SWEG2102 Fall 2026

Fundamentals of Programming II

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

Lecture 14

Programming Best Practices

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

Learning Objectives

By the end of this lecture, you will be able to:

- Explain the importance of programming best practices in software development.
- Apply proper **naming conventions** and **code readability** techniques.
- Organize code and structure using **functions**, **headers**, and **modular design**.
- Implement proper **error handling** and **debugging** strategies.
- Choose and use appropriate **data structures** for efficient problem-solving.
- Apply **code optimization** techniques to write performant programs.

Introduction to Programming Best Practices

What Are Programming Best Practices?

Definition

Programming best practices are a **set of** proven **guidelines**, **conventions**, and **techniques** that help developers write code that is readable, maintainable, scalable, efficient, and reliable.

- *They are not **strict rules** imposed by the programming language.*
- *They are recommendations developed through years of industry experience and research.*

Good programmers do not simply write code that works; they write code that can be understood, modified, tested, and maintained by others following industry best practices.

Cont'd

Real-world Analogy:

Flight Analogy

Source: *AI-generated image (ChatGPT)*

Cont'd

Why Best Practices Matter

📖 Readability

- Code is **read** more often than it is **written**.
- Clear and well structured code is easy to understand and reduces cognitive load.

🔧 Maintainability

- Well-structured code is easier to read, modify, and extend by any developer.
- Saving time and money long-term.

🏆 Professionalism Standards

- Allow you to write professional-quality code that meet industry grade code.

⚡ Performance & Efficiency

- Choosing the right data structures, good memory management and optimizing algorithms reduces execution time and improve program efficiency.

Fewer Bugs & Errors

Following consistent patterns, input validation and error handling prevent program crashes

🤝 Team Collaboration

- Teams work better when everyone follows shared conventions including consistent style, using version control, and documentation.

Cont’d

Characteristics of Professional-Quality Code

Correct	Produces the expected results.
Readable	Easy to understand.
Maintainable	Easy to modify and extend.
Reliable	Handles errors appropriately.
Reusable	Can be used in multiple contexts.
Efficient	Uses resources responsibly.
Well-Documented	Provides useful explanations when necessary.

Writing Readable and Maintainable Code

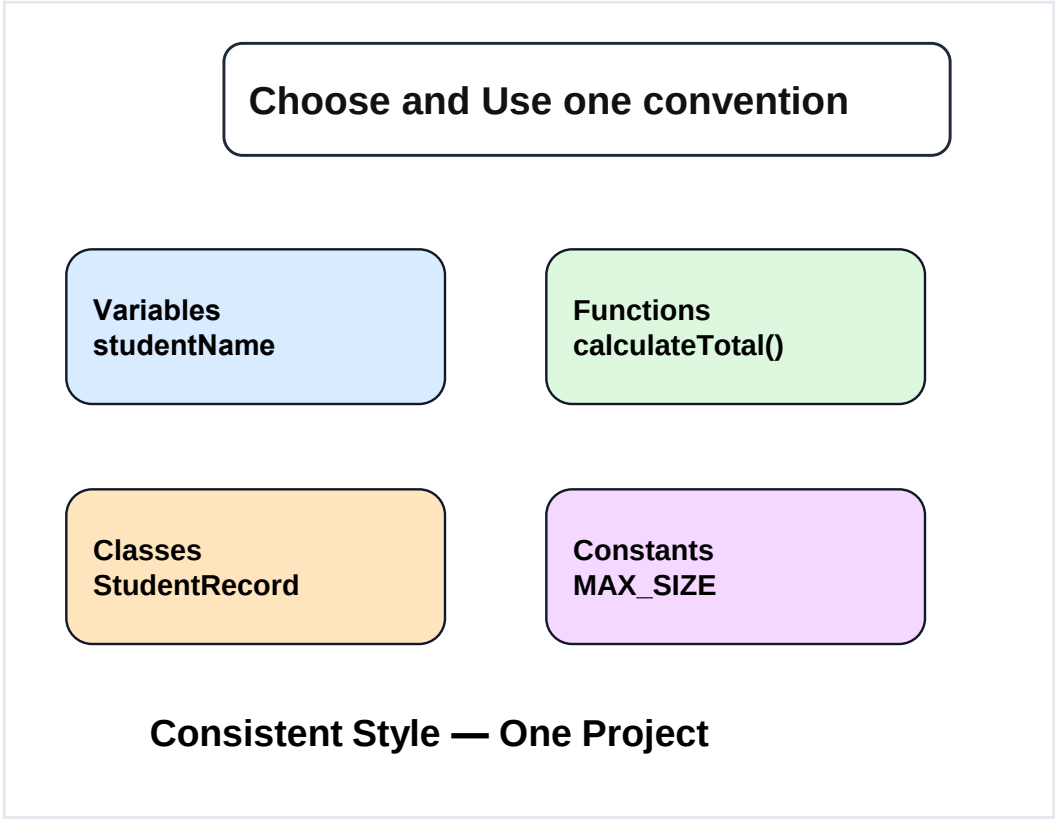
2.1 Naming Conventions

Core Concepts

Clear names make code readable and predictable:

- Variables use descriptive nouns: studentName, totalScore
- Functions start with verbs: calculateTotal(), printReport()
- Classes use PascalCase: BankAccount, StudentRecord
- Constants use UPPER_CASE: MAX_SIZE, TAX_RATE

Visual Diagram



Note: 👉 Consistent names make code easier to read, test, and maintain.

Cont’d

Example: Naming Conventions in C++

```
// camelCase → local variables & functions
int studentAge = 20;
void displayResult();

// PascalCase → class names
class BankAccount { ... };

// snake_case → constants / config
const int max_retry_count = 3;
```

```
// BAD Practice – unclear names
int x = 86400;
void calc(int a, int b) { ... }

// GOOD Practice – meaningful names
const int SECONDS_PER_DAY = 86400;
void calculateInterest(int principal, int rate) { ... }
```

</> Cont'd

4 BENEFITS OF WRITING READABLE CODE

Readable code isn't just nice to have—it's a superpower for you and your team.

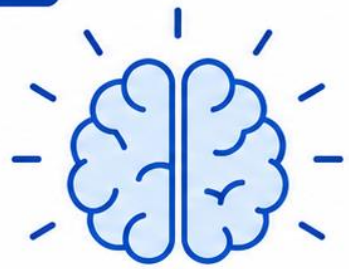
1 Easier Debugging

Readable code is easier to understand, so you can quickly identify issues, trace logic, and fix bugs with less time and frustration.

2 Faster Onboarding

New team members can understand the codebase quickly, get productive sooner, and contribute with confidence.

3 Reduces Cognitive Load



Clear, logical, and consistent code is easier to read and reason about. It helps your brain focus on solving problems instead of figuring out what the code is doing.

4 Prevents Magic Number Bugs

Readable code encourages the use of meaningful names and constants instead of unexplained numbers, reducing errors and making the code more reliable and maintainable.

Bottom line: Readable code saves time, reduces bugs, empowers your team, and builds better software.

2.2 Code Formatting & Organization

Goal: make every file predictable

Apply consistent style rules so reviews focus on correctness instead of formatting debates.

Code Formatting & Style

- Set one indentation, spacing, and brace style.
- Follow naming conventions for variables, classes, and methods.
- Break long expressions into clear readable parts.
- Use automated formatters to prevent style drift.

Code Organization

- Group related functions, data, and classes together.
- Keep each file or method focused on one responsibility.
- Order code from high-level flow to supporting details.
- Separate reusable logic from test, demo, or setup code.

Best approach:

- *choose a style guide, apply it consistently, and organize code around reader intent.*

Readable Code: Comments & Clarity

Code documentation / Comments

- Use comments to explain purpose, assumptions, and edge cases.
- Document method inputs, outputs, and side effects when helpful.
- Update comments when behavior changes so they stay trustworthy.
- Avoid repeating obvious statements already clear from the code.

Self-explanatory code

- Name variables and methods after the job they perform.
- Prefer small functions with clear intent.
- Use constants to replace unexplained “magic” values.
- Let structure reduce the need for extra comments.

Prefer: meaningful names + small functions + focused comments = code that future readers can trust.

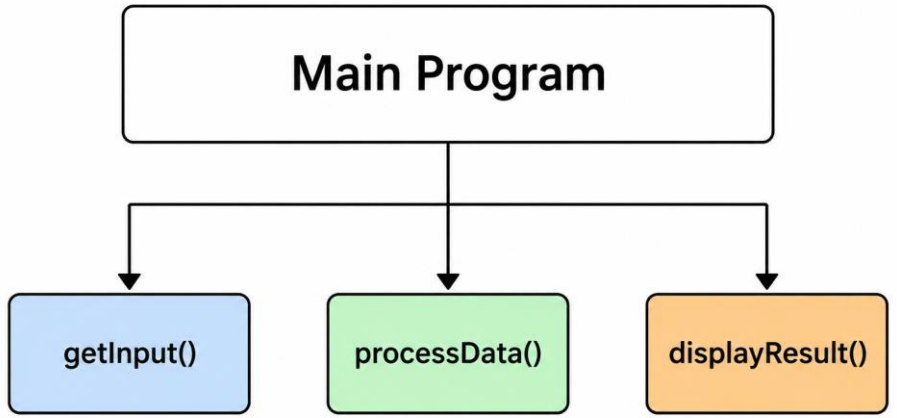
Modular Design

Core Concepts

Modular design is the practice of dividing a program into independent, self-contained units (functions, classes, files) — each with a single, well-defined responsibility.

- Each function does **one thing only** (**Single Responsibility Principle**).
- Code is not repeated — shared logic lives in **one reusable function** (**DRY Principle**).

Visual Diagram



Source:
AI-generated image (Gemini 3)

Note: 👉 Modular code is easier to **test**, **reuse**, and **maintain** — one function, one job.

SRP & DRY

Two concepts that keep programs easier to change and maintain

SRP — Single Responsibility Principle

Definition: A class, function, or module should do one job and have one clear reason to change.

How to apply:

- Split mixed responsibilities into focused units.
- Keep UI, validation, data access, and business rules separate.
- Name each unit by the single job it performs.

DRY — Don't Repeat Yourself

Definition: Each piece of knowledge or business logic should live in one place, not be copied across the codebase.

How to apply

- Extract repeated code into a shared function, class, or configuration.
- Reuse constants for repeated values.
- Update the rule once and let all callers use it.

Relationship: SRP separates responsibilities; DRY centralizes repeated knowledge.
Use both to make changes smaller, safer, and easier to test.

Cont'd

SRP Vs. DRY — Key Differences

Aspect	SRP (Single Responsibility)	DRY (Don't Repeat Yourself)
Question answered	How many jobs does this function have?	Is this logic duplicated elsewhere?
Concept	One function = one responsibility	Every piece of logic expressed once
Core focus	Separation of concerns	Elimination of redundancy
Achieved via	Small, focused functions & classes	Shared utility functions, constants
Example	printReport() only prints — never calculates	calculateTax() defined once, used everywhere
Dependency	Can be applied independently	Often requires SRP to be effective

Memory hook:

SRP = One function, one job **(the chef rule)**.

DRY = Write it once, use it everywhere **(the recipe rule)**.

3.1 Common Readability Problems

Recognize the warning signs

> Readable code makes the intent clear. When readers must guess, errors become harder to find, fix, and test.

Naming & formatting

- Vague names like data, temp, or x
- Inconsistent indentation, spacing, or line length

Program structure

- Functions that do several unrelated jobs
- Deep nesting and repeated logic in many places

Hidden meaning

- Magic numbers with no explanation
- Comments that repeat code instead of explaining why

Memory hook: If it takes extra effort to explain, refactor it before it grows.

Quick Check

1. Which naming style should be used for class names in C++?

A. PascalCase
B. camelCase
C. snake_case
D. UPPER_CASE

Answer: A → PascalCase

2. What is the PRIMARY problem with using magic numbers in code?

A. slow down execution

B. make code unreadable and error-prone

Answer: B

3. Which of these variable names follows best practices?

A. int d = 30; — because it's short

B. int x1 = 30; — because it has a number

C. int age = 30; — because it's self- explanatory

D. int DAYS = 30; — because it's uppercase

Answer C

Quick Check

4. What does the Single Responsibility Principle state?

A. A function should handle multiple tasks .

B. Functions should never call other functions

C. Code should not be split across files

D. A class should have only one reason to change

Answer: D

5. Where should function declarations be placed in a well-organized C++ project?

A. In a .h (header) file

B. In the main.cpp file only

C. At the bottom of each .cpp file

D. Declarations are not needed in C++

Answer: A

6. Which of these violates the DRY principle?

A. A utility function used in 5 different files

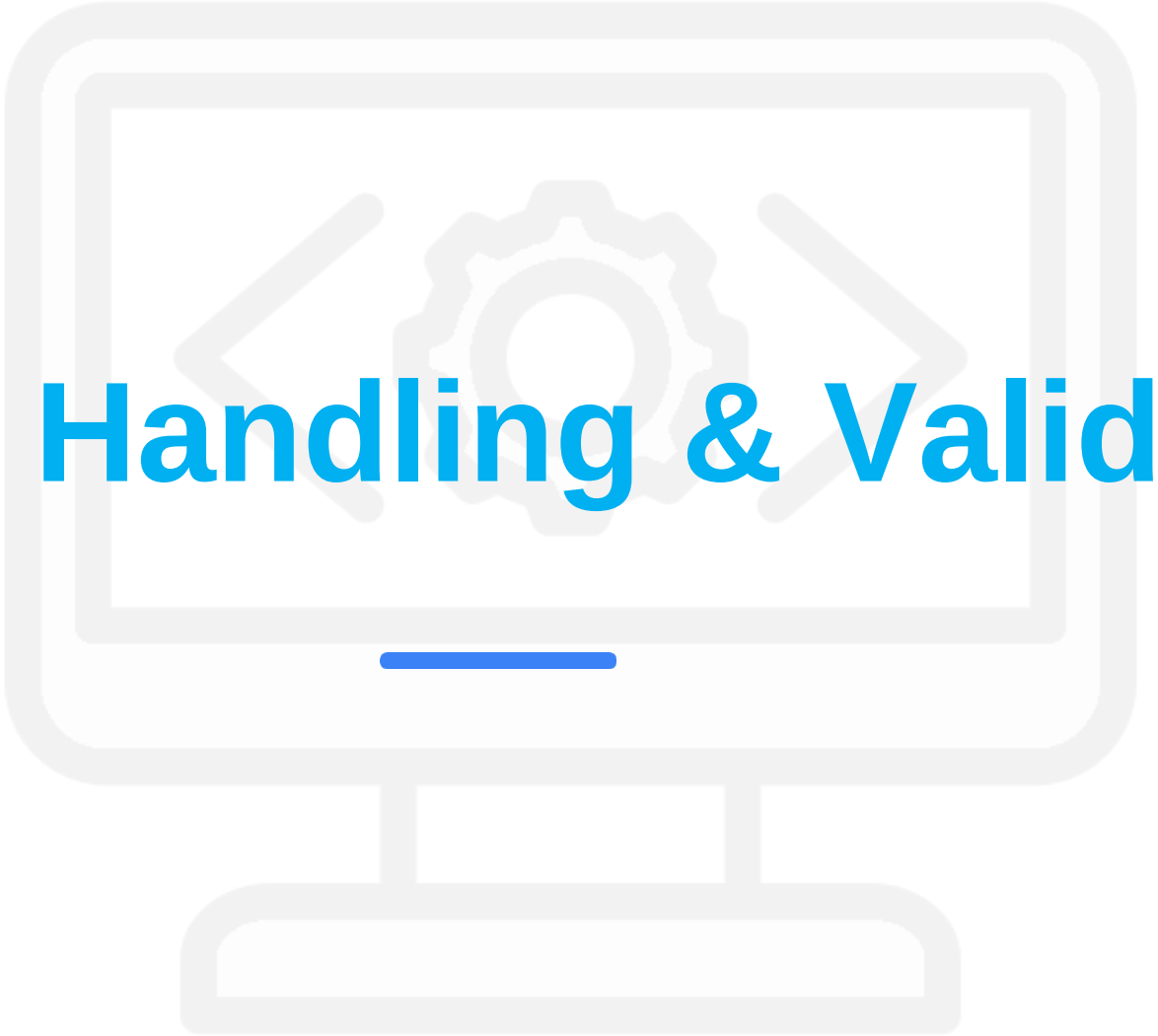
B. The same formula written in 4 separate functions

C. A constant defined once & referenced everywhere

D. A class delegating tasks to helper functions

Answer: B

Error Handling & Validation



3.2 Error Handling & Validation

Core Concepts

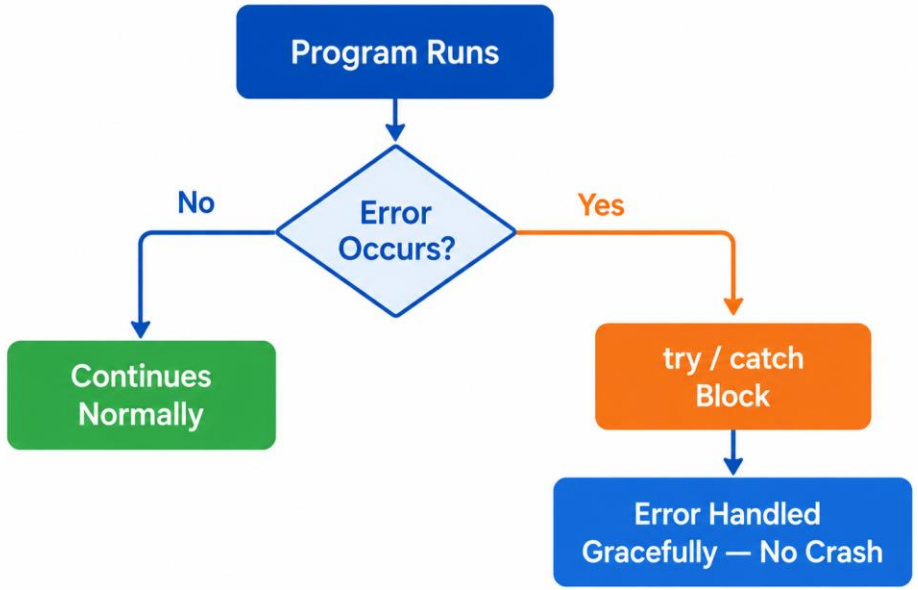
Error handling is the process of anticipating, detecting, and responding to program errors and unexpected inputs — ensuring the program behaves correctly even when things go wrong.

- *Programs encounter three types of errors: **syntax**, **runtime**, and **logic***
- *Unhandled errors cause **crashes**, **data loss**, and **security vulnerabilities***

Purpose:

- **Catch** and **recover** from runtime errors, **validate** user inputs, and provide user-friendly error messages.

Visual Diagram



Source:
AI-generated image(Gemini 3)

Cont'd

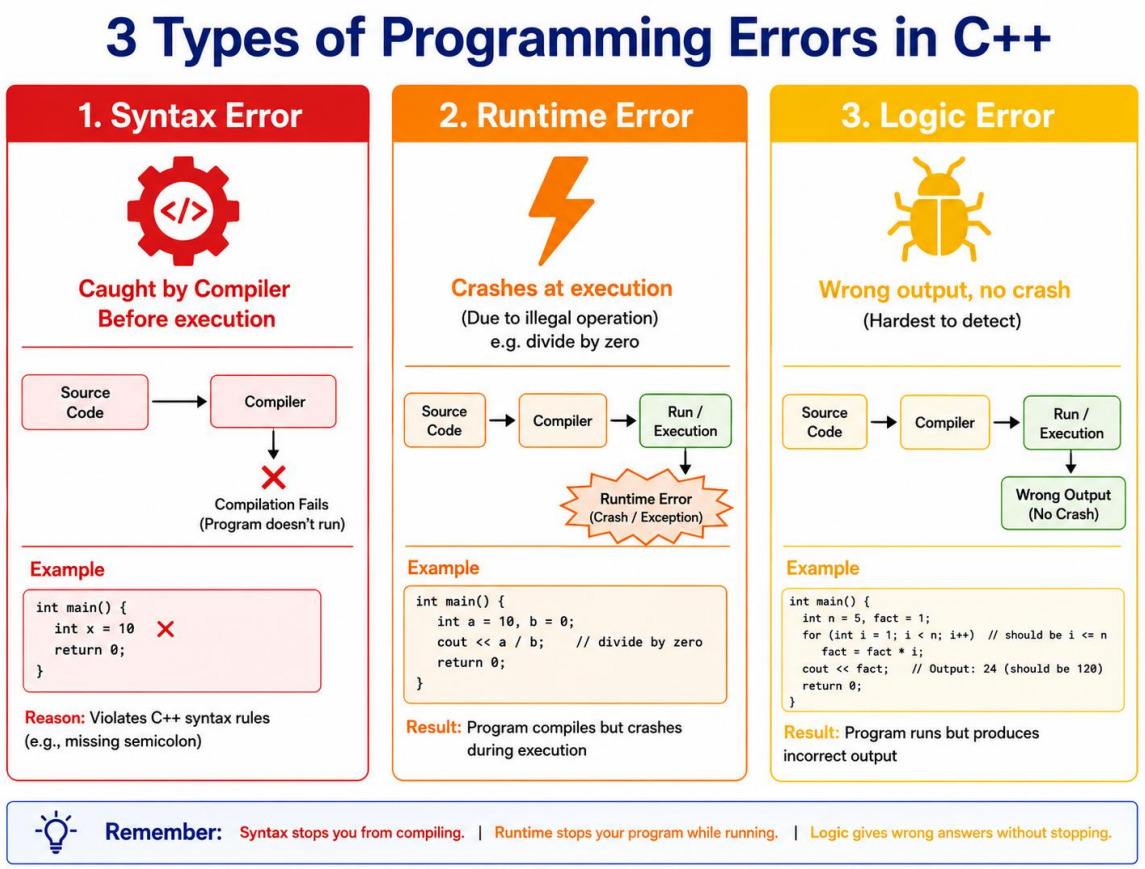
Types of Errors in C++

- **Syntax Errors**:
  - ✓ Detected by the compiler before execution.
  - ✓ E.g., missing semicolons, undeclared variables.
  - ✓ Example: int x = ; → compiler error
- **Runtime Errors**:
  - ✓ Occur during execution — program compiles but crashes.
  - ✓ E.g., division by zero, accessing null pointer.
  - ✓ Example: int r = x / 0; → crash at runtime

Logic Errors:

- Program runs but wrong output
- Hardest to detect
- E.g: Using > instead of >=

Visual Diagram



Source:
AI-generated image (ChatGPT)

Cont’d

Error Handling Techniques

1. try / catch

- Wraps risky code and catches exceptions to prevent crashes.

```
try { } catch(e) { }
```

2. throw

- Signals an error — can throw built-in types or custom exception objects.

```
throw invalid_argument("msg")
```

3. Custom Exceptions

- Create your own exception classes derived from std::exception for domain-specific errors.

```
class MyError : public exception
```

4. Input Validation

- Check all user inputs before use — reject negatives, empty strings, or out-of-range values.

```
if (age < 0) throw ...
```

5. static_assert

- Compile-time assertion — fails at build time if a condition is false. Catches bugs before execution.

```
static_assert(N > 0, "...")
```

Examples: Validation & Exceptions

Example in C++

#1 Input Validation

```cpp
// #1 Input Validation
int getAge() {
    int age;
    cin >> age;
    if (age < 0 || age > 150)
        throw invalid_argument("Age out of range.");
    return age;
}
```

#2 try / catch / throw

```cpp
// #2 try / catch / throw
int main() {
    try {
        int age = getAge();
        cout << "Age: " << age << endl;
    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << endl;
    }
}
```

#3 Custom Exception Class

```cpp
// #3 Custom Exception
class NegativeValueError : public exception {
public:
    const char* what() const noexcept override {
        return "Negative value is not allowed.";
    }
};
```

Examples: Debugging & Messages

Example in C++

#4 Debugging Techniques

```cpp
// #4 Debugging: use cerr and assert
#include <cassert>
void setScore(int s) {
    assert(s >= 0 && s <= 100); // halts in debug mode
    score = s;
}
// compile with: g++ -Wall -Wextra main.cpp
```

#5 User-Friendly Error Messages

```cpp
// #5 User-Friendly Messages
catch (const exception& e) {
    cerr << "[ERROR] " << e.what() << "\n";
    cerr << "Please enter a valid positive number.\n";
}
```

Note

- **Ignoring compiler warnings**:
  - ✓ Never ignore compiler warnings — they often signal real bugs. Enable all warnings with -Wall -Wextra flags when compiling with g++.

Why Reliable Error Handling Matters

Why Error Handling?

Prevents Crashes:
Exception handling keeps programs running on unexpected input.

Improves User Experience:
Error messages help users fix input instead of cryptic crashes.

Isolates Failures:
Errors are caught at failure point, easing bug tracing and fixes.

Enforces Data Integrity:
Input validation prevents invalid data and silent corruption.

Required for Robust Software:
Production C++ handles edge cases, not assuming valid input.

Common Error Handling Mistakes:

- ✕ Using empty catch blocks that swallow errors silently
- ✕ Catching ... (all exceptions) without logging or re-throwing
- ✕ Skipping input validation because "the user will enter correct data"

Tip: Always log errors to **cerr** — never let failures pass silently.

Quick Check: Error Handling & Debugging

1. Which block in C++ is used to signal (raise) an error?

A. try

B. catch

C. throw

D. assert

Answer: C

2. What is the difference between a runtime error and a logic error?

A. Runtime errors may crash; logic errors produce wrong output silently

B. They are the same

C. Logic errors always crash the program

D. Runtime errors only occur in loops

Answer: A

2. A function receives age = -5 from the user. What should the program do?

Answer: Validate the input and throw an invalid_argument exception with a descriptive message.

Efficient Memory Management

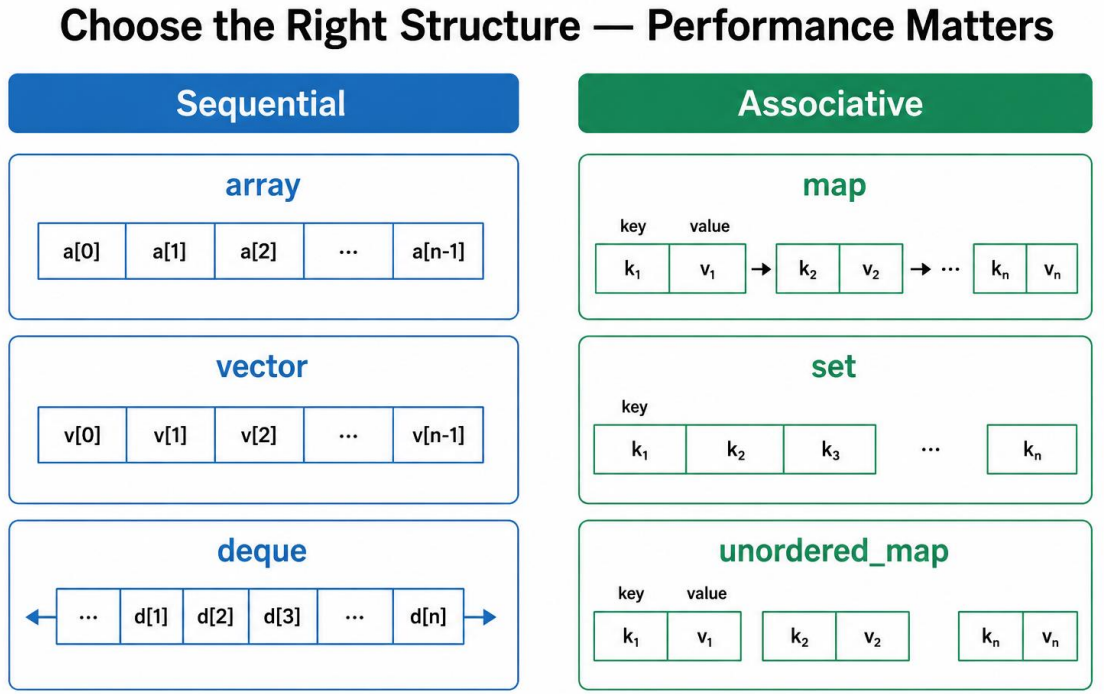
Efficient Data Structure Usage

Core Concepts

Data structures are organized formats for storing, accessing, and managing data in memory. Choosing the right structure directly impacts program speed and efficiency.

- Arrays offer fast indexed access but **fixed size**.
- **vector, map, set** from the C++ STL cover most use cases.
- Always consider time and space complexity (**Big-O**) when choosing.
- Key structures:
  - *Sequential — array, vector, deque*
  - *Associative — map, set, unordered_map*

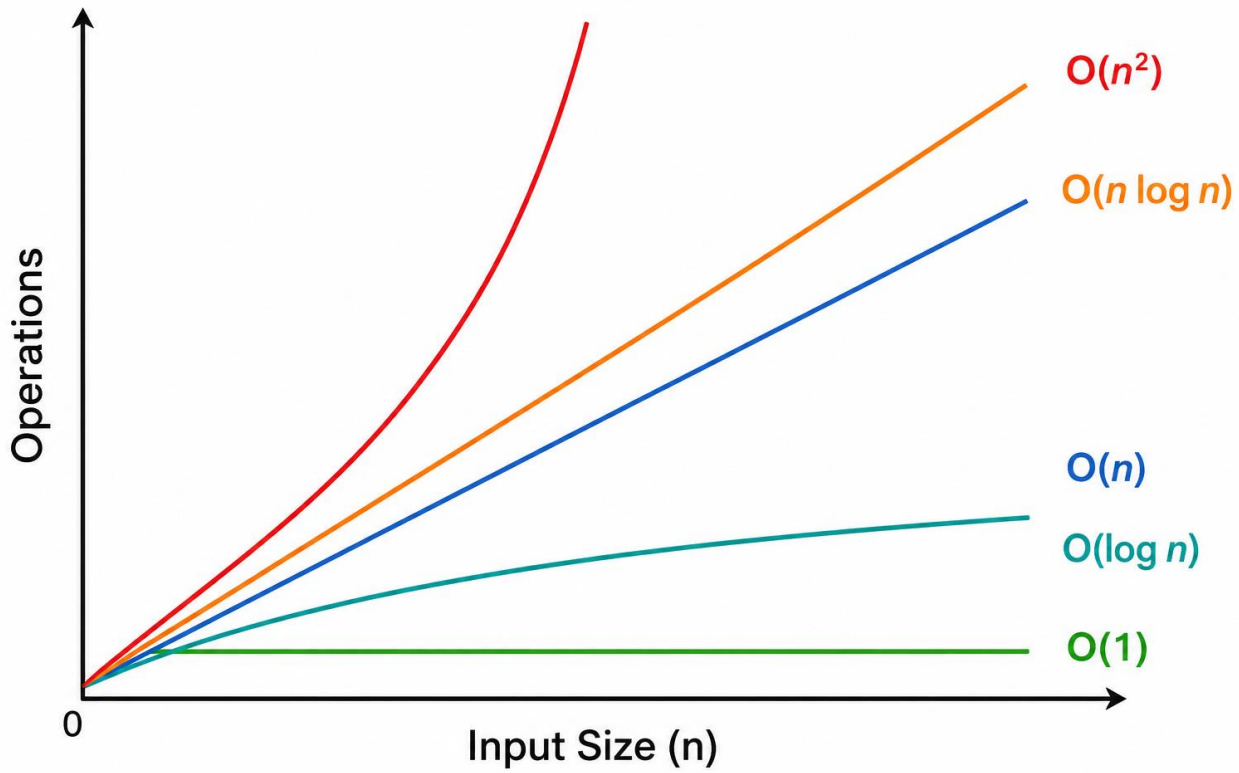
Visual Diagram



Source:
AI-generated image(Gemini 3)

Cont'd

Visual Example



Big-O	Name	Example
$O(1)$	Constant	Array index access arr = [10 20 30 40 50] 0 1 2 3 4 arr[2] → 30
$O(\log n)$	Logarithmic	Binary search low mid high 1 3 5 7 9 11 13 15 Search 7: found in the middle
$O(n)$	Linear	Loop through vector `for (int x : vec) { process(x); }` vec 10 20 30 40 50 0 1 2 3 4
$O(n \log n)$	Linearithmic	std::sort 5 2 9 1 5 6 → 1 2 5 5 6 9
$O(n^2)$	Quadratic	Nested loops `for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) work(i, j);` 0 1 2 … n-1 Each element is paired with every other element. Total ~ n^2 operations.

Source:
AI-generated image(Gemini 3)

Memory Management in C++

Memory Management

Safe dynamic memory follows one clear ownership path.

Manual memory flow

1. **Allocate** — new
2. **Own** — owner
3. **Release** — delete

minimal safe pattern

```
int* arr = new int[10];
// use arr
delete[] arr;

// better
auto arr2 = make_unique<int[]>(10);
```

Decision rule

1. **Match operation** — delete with new; delete[] with new[]
2. **Name owner** — one cleanup path
3. **Prefer RAII** — smart pointers first

Ownership prevents leaks.

RAII & Smart Pointers in C++

RAII & Smart Pointers

RAII makes cleanup automatic at scope exit.

Scope controls resource lifetime

scope-based ownership

```
#include <memory>

auto p1 = make_unique<int>(42);
// freed at scope exit

auto p2 = make_shared<int>(100);
auto p3 = p2;
```

Ownership model

unique_ptr
sole owner

shared_ptr
shared owner

RAII principle
cleanup by lifetime

Recommended default
Prefer make_unique / make_shared.

Quick Check

1. Which STL container provides O(1) average-time key-value lookup?

A. vector
B. map
C. unordered_map
D. set

Answer: C

2. What is the correct way to free a dynamically allocated array in C++?

A. free(arr)
B. delete arr
C. delete[] arr
D. arr = nullptr

Answer: C

3. What principle guarantees resource cleanup even when an exception is thrown?

A. DRY
B. SRP
C. RAII
D. Big-O

Answer: C — RAII (Resource Acquisition Is Initialization)

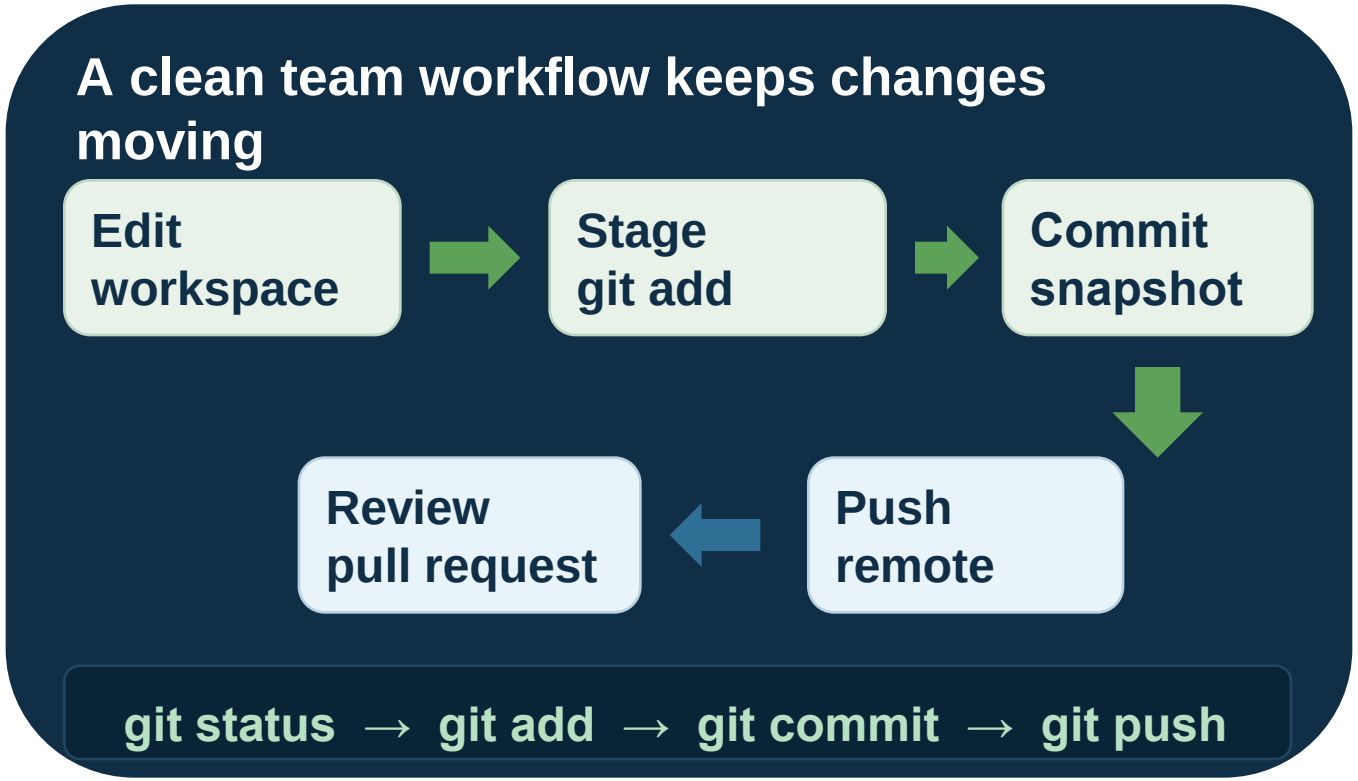
Version Control and Collaboration

Version Control and Collaboration

Git

Version Control and Collaboration

Use a shared workflow so every code change is traceable, reviewable, and easy to recover.



Collaboration habits

Align work before changes merge.

1. **Branch for every change**
   Separate experiments from stable work.
2. **Commit in small units**
   Review or roll back each change easily.
3. **Review before merge**
   Resolve conflicts early.

Rule: commit small, message clearly, and review before merging.

Practice Exercise

Practical Exercise:

Student Grade Report System

Objective

Implement a Student Grade Report System in C++ that demonstrates:

1. **Modular Design**
   - *Separate input, processing, and output into distinct functions*
2. **Error Handling**
   - *Validate student scores (0–100); throw exceptions for invalid input*
3. **Data Structures**
   - *Store student records using vector and map*
4. **Output Formatting**
   - *Use iomanip to format a grade table*

Requirements

- A university system manages student names and their scores for 3 subjects.
- Each student has: name, 3 subject scores, and a calculated average.
- Read student name and 3 scores; throw invalid_argument if any score < 0 or > 100
- Calculate average; assign letter grade (A/B/C/D/F); store in vector<Student>

✓ Display a formatted table using setw(15) and setprecision(2) for alignment

Cont’d

Solution:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
#include <stdexcept>
using namespace std;

struct Student {
    string name;
    double scores[3];
    double average;
    char   grade;
};
```

```cpp
// Input Module
Student readStudent() {
    Student s;
    cout << "Enter name: ";
    cin  >> s.name;
    for (int i = 0; i < 3; i++) {
        cout << "Score " << i+1 << ": ";
        cin  >> s.scores[i];
        if (s.scores[i] < 0 || s.scores[i]
        > 100)
            throw invalid_argument("Score
            out of range (0-100).");
    }
    return s;
}
```

```cpp
// Processing Module
void processStudent(Student& s) {
    double sum = 0;
    for (int i = 0; i < 3; i++) sum += s.scores[i];
    s.average = sum / 3.0;
    if      (s.average >= 90) s.grade = 'A';
    else if (s.average >= 80) s.grade = 'B';
    else if (s.average >= 70) s.grade = 'C';
    else if (s.average >= 60) s.grade = 'D';
    else                      s.grade = 'F';
}

// Output Module – formatted table
void printReport(const vector<Student>& students) {
    cout << left
         << setw(15) << "Name"
         << setw(10) << "Average"
         << setw(6)  << "Grade" << "\n";
    cout << string(31, '-') << "\n";
    for (const auto& s : students) {
        cout << setw(15) << s.name
             << setw(10) << fixed << setprecision(2) << s.average
             << setw(6)  << s.grade << "\n";
    }
}
```

Best Practices Checklist

Use this checklist before submitting your program

A clean solution should be readable, testable, and consistent with the assignment requirements.

1 Plan the algorithm

Define inputs, outputs, steps, and edge cases before coding.

2 Keep modules focused

Use small functions; each function should do one job clearly.

3 Use clear names

Choose meaningful names for variables, functions, and files.

4 Validate and guard data

Check input ranges, vector indexes, and invalid cases early.

5 Test incrementally

Compile often; test normal, boundary, and error scenarios.

6 Format and clean up

Use setw/precision, add useful comments, and remove dead code.

Final check: run, compare with requirements, then submit the cleaned version.

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia