

Aspect	High-level language	Virtual machine
Purpose	Writing programs with simple syntax	Executing / managing translated code
User	Programmers	The run-time system
Translation	Converted to bytecode / intermediate code	Runs bytecode, or manages its execution
Execution	Does not run directly on hardware	Executes code in a safe environment
Examples	Python, Java, C#, Kotlin	JVM, .NET CLR, Python VM

The heart of every VM is a loop that fetches, decodes and executes bytecode — a von Neumann machine written in software. This tiny **stack-based VM** runs a 6-instruction bytecode program for `(4 + 6) * 7`:

C++ tinyvm.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;
enum Op { PUSH, ADD, MUL, PRINT, HALT };
int main() {
    vector<int> code = {PUSH, 4, PUSH, 6, ADD, PUSH, 7, MUL, PRINT,
HALT};
    vector<int> stack;  int pc = 0;           // program counter
    while (true) {
        int op = code[pc++];                  // fetch + decode
        if (op == PUSH) stack.push_back(code[pc++]);
        else if (op == ADD || op == MUL) {    // execute
            int b = stack.back(); stack.pop_back();
            int a = stack.back(); stack.pop_back();
            stack.push_back(op == ADD ? a + b : a * b);
        }
        else if (op == PRINT) cout << "result = " << stack.back() <<
endl;
        else break;                           // HALT
    }
    return 0;
}
```

▶ Output

```
result = 70
```

Screen 6.2 A virtual machine in 20 lines: the bytecode is the same on every computer; only the VM (this C++ program) must be compiled for each platform.

Features

Abstraction from hardware, easier development (simple syntax, libraries), portability, run-time services (memory management, security, threads), managed execution.

Benefits

Higher productivity, "run on different platforms without changes", reduced hardware dependence, safety (VM protects system resources), easier debugging.

COMMON MISTAKE 6.1 — "C++ RUNS ON A VIRTUAL MACHINE LIKE JAVA"

Standard C++ is compiled **directly to native machine code** for one processor and OS — there is no bytecode and no VM. That is why a C++ `.exe` must be recompiled for Linux, while a Java `.class` file runs on any JVM. (C++/CLI, a Microsoft variant, is the exception: it targets .NET.)

★ **A high-level language helps humans write programs; a virtual machine provides a portable environment to execute them.**

6.4 Abstract Machine

DEFINITION 6.4 — ABSTRACT MACHINE

An **abstract machine** is a **conceptual model** of a computer that describes how programs execute, without detailing the physical hardware. It focuses on behaviour, rules and structure — a bridge between theory and implementation.

Table 6.4 Elements of an abstract machine

Element	Description
Instruction set	The operations the machine can perform
Memory model	How data and instructions are stored and accessed
Control mechanism	How instructions are fetched, decoded and executed
Input / output	How the machine interacts with external devices
Execution rules	Order and conditions of execution

Key features

Conceptual clarity (what, not how), simplified design, basis for interpreters and compilers, teaching aid, portability of concepts to many real machines.

Applications

Language specification (the C++ Standard defines behaviour on an abstract machine), compiler design (target model), virtual machines, **automata models** in theory, education.

REAL-LIFE VIEW

An abstract machine is like the **blueprint** of a building: it shows how rooms, connections and systems work without showing every brick and nail.

★ **An abstract machine is a simplified conceptual computer model used to describe how programs behave and execute.**

6.5 Java Virtual Machine

DEFINITION 6.5 — JVM

The **Java Virtual Machine (JVM)** executes Java **bytecode**. It provides a platform-independent run-time environment, so the same Java program runs on different systems without modification — "**Write Once, Run Anywhere**" (WORA).

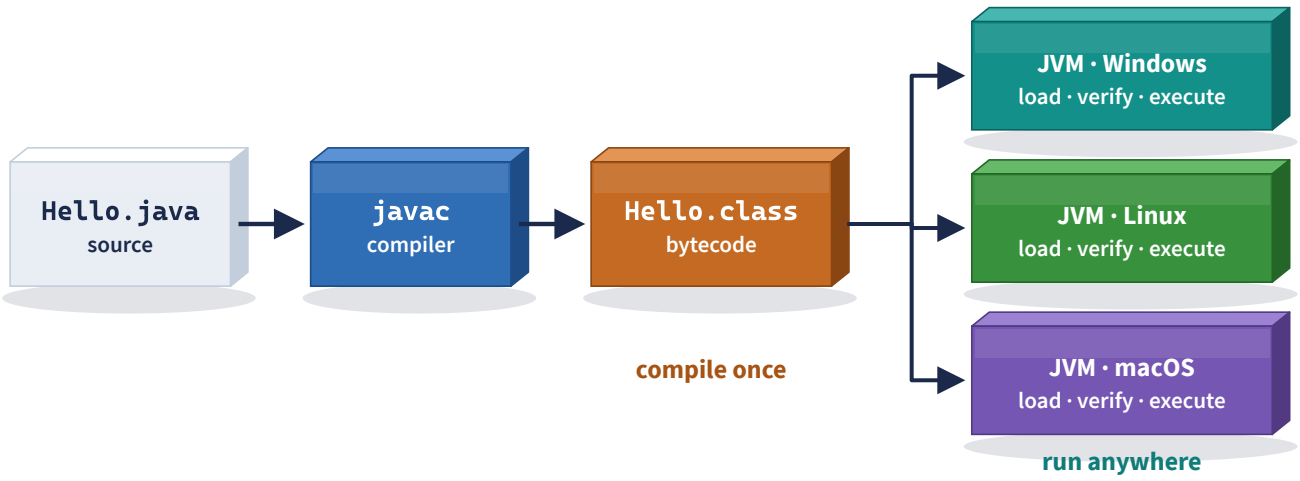


Figure 6.2 Java's path: `javac` compiles source to platform-neutral bytecode once; a JVM on each platform loads, verifies and executes it.

Table 6.5 JVM architecture

Component	Role
Class loader	Loads `.class` files into memory and links them
Bytecode verifier	Checks bytecode for safety, type correctness and security
Run-time memory areas	Method area, heap (objects), stacks, PC registers
Execution engine	Executes bytecode with an interpreter and/or **JIT (Just-In-Time) compiler**
Garbage collector	Automatically frees unused memory

EXAM POINT

Java source → `javac` → bytecode (`.class`) → JVM. The JVM executes **bytecode, not source code**. Features: platform independence, security (verification, sandbox), garbage collection, automatic memory management, exception handling. The **JVM itself is platform dependent** (a different JVM for each OS); the bytecode is platform independent.

★ **The JVM executes Java bytecode in a portable, managed run-time environment.**

6.6 Role of Virtual Computers in Portability

DEFINITION 6.6 — PORTABILITY

Portability is the ability of software to run on different hardware or operating systems with little or no modification. Virtual computers achieve it by providing an abstraction layer that hides hardware and OS differences.

Table 6.6 Without and with a virtual computer

Aspect	Without virtual computer	With virtual computer
Hardware dependence	Tied to specific hardware and OS	Abstracted; runs on many systems
Code changes	Often needed for each platform	Minimal or none
Deployment	Complex for multiple platforms	Easier and faster
Maintenance	Higher effort (many versions)	Centralised updates
Compatibility	Limited to the target platform	High across diverse systems

Advantages

Platform independence, software reuse (write once, use many times), faster deployment, easier updates, wider reach.

Challenges

Performance overhead, dependence on the run-time (VM must be installed), version compatibility issues, extra memory and CPU.

NOTE — TWO KINDS OF PORTABILITY

Source portability (C, C++): the same source compiles on any platform that has a compiler; you recompile for each target. **Binary portability** (Java bytecode, .NET): the same compiled file runs anywhere the VM exists.

REAL-LIFE VIEW

A **universal translator** lets one message (Hello, Bonjour, Hola) be understood by speakers of different languages. A virtual computer is that translator for software across different systems.

★ **Virtual computers improve portability by providing a common execution environment that hides differences among physical systems.**

6.7 Advantages and Limitations of Virtual Computers

Advantages

- **Portability** — move and use on different hardware and clouds.
- **Isolation and safety** — a failure rarely affects others or the host.
- **Resource sharing** — many systems on the same hardware.
- **Learning and testing** — ideal for safe experiments.
- **Reuse and flexibility** — snapshots, clones, quick rollback.

Limitations

- **Overhead** — an extra software layer reduces performance.
- **Complexity** — set-up, configuration and management.
- **Dependence** — needs the host and virtualisation software.
- **Performance cost** — some operations slower than on real hardware; debugging harder.
- **Resource consumption** — more memory and CPU.

REAL-LIFE VIEW

A **rental apartment** gives you a private, convenient space — but you still depend on the building management, shared utilities and its rules. Evaluate both sides and choose the approach that suits the need.

★ **Virtual computers provide flexibility and portability, but add overhead, complexity and dependence on supporting software.**

Chapter 6 summary

- Virtual computer: software layer that behaves like a real machine.
- Host = real machine; guest = virtual environment; abstraction layer hides hardware.
- Machine language: binary, 1GL, executed directly; assembly: mnemonics, 2GL, needs an assembler.
- HLL → bytecode → VM (JVM, CLR, Python VM); C++ compiles to native code.
- Abstract machine: conceptual model (instruction set, memory, control, I/O, rules).
- JVM: class loader, verifier, memory areas, execution engine (interpreter + JIT), GC; WORA.
- Source portability (C++) vs binary portability (Java).
- Pros: portability, isolation, sharing; cons: overhead, complexity, dependence.

Next: Chapter 7 — binding and binding times: when names get their types, addresses and values.

CHAPTER · UNIT VI · LANGUAGE DESIGN

7 Binding and Binding Times

OPENING SITUATION

You write `int marks = 90;`. When does `marks` become an `int`? When does it get a memory address? When does it hold 90? These things do not happen at one moment. The type is fixed when the compiler reads the line, the address when the function runs, and the value when the assignment executes. Each of these associations is a **binding**, and the moment it happens is its **binding time**. This one idea explains why C++ is fast, why Python is flexible, and how `virtual` functions work.

IN THIS CHAPTER

- 7.1 Meaning of Binding
- 7.2 Types of Binding
- 7.3 Static Binding
- 7.4 Dynamic Binding
- 7.5 Binding Time Concepts
- 7.6 Language Design Time
- 7.7 Language Implementation Time
- 7.8 Compile Time
- 7.9 Link Time
- 7.10 Load Time
- 7.11 Run Time
- 7.12 Importance of Binding in Language Translation

YOU NEED

- Chapter 5: compiler, linker, loader. Chapter 6: machine code and virtual machines.

7.1 Meaning of Binding

DEFINITION 7.1 — BINDING

Binding is the **association between a program entity and one of its attributes**. Examples: a name and its type, a variable and its memory location, an operator symbol and the operation it performs. The **binding time** is the moment at which that association is made.

Table 7.1 What gets bound to what

Entity	Attribute bound to it	C++ example
Variable name	Memory location and value	`price` → address, 99.5
Identifier	Data type	`double price;`
Operator symbol	Implementation (which operation)	`+` → integer add / string join
Function name	Code body to execute	`area()` → its machine code
Scope name	Region where the name is visible	a local visible only in its block

One variable carries several bindings at once. In this program the **type** and **address** of `price` stay fixed, while the **value** binding changes: