5.2 Binary File Operations

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

5.2 Binary File Operations
Unit 5: File Handling and Exception Handling
Object Oriented Programming with C++

5.2 Binary File Operations

- Binary vs. Text Files
- Reading and Writing Binary Data
- Random Access and Updating Records
- File Error Handling and Stream States

5.2 Binary File Operations
- Binary vs. Text Files
- Reading and Writing Binary Data
- Random Access and Updating Records
- File Error Handling and Stream States

Welcome back to Object Oriented Programming with C++. Today, we are diving into Lecture 38, focusing on Binary File Operations. We'll explore how binary files differ from text files, how to read, write, and importantly, how to update records in place. We will also cover how to handle errors robustly during these file operations.

Text vs. Binary Files

- Text Files: Store data as human-readable ASCII/Unicode characters.
- Text Files: Character translations occur (e.g., '\n' to CRLF on Windows).
- Binary Files: Store data in the exact format it is represented in computer memory.
- Binary Files: No character translations occur. Raw bytes are written and read.

Text vs. Binary Files
- Text Files: Store data as human-readable ASCII/Unicode characters.
- Text Files: Character translations occur (e.g., '\n' to CRLF on Windows).
- Binary Files: Store data in the exact format it is represented in computer memory.
- Binary Files: No character translations occur. Raw bytes are written and read.

Let's clarify the difference. When you write the integer 12345 to a text file, it's saved as 5 distinct characters: '1', '2', '3', '4', '5'. But in a binary file, if it's a 4-byte integer, it's saved exactly as the 4 bytes in RAM. Binary files don't care about human readability; they care about memory layout.

Why Use Binary Files?

- Efficiency: Faster read/write speeds since no formatting or translation is needed.
- Storage Capacity: Often takes up less disk space for numerical or complex data.
- Security/Obscurity: Not easily readable in standard text editors without knowing the exact structure.
- Direct Mapping: Can directly write or read entire C++ structures or objects in a single operation.

Why Use Binary Files?
- Efficiency: Faster read/write speeds since no formatting or translation is needed.
- Storage Capacity: Often takes up less disk space for numerical or complex data.
- Security/Obscurity: Not easily readable in standard text editors without knowing the exact structure.
- Direct Mapping: Can directly write or read entire C++ structures or objects in a single operation.

Why do we bother with binary? Speed and convenience. Imagine writing an array of 1,000 objects. With text files, you have to format each field. With binary, you just dump the block of memory directly to the disk. It's incredibly fast.

Opening Files in Binary Mode

- Use the ios::binary flag when opening a file.
- Syntax: fstream file("data.dat", ios::in — ios::out — ios::binary);
- Must be combined with standard directional flags (ios::in for read, ios::out for write).
- The .dat extension is commonly used, but the extension does not dictate the file type—the open mode does.

Opening Files in Binary Mode
- Use the ios::binary flag when opening a file.
- Syntax: fstream file("data.dat", ios::in — ios::out — ios::binary);
- Must be combined with standard directional flags (ios::in for read, ios::out for write).
- The .dat extension is commonly used, but the extension does not dictate the file type—the open mode does.

To tell C++ to skip character translations, we MUST include the ios::binary flag when opening the stream. Notice that we often combine flags using the bitwise OR operator. The file extension .dat or .bin is just a convention; C++ only cares about the ios::binary flag.

Writing Data: The write() Function

- Standard stream insertion (¡¡) is for text files.
- Binary files use the write() method of ostream.
- Syntax: ostream& write(const char* s, streamsize n);
- s: A pointer to the character array (memory address).
- n: The number of bytes to write.

Writing Data: The write() Function
- Standard stream insertion (¡¡) is for text files.
- Binary files use the write() method of ostream.
- Syntax: ostream& write(const char* s, streamsize n);
- s: A pointer to the character array (memory address).
- n: The number of bytes to write.

We cannot use cout-style formatting operators (¡¡) for binary files because they format data into text. Instead, we use the write() function. It takes two arguments: where in memory to start copying from, and how many bytes to copy.

Example: Writing a Record

- struct Student { int id; char name[50]; float gpa; };
- Student s1 = {101, "Alice", 3.8};
- ofstream outFile("students.dat", ios::binary);
- outFile.write((char*)&s1, sizeof(Student));
- outFile.close();

Example: Writing a Record
- struct Student { int id; char name[50]; float gpa; };
- Student s1 = {101, "Alice", 3.8};
- ofstream outFile("students.dat", ios::binary);
- outFile.write((char*)&s1, sizeof(Student));
- outFile.close();

Look at this code. We have a Student struct. To write it, we take the address of s1 (&s1), and we MUST cast it to a char pointer (char*) because write() expects a byte stream. We use sizeof(Student) to ensure we write exactly the right number of bytes.

Reading Data: The read() Function

- Standard stream extraction (¿¿) is for text files.
- Binary files use the read() method of istream.
- Syntax: istream& read(char* s, streamsize n);
- s: Address in memory to store the incoming data.
- n: The number of bytes to extract from the file.

Reading Data: The read() Function
- Standard stream extraction (¿¿) is for text files.
- Binary files use the read() method of istream.
- Syntax: istream& read(char* s, streamsize n);
- s: Address in memory to store the incoming data.
- n: The number of bytes to extract from the file.

Reading is the exact inverse of writing. We use the read() function. We give it the address in memory where the data should be loaded, and the number of bytes to pull from the file.

Example: Reading a Record

- Student s2;
- ifstream inFile("students.dat", ios::binary);
- inFile.read((char*)&s2, sizeof(Student));
- inFile.close();
- // Now s2 contains {101, "Alice", 3.8}

Example: Reading a Record
- Student s2;
- ifstream inFile("students.dat", ios::binary);
- inFile.read((char*)&s2, sizeof(Student));
- inFile.close();
- // Now s2 contains {101, "Alice", 3.8}

Here we read the data back. We declare an empty Student object s2. We call read(), passing the casted address of s2 and the size. The raw bytes from the file are pushed directly into s2, instantly populating all its fields.

Random Access and File Pointers

- Sequential Access: Reading data from the beginning to the end, in order.
- Random Access: Jumping directly to any specific location in a file.
- C++ maintains two internal file pointers:
- 1. get pointer: Points to the next byte to be read.
- 2. put pointer: Points to the next byte to be written.

Random Access and File Pointers
- Sequential Access: Reading data from the beginning to the end, in order.
- Random Access: Jumping directly to any specific location in a file.
- C++ maintains two internal file pointers:
- 1. get pointer: Points to the next byte to be read.
- 2. put pointer: Points to the next byte to be written.

Up until now, we've read from the start to the end. But what if we have a file with 10,000 students and only want to update student #5,000? We don't want to read the first 4,999. This is where Random Access comes in, managed by internal pointers.

Navigating Pointers: seekg() and seekp()

- seekg(offset, direction): Moves the get (read) pointer.
- seekp(offset, direction): Moves the put (write) pointer.
- offset: Number of bytes to move.
- direction: Where to start the move from.
- - ios::beg (beginning of file)
- - ios::cur (current position)
- - ios::end (end of file)

Navigating Pointers: seekg() and seekp()
seekg(offset, direction): Moves the get (read) pointer.
seekp(offset, direction): Moves the put (write) pointer.
offset: Number of bytes to move.
direction: Where to start the move from.
- ios::beg (beginning of file)
- ios::cur (current position)
- ios::end (end of file)

To move these pointers, we use seekg for 'get' and seekp for 'put'. Think of 'g' for get/read, and 'p' for put/write. We specify an offset in bytes, and a starting anchor—either the beginning, current position, or end of the file.

Locating Pointers: tellg() and tellp()

- tellg(): Returns the current byte position of the get pointer.
- tellp(): Returns the current byte position of the put pointer.
- Example trick to find file size:
- file.seekg(0, ios::end);
- long size = file.tellg();

Locating Pointers: tellg() and tellp()
tellg(): Returns the current byte position of the get pointer.
tellp(): Returns the current byte position of the put pointer.
Example trick to find file size:
file.seekg(0, ios::end);
long size = file.tellg();

If you are lost in a file, tellg() and tellp() tell you exactly where you are, returning the byte index. A classic trick to find a file's size is to seek to the end (offset 0 from ios::end) and then call tellg() to get the byte count.

The Process of Updating Records

- To update a record in place, you must open the file in both input and output mode: ios::in — ios::out.
- 1. Open file and read records sequentially to find the target.
- 2. Once found, modify the record data in memory.
- 3. Calculate the starting byte position of the record in the file.
- 4. Move the put pointer backwards to that position using seekp().
- 5. write() the modified record back to the file.

The Process of Updating Records
- To update a record in place, you must open the file in both input and output mode: ios::in — ios::out.
- 1. Open file and read records sequentially to find the target.
- 2. Once found, modify the record data in memory.
- 3. Calculate the starting byte position of the record in the file.
- 4. Move the put pointer backwards to that position using seekp().
- 5. write() the modified record back to the file.

Updating is a multi-step process. Crucially, you must open the file for both reading and writing simultaneously. When you find the record, your file pointer has already moved PAST it. So, to overwrite it, you must move the pointer backwards by the size of the record.

Example: Updating a Record

- // Assuming file is opened with ios::in — ios::out — ios::binary
- file.read((char*)&s, sizeof(Student));
- if(s.id == targetId) {
- s.gpa = 4.0; // Modify in memory
- long pos = file.tellg() - sizeof(Student); // Calculate rewind
- file.seekp(pos, ios::beg); // Rewind pointer
- file.write((char*)&s, sizeof(Student)); // Overwrite
- }

Example: Updating a Record
- // Assuming file is opened with ios::in — ios::out — ios::binary
- file.read((char*)&s, sizeof(Student));
- if(s.id == targetId) {
- s.gpa = 4.0; // Modify in memory
- long pos = file.tellg() - sizeof(Student); // Calculate rewind
- file.seekp(pos, ios::beg); // Rewind pointer
- file.write((char*)&s, sizeof(Student)); // Overwrite
- }

Let's look at the math. After read() finishes, the get pointer is at the next record. We take tellg(), subtract the size of one Student to find where our target record started, use seekp() to go there, and then write(). The record is updated seamlessly in place.

Error Handling in Files

- File operations are prone to external failures.
- Common issues: File not found, lack of read/write permissions, disk full, corrupted media.
- C++ stream objects maintain internal state flags to monitor stream health.
- We should never assume a file operation succeeded without checking.

Error Handling in Files
- File operations are prone to external failures.
- Common issues: File not found, lack of read/write permissions, disk full, corrupted media.
- C++ stream objects maintain internal state flags to monitor stream health.
- We should never assume a file operation succeeded without checking.

Moving on to error handling. Unlike variables in RAM, files are on the disk. They can be deleted by another program, permissions can change, disks can fill up. Robust software must account for this.

Stream State Flags

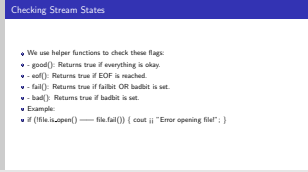
- C++ provides four bit-flags to represent stream status:
- 1. goodbit: Everything is perfectly fine (value 0).
- 2. eofbit: The End-Of-File has been reached during a read.
- 3. failbit: A logical error occurred (e.g., trying to read a string into an int variable).
- 4. badbit: A severe physical error occurred (e.g., hardware failure, corrupted stream).

Stream State Flags
C++ provides four bit-flags to represent stream status:
1. goodbit: Everything is perfectly fine (value 0).
2. eofbit: The End-Of-File has been reached during a read.
3. failbit: A logical error occurred (e.g., trying to read a string into an int variable).
4. badbit: A severe physical error occurred (e.g., hardware failure, corrupted stream).

Behind the scenes, every file stream has a set of flags. The goodbit means all is well. The eofbit is very common; it triggers when you hit the end. The failbit means you made a mistake like data type mismatch. The badbit is the worst—usually means the disk failed or the stream is dead.

Checking Stream States

- We use helper functions to check these flags:
- - good(): Returns true if everything is okay.
- - eof(): Returns true if EOF is reached.
- - fail(): Returns true if failbit OR badbit is set.
- - bad(): Returns true if badbit is set.
- Example:
- if (!file.is_open() —— file.fail()) { cout ¡¡ "Error opening file!"; }



Instead of checking the bits directly, C++ provides boolean functions. You should frequently use .fail() or just check the stream in an if statement (e.g., `if(file)`) which implicitly checks for failure. .is_open() is also crucial when initially opening a file.

Clearing Error States: clear()

- Once a stream enters an error state (failbit, eofbit, badbit), it "locks up".
- Further read/write operations will be completely ignored.
- Use the clear() function to reset all state flags back to goodbit.
- Example:
- file.clear(); // Resets the stream state
- file.seekg(0, ios::beg); // Safe to move pointers again

Clearing Error States: clear()
- Once a stream enters an error state (failbit, eofbit, badbit), it "locks up".
- Further read/write operations will be completely ignored.
- Use the clear() function to reset all state flags back to goodbit.
- Example:
- file.clear(); // Resets the stream state
- file.seekg(0, ios::beg); // Safe to move pointers again

Here is a massive pitfall for beginners. If you read until the end of a file, the eofbit is set. If you then try to seek back to the beginning and read again, it will fail silently! The stream is locked. You MUST call file.clear() to clear the error flags before you can use the stream again.

Summary & Q&A

- Binary files map data directly between memory and disk.
- Use write() and read() for binary I/O, utilizing typecasting and sizeof().
- seekp/seekg and tellp/tellg allow random access and in-place record updates.
- Robust programs utilize stream state functions to handle errors defensively.

Summary & Q&A
- Binary files map data directly between memory and disk.
- Use write() and read() for binary I/O, utilizing typecasting and sizeof().
- seekp/seekg and tellp/tellg allow random access and in-place record updates.
- Robust programs utilize stream state functions to handle errors defensively.

To summarize, binary files are powerful tools for storing structured data efficiently. By mastering pointers and stream states, you can create complex, robust file databases in C++. Are there any questions?