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 09

File Management (Part I)

Handling Text Files

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

Outline

Topics Covered

01 Introduction to File Management

02 Types of Files & File Operations

03 File Streams in C++

04 Opening & Closing Files

05 Writing to and Reading from Text Files

06 File Operation Error Handling

07 Practical Integrated Example

Learning Objectives

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

- Explain the concept and importance of **file management** in programming.
- Differentiate between **text & binary files**, and describe common file operations.
- Apply **C++ file stream classes** to manage file input and output operations.
- Open, close, read from, and write to text files using appropriate file modes
- Apply basic **file error handling techniques** to validate file operations.
- Develop simple C++ programs that integrate multiple file handling concepts.
- Identify common file operation **mistakes** and apply **best practices**.

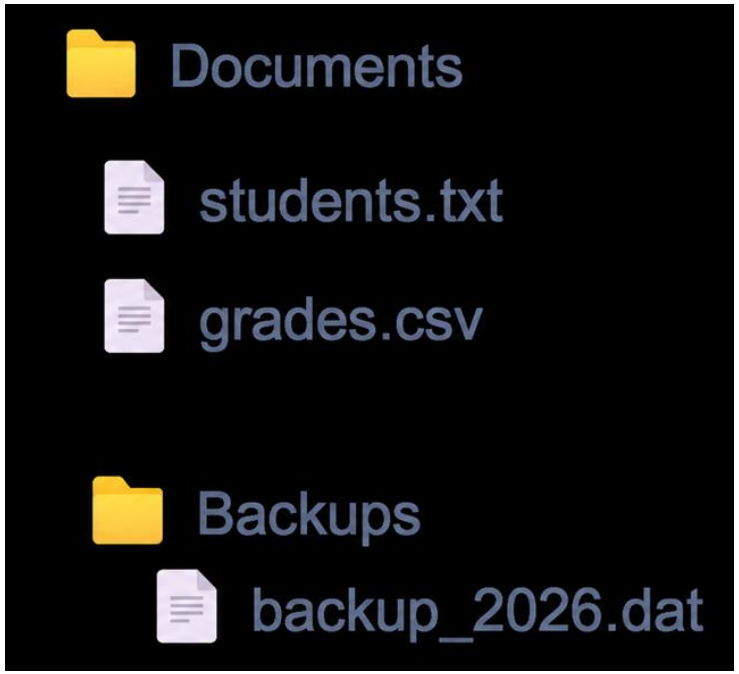
Introduction

What is a File?

Definition

A **file** is a **named collection** of related data stored on a persistent storage device (*hard disk, SSD, USB drive).*

- Every file has these key properties:
  - ✓ Names & Extension
    - *students.txt, data.csv, etc.*
  - ✓ Paths / Location
    - */home/user/docs/ or C:\Users\Name\Desktop*
  - ✓ Size / occupied memory space
    - *1 KB, 5 MB, 2 GB — depends on content*
  - ✓ format



Source: *AI-generated image (ChatGPT)*

Cont’d

Why do file matters?

Real-World Motivation - Without files

- every time you restart a **game**, you start from level 1 — no save data
- your **browser** forgets all bookmarks and login sessions every restart
- **banks** cannot maintain account balances — all transactions lost on reboot

From Program Perspective

- Programs need to save user data between runs
- Databases are built on file systems
- Games save progress; apps save preferences

Why Programs Need Permanent Storage?

RAM vs Disk Storage — The Fundamental Trade-off:

Aspect	RAM (Volatile)	Disk / File (Persistent)
Speed	Very fast (nanoseconds)	Slower (milliseconds)
Persistence	Lost when program exits	Survives program termination
Capacity	Limited (GB range)	Large (TB range)
Use Case	Active program execution	Long-term data storage

Data Persistence in Action

Lifecycle of a Program with File I/O:

Program Run 1

User enters data (name, score). Program writes to file students.txt. Program exits — RAM cleared.

File on Disk

students.txt persists on disk. Data is safe even when computer is powered off. File contains: 'Alice 95 Bob 78'.

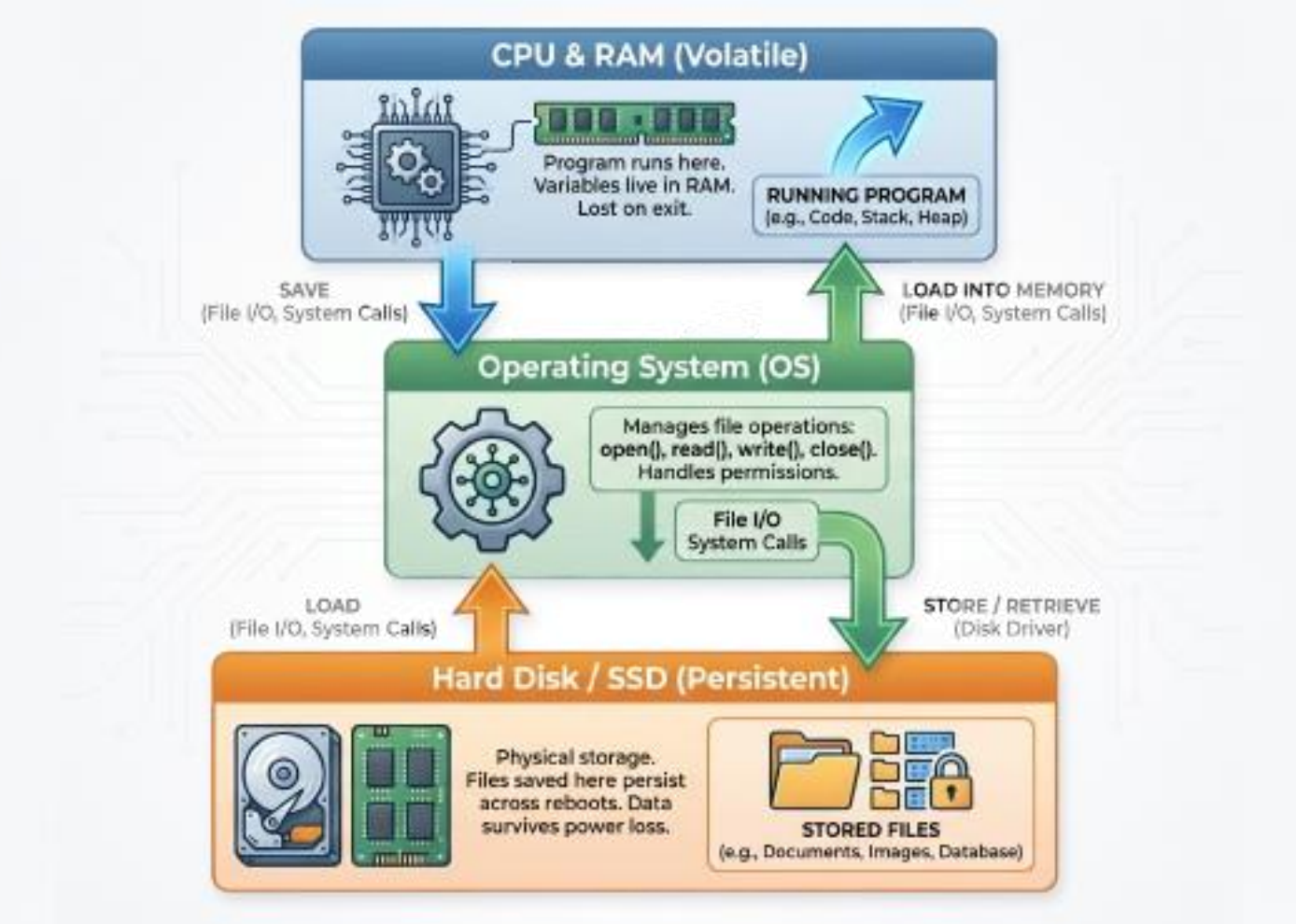
Program Run 2

Program restarts. Reads students.txt. Loads Alice and Bob's data back into RAM. Continues from saved state.

Cont’d . . .

Visual Diagram:

Computer Storage Architecture



Source: Conceptual system architecture diagram (author-created)

Types of Files

Files are categorized by content format — know the difference:

Text Files

- ▶ Human-readable ASCII/UTF-8 characters
- ▶ Can open and edit in any text editor (Notepad, VS Code, vim)
- ▶ Examples: .txt, .csv, .log, .html, .cpp source code
- ▶ Advantages: portable, editable, debuggable
- ▶ Disadvantages: larger size, slower parsing for complex data

Binary Files

- ▶ Machine-readable bytes — not human-readable
- ▶ Requires specific program to interpret
- ▶ Examples: .exe, .jpg, .png, .pdf, .dat, .bin
- ▶ Advantages: compact, fast processing, efficient for structured data
- ▶ Disadvantages: not portable, harder to debug, format-specific tools needed

Advantages of File Handling

Why File Handling is Essential:

Data Persistence

Save program state between executions — no need to re-enter data every run

Large Data Storage

RAM is limited; files store gigabytes of data that don't fit in memory

Backup & Recovery

Files enable backup strategies — duplicate critical data to prevent loss

Data Sharing

Files enable data exchange between programs and users

Scalability

Programs can process datasets larger than available RAM by reading in chunks

Portability

Text files especially are portable across different systems and languages

File Operations Overview

1. Open File

Create or open file stream with specified mode

- Entry Point

2. Check Status

Verify file opened successfully

- Validation

3. Process Data

Read, write, or append operations

- Core Operations

4. Close File

Save changes and release resources

- Cleanup

→ Flow Direction →

Key Operations

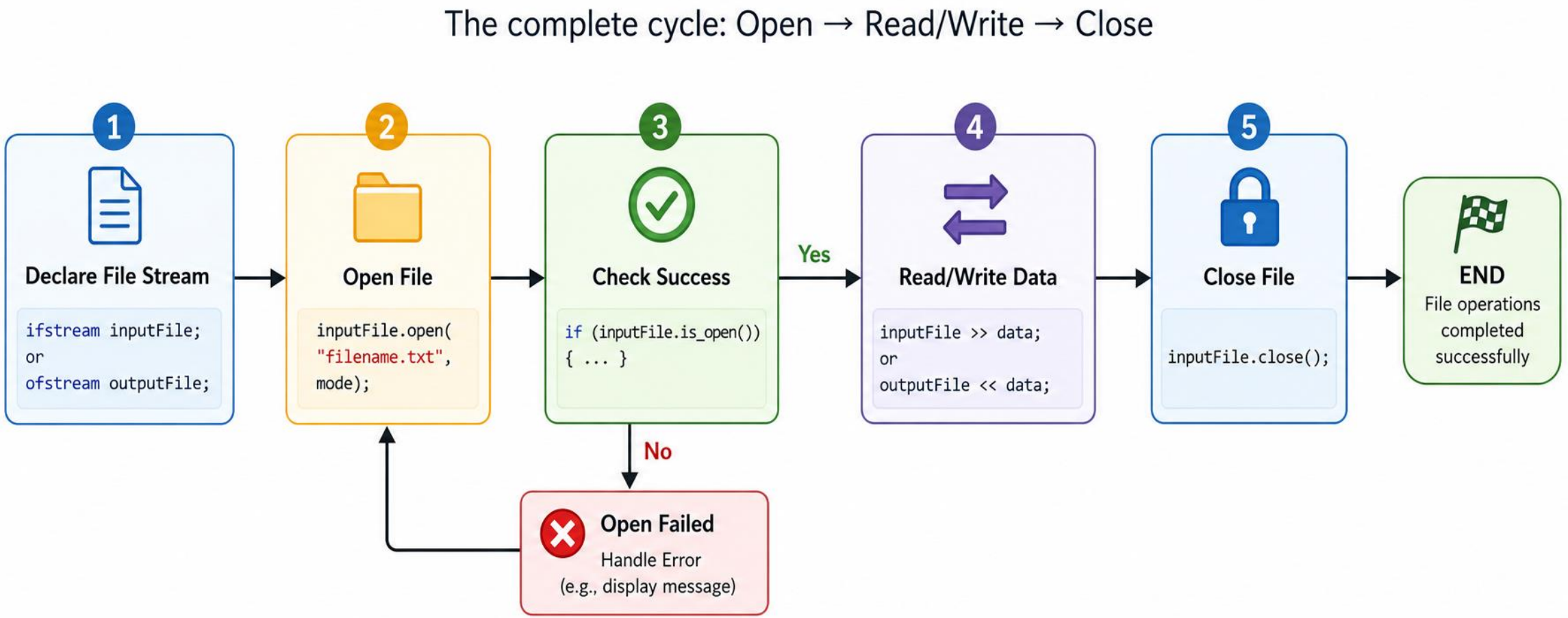
- File operations include: create, open, read, write, append, and close.
- Each operation must be performed in sequence with proper error checking.

Error Handling

Always check is_open() after opening, and verify read/write success before closing to prevent data loss.

File Operations – Lifecycle Flowchart

The complete cycle: Open → Read/Write → Close



File Streams in C++

Three main classes for file I/O — choose based on your need:

ifstream

Input File Stream

Read data FROM a file into your program

```
ifstream inFile("data.txt");
```

ofstream

Output File Stream

Write data FROM your program TO a file

```
ofstream outFile("data.txt");
```

fstream

File Stream (Both)

Read AND write — full bidirectional access

```
fstream file("data.txt", ios::in | ios::out);
```

File Stream Comparison

Feature	ifstream	ofstream	fstream
Read Data	✓ Yes	✗ No	✓ Yes
Write Data	✗ No	✓ Yes	✓ Yes
Header File	#include <fstream>	#include <fstream>	#include <fstream>
Common Use	Load from file	Save to file	Full access
Default Mode	ios::in	ios::out	ios::in | ios::out

Opening and Closing Files

Constructor vs open()

Method 1: Constructor: ifstream file("name.txt"). open():

Method 2: file.open("name.txt").

Critical Rules

Rule #1: Always check is_open()

- File might not exist or permissions blocked.
- Check before reading/writing.

Rule #2: Close files explicitly

- Frees OS resources.
- Destructor closes too, but explicit is safer.

```
1  // Open file for reading
2  ifstream inputFile("data.txt");
3
4  // Alternative: open method
5  ofstream outputFile;
6  outputFile.open("results.txt");
7
8  // Always check if file opened
9  if (!inputFile.is_open()) {
10   cout << "Error: Could not open file";
11   return;
12 }
13
14 // Use the file
15 inputFile >> data;
16
17 // Always close when done
18 inputFile.close();
```

File Opening Modes

Modes control how the file is opened — READ, WRITE, APPEND, etc.:

`ios::in`

Open for reading. File must exist. (Default for ifstream)

`ios::out`

Open for writing. Creates file if not exists; truncates if exists. (Default for ofstream)

`ios::app`

Append mode. Writes at end of file without truncating.

`ios::binary`

Open in binary mode (not text). Used for non-text files.

`ios::ate`

At end — position pointer starts at file end.

`ios::trunc`

Truncate — erase contents if file exists. (Default with ios::out)

Combining Modes

Use the pipe operator | to combine multiple modes.

```
file.open("data.txt", ios::in | ios::binary);
```

Read a binary file

```
file.open("log.txt", ios::out | ios::app);
```

Write/append to a log file

```
file.open("data.dat", ios::in | ios::out);
```

Read and write to same file

```
fstream file("mixed.txt", ios::in | ios::out | ios::app);
```

Read, write, append all modes

Writing to Text Files – Example Program

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ofstream outFile("students.txt");

  if (!outFile.is_open()) {
    cout << "Error!";
    return 1;
  }

  outFile << "Alice 95 CS101" << endl;
  outFile << "Bob 87 MATH201" << endl;
  outFile.close();
  return 0;
}
```

File Created: students.txt

Contents:

```
Alice 95 CS101
Bob 87 MATH201
```

Writing to Files – Append Mode

Problem with ios::out:

Every time you open with ios::out, the file is erased. Run the program twice → you only have the data from the second run.

Solution — ios::app:

Opens the file and moves the write pointer to the end. All new writes are added after existing content.

When to use append:

- Log files (recording events over time)
- Audit trails
- Collecting data across multiple runs

C++ Code

```
// WRONG: Erases previous data
ofstream f("log.txt", ios::out);
f << "Entry at run 2\n";
f.close();

// RIGHT: Appends to existing data
ofstream f("log.txt", ios::app);
f << "Entry at run 2\n";
f.close();
```

Before/After comparison:

```
Before (after run 1):     After (with ios::app):
Entry at run 1            Entry at run 1
                          Entry at run 2
```

Practice Exercise 1 – Writing Student Data

Exercise:

Write a program that prompts for student name, ID, and GPA, then saves them to a file called grades.txt. Open in append mode so multiple students can be added.

```cpp
int main() {
  string name, id;
  double gpa;

  ofstream file("grades.txt", ios::app);
  cin >> name >> id >> gpa;
  file << name << " " << id << " " << gpa << endl;
  file.close();
}
```

input.txt

```
Abebe ASR/1234/18 3.85
```

output.txt

```
Abebe ASR/1234/18 3.85
```

Reading from Text Files – Example Program

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream inFile("data.txt");
  string name;
  int score;

  while (inFile >> name >> score) {
    cout << name << ": " << score << endl;
  }
  inFile.close();
  return 0;
}
```

Sample File (data.txt):

```
Alice 95
Bob 87
Charlie 92
```

Reading from Files – Using while Loop

The while (file >> variable) pattern is the standard way to read multiple records from a file. It automatically detects EOF.

```
ifstream file("data.txt");
int count = 0;
string line;

while (file >> line) {
  count++;
  cout << "Line " << count << ": " << line << endl;
}

cout << "Total lines: " << count << endl;
```

- while (file >> variable) reads until EOF automatically
- Returns false when no more data can be read
- Works with any data type: int, double, string, char
- Loop terminates gracefully — no crash

Reading from Files – Reading Full Lines

When data contains spaces, use getline() instead of >> operator:

Using >> Operator

Note: Stops at whitespace (space, tab, newline)

```
file >> word; // Reads "Alice" only, not "Alice Smith"
```

Using getline()

Note: Reads entire line including spaces

```
getline(file, line); // Reads entire "Alice Smith" line
```

```
1 // Reading full names with spaces
2 string fullName;
3 while (getline(file, fullName)) {
4   cout << "Name: " << fullName << endl;
5 }
```

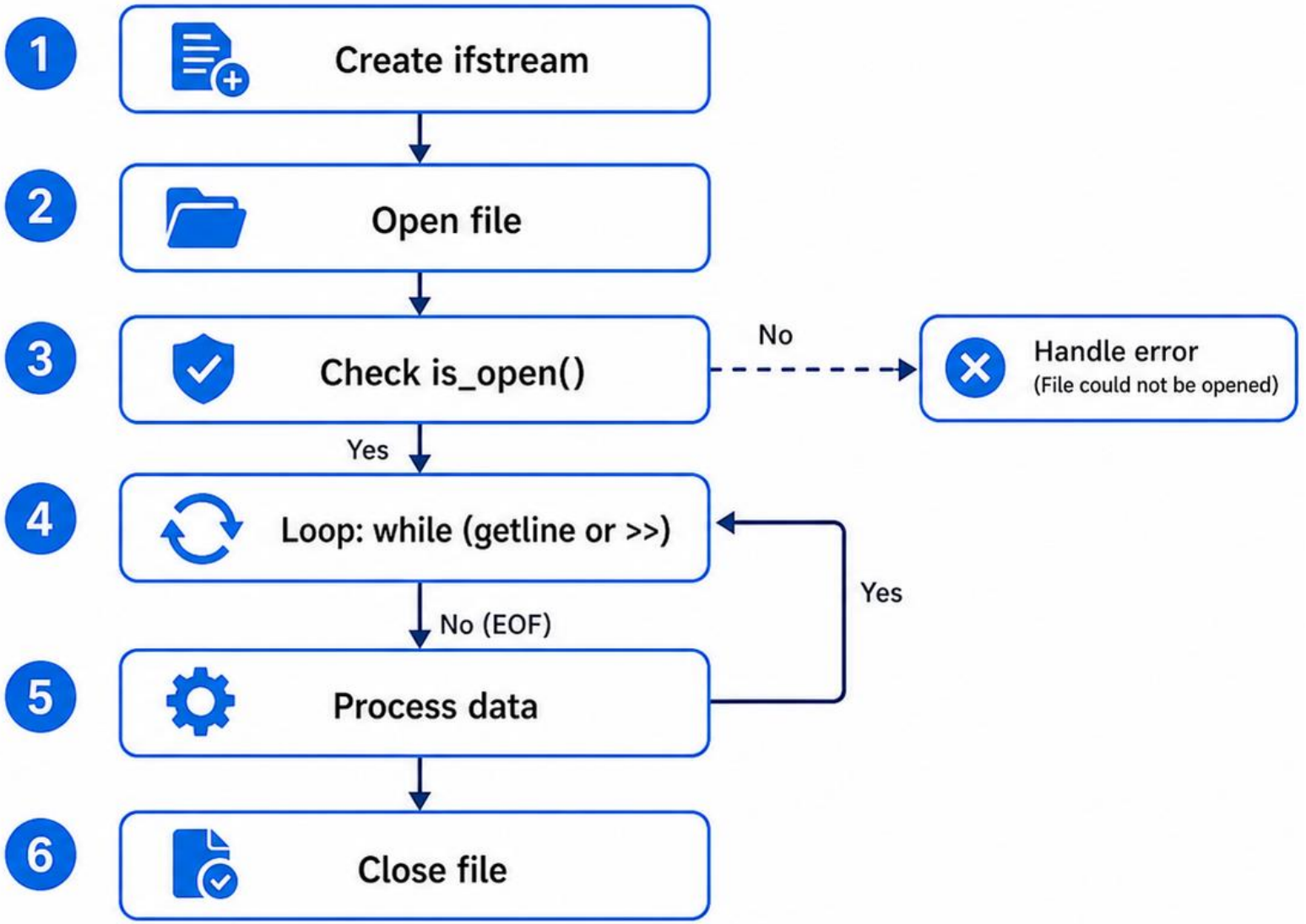
Example: Reading Sentences

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream inFile("notes.txt");
  string sentence;

  while (getline(inFile, sentence)) {
    cout << sentence << endl;
  }
  inFile.close();
  return 0;
}
```

Sample File (notes.txt):

```
This is line 1
Line 2 with spaces
Final line here
```

File Reading Flowchart



Common Reading Problem – Skipped Line

Problem

```
1 string line;
2 file >> line;  // Reads first word only
3 while (getline(file, line)) {
4   // Now reads from MIDDLE of file!
5 }
```

Solution

```
1 string line;
2 // Correct
3 while (getline(file, line)) {
4     // Reads from beginning of file
5  }
```

Debugging Tips for File I/O

Check File Path

Verify the file exists and path is correct. Relative paths are relative to the executable, not source.

Always is_open()

Print error message if file doesn't open. Helps identify permission, path, or disk issues.

Print First Read

After reading, immediately print the value to confirm data was read correctly.

Watch Loop Count

Count iterations to ensure loop terminates. Infinite loops often indicate missed EOF.

Check Format Mismatch

If reading int but file has text, >> fails silently. Validate file format matches code.

Use cerr for Errors

Print errors to cerr (stderr) not cout. Easier to distinguish from normal output.

File Error Handling – Why It Matters

- A missing file doesn't crash your program, it fail silently and reads garbage.
- Check status before using.

Error: File not found	is_open() returns false. Read/write ops fail silently.	Check is_open() before proceeding.
Error: Permission denied	OS blocks access. File remains closed.	Prompt user or try alternate path.
Error: Corrupted data in file	Read ops succeed but data is garbage.	Validate data after reading; skip bad records.
Error: Disk full	Write fails; data is incomplete.	Check fail() after write operations.

Checking File Status

`file.is_open()`

Returns: true if file opened successfully

Use: Check after open(). Essential gate.

`file.fail()`

Returns: true if last operation failed

Use: Check after read/write if unexpected.

`file.eof()`

Returns: true if EOF reached

Use: Use in loops to detect end of file.

`file.good()`

Returns: true if all prior ops succeeded

Use: Alternative to is_open() — stricter check.

Example: Error Handling Program

```
ifstream file("data.txt");
 3 if (!file.is_open()) {
  cerr << "Error: File not found!" << endl;
  return 1;
}
 8 string line;
while (getline(file, line)) {
  if (file.fail()) {
    cerr << "Read error!" << endl;
    break;
  }
  cout << line << endl;
}

file.close();
```

- Check is_open() immediately after open
- Use cerr for error messages, not cout
- Check fail() if operations behave unexpectedly
- Close file in all code paths (or use RAII)

Practical Example – Objective & System Design

Build a Student Grade Management System that reads student records from a file, computes GPA, and writes results to an output file.

Input File

Read student name, ID, and three exam scores from input.txt

Processing

Calculate average of three scores as GPA (weighted or simple mean)

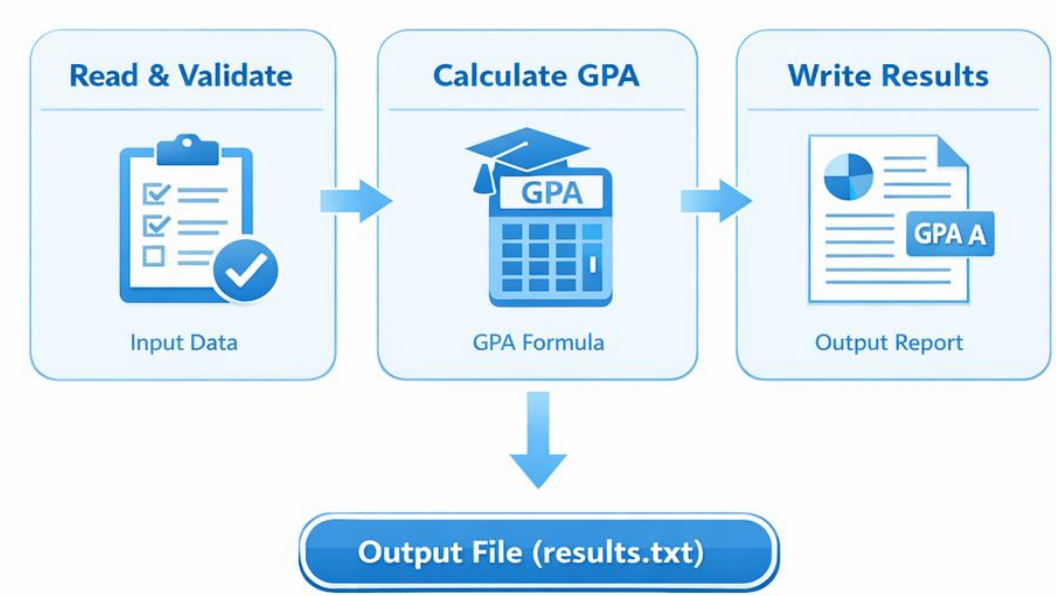
Output File

Write student name, ID, and computed GPA to output.txt

Error Handling

Handle missing file, invalid data, write failures gracefully

System Flow:



Practical Example – Full C++ Code

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream inFile("input.txt");
  ofstream outFile("output.txt");

  if (!inFile.is_open() || !outFile.is_open()) {
    cerr << "File error!" << endl;
    return 1;
  }

  string name, id;
  double s1, s2, s3;

  while (inFile >> name >> id >> s1 >> s2 >> s3) {
    double gpa = (s1 + s2 + s3) / 3.0;
    outFile << name << " " << id << " " << gpa << endl;
  }

  inFile.close();
  outFile.close();
  return 0;
}
```

Input (input.txt):

```
Alice 101 85 90 92
Bob 102 78 82 80
Charlie 103 95 93 97
```

Output (output.txt)

```
Alice 101 89
Bob 102 80
Charlie 103 95
```

Common Pitfalls and Best Practices

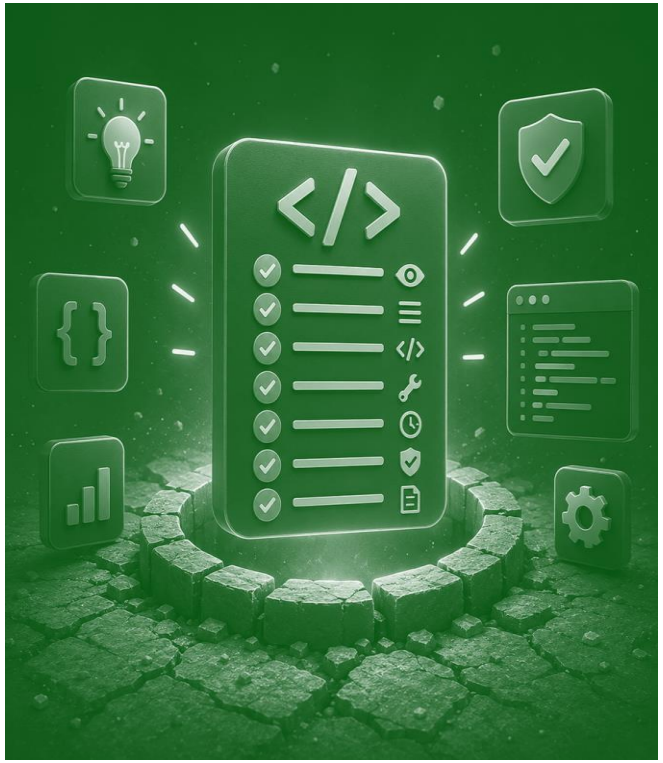
Common Mistakes

- ✕ Not checking **is_open()** and validate file opening
- ✕ Using **>>** with **multi-word data** which stops when encounter space.
- ✕ Mixed use of **>>** and **getline** method.
- ✕ Forgetting to **close files**.
- ✕ Hardcoding file paths like "C:\Data\file.txt" .
- ✕ Fail to validate input

Note: *Avoid these pitfalls — they cause silent failures and data loss*

Cont’d

Best Practices



Check is_open():
- → ALWAYS verify file opened before any operation
- → Exit gracefully if it fails.

Validate data after reading:
- → Check that read values are reasonable.
- → Skip corrupted records.

Close explicitly:
- → Good practice to close files explicitly

Use relative paths:
- → paths relative to executable or current directory.
- → This provide better portability.

Test with invalid files:
- → *Check for missing files, empty files, truncated data.*
- → *Code should handle all gracefully.*

Note: *Apply these **best practices** consistently to write clean and maintainable code*

References

Test Books

- **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