SWEG2102 Fall 2026

Fundamentals of Programming II

Lecture 10

File Management (Part II)

Handling Binary Files

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

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

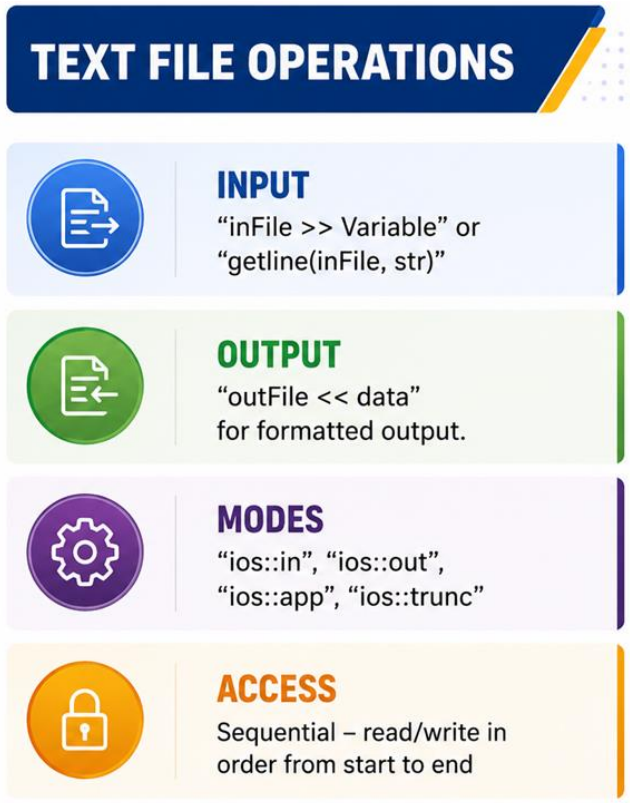
C++ **Standard ISO/IEC 14882**
Programming Language

Recap

Topics covered in previous sessions:

👉 Handling Text Files

- Introduction to File Management
- Types of Files & File Operations
- File Streams in C++
- Opening & Closing Files
- Writing to and Reading from Text Files
- File Operation Error Handling
- Practical Integrated Example



Outline

Topics Covered

01 Binary File Concepts & Representation

02 Advantages & Limitations of Binary Files

03 Opening, Writing and Reading Binary Files

04 **Serialization Concepts & Applications**

05 Random Access Files & File Pointers

06 Integrated Practical Example

07 Common Errors & Best Practices

Learning Objectives

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

- Explain **binary file** concepts and distinguish them from text files.
- Describe the advantages, limitations, and common applications of binary files.
- Apply C++ **file stream** classes to open, close, read from, and write to binary files.
- Explain **serialization** and apply them to store and retrieve structured data.
- Apply random access operations using: seekg(), seekp(), tellg() & tellp()
- Develop small C++ programs that integrate binary file handling concepts.
- Identify common binary file operation errors and apply best practices.

Binary File Concepts

What is a Binary File?

Definition box:

A binary file is a file stores data as raw bytes – exactly as it exists in memory.

Unlike text files, binary are not human-readable and do not use character encoding

👉 Key Characteristics ?

- **Byte-level storage:** Data stored as raw bytes (value 0-255)
- **Exact representation:** Mirror the memory layout precisely
- **Non-text format:** Cannot be opened or read with Notepad
- **Platform-Specific:** May have endianness difference across system.

Text Vs. Binary Representation

👉 Text Representation

- Character stored as ASCII codes
- ‘1’ = 0x31, ‘2’ = 0x32, ‘3’ = 0x33, ‘4’ = 0x34, ‘5’ = 0x35
- **Total size: 5bytes**
- **Human-readable**

👉 Binary Representation

- Integer stored as 4-byte binary value
- ‘0x00 0x00 0x30 0x39’ = 12345 in memory
- **Total size: 4 bytes**
- Efficient ✅ | Not human-readable ❌

Binary Files Advantages and Disadvantages

👉 Advantages

- ✓ **Smaller File Size** — 30–50% smaller than text files for numeric data
- ✓ **Faster I/O** — No text parsing or conversion needed
- ✓ **Exact Numeric Storage** — Preserves floating-point precision
- ✓ **Efficient for Large Datasets** — Ideal for structured data

👉 Disadvantages

- ✗ **Not Human-Readable** — Cannot view with standard text editors
- ✗ **Platform-Dependent** — May not transfer between OS/architectures
- ✗ **Structure-Sensitive** — Any struct change breaks compatibility
- ✗ **Harder to Debug** — Requires hex editors or specialized tools

Opening
Binary Files

Opening Binary Files in C++

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

int main() {
// Method 1: Using open()
ofstream outFile;
outFile.open("data.bin", ios::out | ios::binary);
if (!outFile.is_open()) return 1;

// Method 2: Constructor (shorter)
ifstream inFile("input.bin", ios::in | ios::binary);
if (!inFile) return 1;
outFile.close();
inFile.close();
return 0;
}
```

key Concepts

- “ios::binary” —Required flag; prevents text transiation (newline conversion)
- **Method 1 (open())** — Create stream object first, then call open()
- **Method 2 (Constructor)** — Pass filename and mode directly in constructor
- **Error checking** — Always verify with is_open() or operator!
- **Close files** — Explicitly call close() to flush buffers

Combining Binary File Modes

Mode Combination	Description	Use Case
ios::in | ios::binary	Read-only binary	Config files, data input
ios::out | ios::binary	Write-only binary (creates/truncates)	Data output, reports
ios::app | ios::binary	Append to end in binary	Log files, adding records
ios::in | ios::out | ios::binary	Read and write binary	Databases, in-place updates
ios::ate | ios::binary	Open at end, allows seeking	File size operations

```
// Most common for update operations:
fstream file("data.bin", ios::in | ios::out |
ios::binary);
if (!file) { cerr << "Failed to open file" << endl; }
```

Binary
I/O Operations

Writing Structures to Binary Data

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

int main() {

ofstream out("data.bin", ios::binary);
if (!out) { cerr << "Error opening file\n"; return 1; }

// Writing an integer (4 bytes)
int value = 42;
out.write(reinterpret_cast<const char*>(&value), sizeof(value));

// Writing a double (8 bytes)
double pi = 3.14159;
out.write((char*)&pi, sizeof(pi));

out.close();
return 0;

}
```

Function Signature & Key Points:

- ‘s’ — Pointer to the data, cast to constchar*
- ‘n’ — Number of bytes to write; use sizeof()
- ‘reinterpret_cast<const char*> ‘— Preferred C++ cast
- (char*) — C-style cast alternative (also acceptable)
- Returns — Reference to the stream (for chaining)
- Always check file opened successfully before writing

Writing Structures to Binary Data

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

struct Student
{
  char name[32]; // Fixed-size: NEVER use std::string
  double gpa;
  int age;
};

int main()
{
  Student s;
  strcpy(s.name, "Ada Lovelace");
  s.gpa = 3.9;
  s.age = 20;
  ofstream out("students.bin", ios::binary);
  if (!out)
  {
    cerr << "Error\n";
    return 1;
  }
  // Write entire struct in one call
  out.write(reinterpret_cast<const char *>(&s), sizeof(s));
  out.close();

  return 0;
}
```

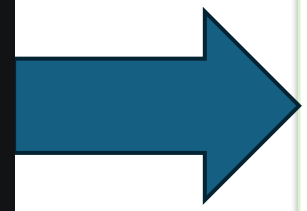
Critical Rules:

1. Use fixed-size struct members (char arrays, int, double, bool)‘n’ — Number of bytes to write; use sizeof()
2. Never use std::string — it contains internal pointers.
3. Open file with ios::binary flag
4. Cast struct address: reinterpret_cast(&s)
5. Size with: sizeof(Student) — writes the entire struct at once

Example Program: Save Student Record

C++ Example

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

struct Student {
  char name[32];
  int id;
  double gpa;
  bool enrolled;
};

int main() {
  Student s;
  strcpy(s.name, "Grace Hopper");
  s.id = 1001;
  s.gpa = 3.8;
  s.enrolled = true;

  ofstream out("students.bin", ios::binary);
  if (!out.is_open()) {
    cerr << "Error: Cannot open file" << endl;
    return 1;
  }

  out.write(reinterpret_cast<const char*>(&s), sizeof(s));
  out.close();
  cout << "Student saved successfully!" << endl;
  return 0;
}
```



Step-by-step breakdown

1. **Define struct** with fixed-size fields (char[], int, double, bool)
2. **Populate** the struct with data using strcpy for char array
3. **Open file** in binary mode with ios::binary
4. **Check** is_open() before proceeding
5. **Write** entire struct using sizeof(Student) bytes
6. **Close** the file to flush and release resources

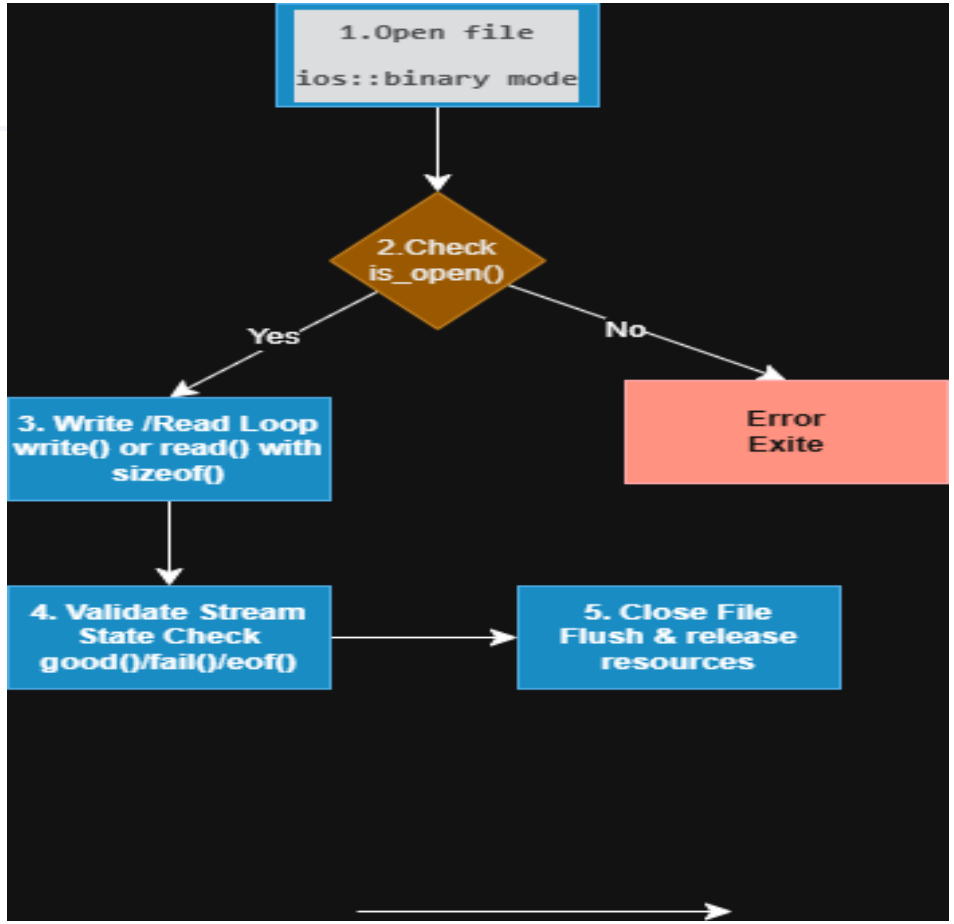
Reading Binary Data

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

struct Student {
  char name[32];
  double gpa;
};

int main() {
  ifstream in("data.bin", ios::binary);
  if (!in) { cerr << "Error\n"; return 1; }

  // Reading a single integer
  int x{};
  in.read(reinterpret_cast<char*>(&x), sizeof(x));
  cout << "Integer: " << x << endl;

  // Reading a struct
  Student s{};
  in.read(reinterpret_cast<char*>(&s), sizeof(s));
  cout << "Name: " << s.name << ", GPA: " << s.gpa << endl;

  // Reading multiple records with a loop
  Student record{};
  while (in.read(reinterpret_cast<char*>(&record), sizeof(record))) {
  cout << record.name << endl;
  }
  return 0;
}
```

key Concepts

- Reads **n bytes** from file into buffer s
- Cast pointer to char* for any data type
- **Loop pattern**: while(in.read(...)) reads all records
- **gcount()** — Returns bytes actually read (use for partial reads)
- Always initialize variables before reading: int x{};

Cont’d



Source: https://claude.ai/

- **Error path:** If is_open() fails → display error → exit
- **Read failure:** Check stream state → clear flags if needed
- **Key operations:** Use reinterpret_cast for type safety
- **Common mistakes:** Missing ios::binary , wrong sizeof() , unchecked stream state

Quick Pause and Checkout

1. Which line correctly writes double d to a binary file?

   A. out << d

   B. out.write((char*)&d, sizeof(d));

   C. write(d);

   D. out << to_string(d);

Answer: B

2. Why must you use a fixed-size char[] instead of std::string in a struct for binary serialization?

   A. std::string is too slow

   B. std::string uses dynamic memory (internal pointers)

   C. std::string is not supported in C++

   D. Char arrays hold more data

Answer: B

Serialization Concepts

</> What is Serialization?

Serialization is converting an object (in memory) into bytes for storage or transmission.

Deserialization reconstructs the object from bytes.

Data Flow:

Object in Memory → **Serialize** → **Byte Stream** → **Deserialize** → **Object Restored**

Real-World Applications:

🎮 **Game Saves**	Store player state, inventory, level progress
⚙ **Config Files**	User preferences and settings efficiently
🗄 **Databases**	Fixed-size records, random access
🌐 **Network Protocols**	Binary message transmission

</> Serialization: Limitations & Example

Critical limitations when serializing to binary files:

❌ **Dynamic Memory**	Never use std::string or std::vector — they contain pointers, not data
❌ **Pointer Serialization**	Never write raw pointers — addresses change between program runs
❌ **Versioning**	Changing struct fields breaks existing files — use version field
❌ **Platform Dependencies**	Endianness and padding vary — test on all target platforms

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

struct Product {
  int id;
  char name[50];
  double price;
  int qty;
};
int main() {
  Product p{1001, "USB-C Cable", 9.99, 120};
  // Serialize (write)
  ofstream out("products.dat", ios::binary);
  out.write((char*)&p, sizeof(p));
  out.close();
  // Deserialize (read back)
  Product q{};
  ifstream in("products.dat", ios::binary);
  in.read((char*)&q, sizeof(q));
  in.close();
}
```

Random Access

</> Sequential vs Random Access

Sequential Access 🐌

- Read from start until target
- Must read every record in order
- Time: O(n)
- Best for: Small files, full scans

Random Access ⚡

- Calculate byte offset → seek
- Jump directly to position
- Time: O(1)
- Best for: Large files, databases

Key formula: byte_offset = record_index × sizeof(RecordType)

Example:

Finding record #500 in file with 72-byte records:

```
Offset = 500 × 72 = 36,000 bytes  →  seekg(36000, ios::beg)  →  read()
[O(1) instant access]
```

</> File Pointers: Get & Put Positions

Get Pointer (Read Position):

- seekg(pos) — Move to absolute byte
- seekg(offset, origin) — Relative move
- tellg() — Query current position

Put Pointer (Write Position):

- seekp(pos) — Move to absolute byte
- seekp(offset, origin) — Relative move
- tellp() — Query current position

Seek Origins (for relative moves):

`ios::beg`	From beginning of file (absolute)
`ios::cur`	From current position (relative)
`ios::end`	From end of file (use negative offset)

Example code:

```
1  file.seekg(100, ios::beg);  // Byte 100
2  streampos pos = file.tellg();      // Query position
3  file.seekp(0, ios::end);   // Move to end
```

Random Access: Jump to Record #5

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

struct Product {
        int id;
        char name[50];
 };

int main() {
      ifstream f("products.bin", ios::binary);
      Product rec{};
      int idx = 5;
     f.seekg(idx * sizeof(Product), ios::beg);
     f.read((char*)&rec, sizeof(rec));
     cout << rec.name << endl;
 }
```

Step-by-Step:

1. *Open file in binary mode*
2. *Calculate offset: idx × sizeof()*
3. *seekg() to that byte position*
4. *read() pulls the struct*
5. *Check stream state*
6. *Display or process data*

</> Updating Specific Records

```
fstream file("products.dat",
             ios::in | ios::out | ios::binary);
Product p{};
int idx = 2;

// 1. Seek & READ
 file.seekg(idx * sizeof(Product), ios::beg);
 file.read((char*)&p, sizeof(p));

// 2. MODIFY field
 p.price = 29.99;

// 3. Seek BACK & WRITE
 file.seekp(idx * sizeof(Product), ios::beg);
 file.write((char*)&p, sizeof(p));
```

The Read-Modify-Write Pattern:

1. Calculate offset
2. seekg() to record
3. read() into memory
4. Modify fields
5. seekp() same position
6. write() back to file

⚠ **IMPORTANT**: After read(), the pointer advances by sizeof(Record) bytes.

➢ You MUST **seekp()** back to the same position before writing, or you'll corrupt the next record.

</> Deleting Records (Logical Deletion)

```
struct Product {
  int id;
  char name[50];
  double price;
  bool deleted = false;  // Flag
};

void markDeleted(fstream& f, int idx) {
  Product p{};
 f.seekg(idx * sizeof(Product), ios::beg);
 f.read((char*)&p, sizeof(p));
 p.deleted = true;
 f.seekp(idx * sizeof(Product), ios::beg);
 f.write((char*)&p, sizeof(p));
}
```

Why Logical Deletion?

- Physical delete = rebuild file (expensive)
- Logical delete = flip flag (instant)
- Preserves record positions
- Skip marked records on display

Best Practice:

- ➢ Use logical deletion for frequent updates.
- ➢ Periodically compact the file during off-peak maintenance by copying all non-deleted records to a new file.
- ➢ This is how production databases handle 'soft deletes'.

Practical Examples

Inventory System Design

Product struct (fixed-size, ~72 bytes):

```
struct Product {
        int id;
        char name[50];
        double price;
        int quantity;
        bool deleted = false;
};
```

System Features:

- ➕ Add products (append)
- 📋 List all (display non-deleted)
- ✏️ Update price (seek & write)
- 🗑️ Mark deleted (logical flag)

Data Flow:

User Input → **Product Struct** → **Binary Write** → **.dat File**

</> Inventory System — Implementation (Part 1)

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

 struct Product { int id; char name[50]; double price; int qty; bool deleted; };

 streamoff offsetOf(size_t idx) {
  return idx * sizeof(Product);
   }

void addProduct(const string& file, Product& p) {
ofstream out(file, ios::app | ios::binary);
  out.write((char*)&p, sizeof(p));
  }

void listAll(const string& file) {
ifstream in(file, ios::binary);
Product p{};
 while (in.read((char*)&p, sizeof(p)))
if (!p.deleted) cout << p.id << " | " << p.name << endl;
  }
```

Cont’d

Inventory System — Implementation (Part 2)

```
    void updatePrice(fstream& file, int idx, double price) {
    Product p{};
    file.seekg(offsetOf(idx), ios::beg);
    file.read((char*)&p, sizeof(p));
    p.price = price;
     file.seekp(offsetOf(idx), ios::beg);
     file.write((char*)&p, sizeof(p));
      }

    void markDeleted(fstream& file, int idx) {
     Product p{};
     file.seekg(offsetOf(idx), ios::beg);
     file.read((char*)&p, sizeof(p));
     p.deleted = true;
     file.seekp(offsetOf(idx), ios::beg);
      file.write((char*)&p, sizeof(p));
     }

    int main() {
       Product p1{1001, "USB Cable", 9.99, 120, false};
     addProduct("products.dat", p1);
      listAll("products.dat");
      return 0;
     }
```

Common Errors & Best Practices

Common Binary File Handling Errors

1. Forgetting ios::binary	Always use: ios::out | ios::binary or ios::in | ios::binary
2. Using std::string in structs	Replace with: char name[50] fixed-size arrays
3. Wrong sizeof() / offset math	Validate: offset = index × sizeof(Record); test small files
4. Ignoring stream state	Always: if (!file) { error; return 1; }
5. Writing pointers to file	Serialize data not pointers: write((char*)&data, sizeof(data))
6. Platform dependencies	Document struct version; test on all target platforms

</> Best Practices

Best Practices ✅

- Use fixed-size POD structs
- Centralize sizeof/offset logic
- Validate ALL I/O operations
- Document struct layout
- Use logical deletion
- Test with small files first

Key Takeaways 📌

1. Binary mirrors memory layout
2. ios::binary is MANDATORY
3. Serialize fixed-size only
4. offset = idx × sizeof()
5. seekg for read, seekp for write
6. Check stream state always

End-of-Lecture Practice:

1. Write a program storing 5 Employee structs; read back employee #3
2. Add updateSalary() using random access to modify a record
3. Implement logical deletion + compact() to remove deleted records

References

Text 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

- **Object-Oriented Programming in C++ (4th Edition)** by Robert Lafore, Sams Publishing, 2022.
- **Principles of Object-Oriented Programming**, by Stephen W. & Dung N., OpenStax CNX, 2022.

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, modify and compile 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