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++

Lecture 42: Binary File Operations

- Unit 5: File Handling and Exception Handling
- Topic: 5.2 Binary File Operations
- Subtopics: Updating Records, Error Handling in Files

Lecture 42: Binary File Operations
- Unit 5: File Handling and Exception Handling
- Topic: 5.2 Binary File Operations
- Subtopics: Updating Records, Error Handling in Files

Welcome to Lecture 42. Today we are diving deeper into binary file operations. While we've previously learned how to write new data to a binary file and read it back, today's focus is on real-world application: updating existing records in place and managing errors gracefully. This is critical for building robust applications like databases or inventory systems.

Why Update Records in Binary Files?

- Efficiency: Modifying a single record without rewriting the entire file.
- Fixed-Length Structures: Binary files allow storing structs/classes of fixed size.
- Random Access: Ability to jump directly to a specific record using byte offsets.
- Real-World Analogy: Like changing a single entry in a physical ledger book instead of copying the whole book.

Why Update Records in Binary Files?
- Efficiency: Modifying a single record without rewriting the entire file.
- Fixed-Length Structures: Binary files allow storing structs/classes of fixed size.
- Random Access: Ability to jump directly to a specific record using byte offsets.
- Real-World Analogy: Like changing a single entry in a physical ledger book instead of copying the whole book.

Imagine an employee database with 10,000 records. If an employee gets a promotion, rewriting the entire file just to update one salary field is extremely inefficient. Because binary files allow us to store structs in fixed-size memory blocks, we can calculate the exact byte location of the employee's record, jump straight to it, and overwrite it. This is known as Random Access.

The Prerequisites for Updating

- File Mode: Must open the file for BOTH reading and writing.
- Syntax: fstream file("data.dat", ios::in — ios::out — ios::binary);
- fstream Object: We use fstream instead of ifstream or ofstream to have bidirectional capabilities.
- Data Structure: A well-defined struct or class with a fixed sizeof().

The Prerequisites for Updating
- File Mode: Must open the file for BOTH reading and writing.
- Syntax: fstream file("data.dat", ios::in — ios::out — ios::binary);
- fstream Object: We use fstream instead of ifstream or ofstream to have bidirectional capabilities.
- Data Structure: A well-defined struct or class with a fixed sizeof().

To update a file, you first have to read the record, modify it in memory, and write it back. This means your file stream must support both input and output operations. We achieve this by instantiating an fstream object and combining the ios::in, ios::out, and ios::binary flags using the bitwise OR operator. Also, remember that your data structure should not contain pointers, like std::string, but rather fixed-size arrays for text.

Navigating Files: File Pointers

- C++ file streams maintain two internal pointers:
- 1. get pointer: Indicates the next byte to be read (used with input).
- 2. put pointer: Indicates the next byte to be written (used with output).
- In an fstream object, moving one pointer generally moves the other, as they often track the same position.

Navigating Files: File Pointers
- C++ file streams maintain two internal pointers:
- 1. get pointer: Indicates the next byte to be read (used with input).
- 2. put pointer: Indicates the next byte to be written (used with output).
- In an fstream object, moving one pointer generally moves the other, as they often track the same position.

Behind the scenes, the C++ stream keeps track of where it currently is in the file. Think of it like a cursor in a word processor. We have the 'get' pointer for reading and the 'put' pointer for writing. When we read a 50-byte struct, the get pointer automatically advances by 50 bytes. Understanding these pointers is the foundation of random access.

Random Access Methods: seekg() and seekp()

- seekg(offset, direction): Moves the get pointer.
- seekp(offset, direction): Moves the put pointer.
- offset: Number of bytes to move (can be positive or negative).
- direction (Seekdir constants):
- - ios::beg : Beginning of the file
- - ios::cur : Current position of the pointer
- - ios::end : End of the file

Random Access Methods: seekg() and seekp()
- seekg(offset, direction): Moves the get pointer.
- seekp(offset, direction): Moves the put pointer.
- offset: Number of bytes to move (can be positive or negative).
- direction (Seekdir constants):
- - ios::beg : Beginning of the file
- - ios::cur : Current position of the pointer
- - ios::end : End of the file

To jump to a specific location, C++ provides seekg (seek get) and seekp (seek put). They take two arguments: the offset in bytes, and the starting reference point. For example, seekg(100, ios::beg) moves the reading cursor exactly 100 bytes from the start of the file. seekp(-20, ios::end) moves the writing cursor 20 bytes backward from the end.

Determining Position: tellg() and tellp()

- tellg(): Returns the current position of the get pointer.
- tellp(): Returns the current position of the put pointer.
- Return Type: std::streampos (an integer type representing bytes).
- Use Case: Finding out the total size of a file, or remembering a location to return to later.

Determining Position: tellg() and tellp()
- tellg(): Returns the current position of the get pointer.
- tellp(): Returns the current position of the put pointer.
- Return Type: std::streampos (an integer type representing bytes).
- Use Case: Finding out the total size of a file, or remembering a location to return to later.

If seek is for moving the cursor, tell is for asking 'where am I?'. tellg() returns an integer representing the byte offset of the get pointer from the beginning of the file. A common trick to find the total size of a file is to seek to the end (ios::end), and then call tellg().

Calculating the Record Offset

- Formula to find the byte offset of the Nth record:
- Offset = (N - 1) * sizeof(RecordType)
- Example: Finding the 5th record of type 'Employee'
- int position = (5 - 1) * sizeof(Employee);
- file.seekg(position, ios::beg);

Calculating the Record Offset
- Formula to find the byte offset of the Nth record:
- Offset = (N - 1) * sizeof(RecordType)
- Example: Finding the 5th record of type 'Employee'
- int position = (5 - 1) * sizeof(Employee);
- file.seekg(position, ios::beg);

Because binary files pack data structures back-to-back, calculating the location of a specific record is simple math. If each record is 100 bytes, the first record starts at byte 0, the second at byte 100, the third at byte 200, and so on. We subtract 1 from N because file offsets are zero-indexed, just like arrays in C++.

Step-by-Step: Updating a Record (Part 1)

- struct Employee { int id; double salary; };
- // 1. Open the file for read and write
- fstream file("emp.dat", ios::in — ios::out — ios::binary);
- // 2. Calculate position of the 3rd record
- int pos = (3 - 1) * sizeof(Employee);
- // 3. Move the get pointer
- file.seekg(pos, ios::beg);

Step-by-Step: Updating a Record (Part 1)
- struct Employee { int id; double salary; };
- // 1. Open the file for read and write
- fstream file("emp.dat", ios::in — ios::out — ios::binary);
- // 2. Calculate position of the 3rd record
- int pos = (3 - 1) * sizeof(Employee);
- // 3. Move the get pointer
- file.seekg(pos, ios::beg);

Let's look at the code for updating the 3rd employee's record. First, we define a simple struct. We open the file using fstream with all three necessary flags. We calculate the byte offset by taking (3 - 1) multiplied by the size of our struct. Then, we use seekg to move our read cursor to that exact byte.

Step-by-Step: Updating a Record (Part 2)

- Employee emp;
- // 4. Read the existing record
- file.read(reinterpret_cast¡char*¿(&emp), sizeof(Employee));
- // 5. Modify the record in memory
- emp.salary += 5000.0;
- // 6. Move the put pointer back to the start of the record
- file.seekp(pos, ios::beg);
- // 7. Write the updated record
- file.write(reinterpret_cast¡char*¿(&emp), sizeof(Employee));

Step-by-Step: Updating a Record (Part 2)
- Employee emp;
- // 4. Read the existing record
- file.read(reinterpret_cast¡char*¿(&emp), sizeof(Employee));
- // 5. Modify the record in memory
- emp.salary += 5000.0;
- // 6. Move the put pointer back to the start of the record
- file.seekp(pos, ios::beg);
- // 7. Write the updated record
- file.write(reinterpret_cast¡char*¿(&emp), sizeof(Employee));

Once the pointer is in place, we read the record into a local object 'emp'. Crucially, reading the data advances the pointer past the record. So, after we update the salary in memory, we must use seekp to move the put pointer back to the original starting position 'pos' before we call write(). Otherwise, we would overwrite the 4th record instead of the 3rd!

Introduction to File Error Handling

- File operations are prone to external failures:
- - File doesn't exist or is locked by another process.
- - Disk is full.
- - Permission denied.
- - Attempting to read past the End Of File (EOF).
- C++ provides built-in mechanisms to detect these failures.

Introduction to File Error Handling
- File operations are prone to external failures:
- - File doesn't exist or is locked by another process.
- - Disk is full.
- - Permission denied.
- - Attempting to read past the End Of File (EOF).
- C++ provides built-in mechanisms to detect these failures.

Transitioning to our second subtopic: Error Handling. When dealing with files, you are interacting with the operating system and the hardware. This means many things can go wrong that are out of your program's control. A robust C++ program never assumes a file operation succeeded. It always checks.

Stream State Flags

- Every stream object maintains internal state flags:
- 1. goodbit: Everything is fine (value is 0).
- 2. eofbit: Reached End-Of-File on an input operation.
- 3. failbit: Logical error on I/O (e.g., type mismatch, file not found).
- 4. badbit: Fatal error (e.g., physical disk failure, stream corruption).

Stream State Flags
- Every stream object maintains internal state flags:
- 1. goodbit: Everything is fine (value is 0).
- 2. eofbit: Reached End-Of-File on an input operation.
- 3. failbit: Logical error on I/O (e.g., type mismatch, file not found).
- 4. badbit: Fatal error (e.g., physical disk failure, stream corruption).

C++ stream objects have a set of boolean flags that indicate their health. If the goodbit is true, the last operation worked perfectly. If eofbit is true, you hit the end of the file. failbit means a recoverable logic error happened—like trying to open a file that doesn't exist. badbit means something catastrophic happened to the stream itself.

Checking Stream States

- Functions to check flags:
- - file.good() : Returns true if no error flags are set.
- - file.eof() : Returns true if eofbit is set.
- - file.fail() : Returns true if failbit OR badbit is set.
- - file.bad() : Returns true if badbit is set.
- Implicit Check: if (file) or if (!file) evaluates the stream's state.

Checking Stream States
- Functions to check flags:
- - file.good() : Returns true if no error flags are set.
- - file.eof() : Returns true if eofbit is set.
- - file.fail() : Returns true if failbit OR badbit is set.
- - file.bad() : Returns true if badbit is set.
- Implicit Check: if (file) or if (!file) evaluates the stream's state.

Instead of reading bits directly, C++ provides handy member functions. Calling file.fail() is the most common way to check if an operation like opening or reading failed. You can also use the stream object directly in a boolean context, like `if(!file)`, which implicitly checks if the failbit or badbit is set. This is a very clean and idiomatic C++ approach.

Example: Safe File Opening

- fstream file("data.dat", ios::in — ios::out — ios::binary);
-
- if (!file) {
- cerr ¡¡ "Error: Could not open the file!" ¡¡ endl;
- return 1; // Exit program with error code
- }
- // Proceed with file operations safely...
- file.close();

Example: Safe File Opening
fstream file("data.dat", ios::in — ios::out — ios::binary);
if (!file) {
cerr ¡¡ "Error: Could not open the file!" ¡¡ endl;
return 1; // Exit program with error code
}
// Proceed with file operations safely...
file.close();

This is the golden rule of file handling: always check if the file opened successfully before attempting to use it. Here, `if(!file)` evaluates to true if the file failed to open (for example, if "data.dat" doesn't exist and we didn't specify ios::trunc). We print to cerr (the standard error stream) and exit gracefully rather than crashing.

Clearing Error Flags: clear()

- Once an error flag (like failbit or eofbit) is set, the stream refuses further operations.
- To recover and continue using the stream, you must clear the flags.
- Method: file.clear();
- Common use case: Reaching EOF, calling clear(), and then using seekg() to go back to the beginning.

Clearing Error Flags: clear()
- Once an error flag (like failbit or eofbit) is set, the stream refuses further operations.
- To recover and continue using the stream, you must clear the flags.
- Method: file.clear();
- Common use case: Reaching EOF, calling clear(), and then using seekg() to go back to the beginning.

A very important concept: if a stream enters a fail state, it goes into 'lockdown'. Any further read or write commands will be ignored. If you want to reuse that stream—for instance, if you hit EOF while searching, but now want to jump back to the top of the file—you MUST call file.clear() first to reset the flags back to goodbit, before you call seekg.

Robust Update Operation with Error Checking

- file.read(reinterpret_cast¡char*¿(&emp), sizeof(Employee));
- if (file.eof()) {
- cout ¡¡ "Record not found (End of File)." ¡¡ endl;
- file.clear(); // Important to clear EOF flag
- } else if (file.fail()) {
- cerr ¡¡ "Error reading file." ¡¡ endl;
- } else {
- // Perform update and write...
- }

Robust Update Operation with Error Checking
file.read(reinterpret_cast¡char*¿(&emp), sizeof(Employee));
if (file.eof()) {
cout ¡¡ "Record not found (End of File)." ¡¡ endl;
file.clear(); // Important to clear EOF flag
} else if (file.fail()) {
cerr ¡¡ "Error reading file." ¡¡ endl;
} else {
// Perform update and write...
}

Let's put it all together. Here we attempt to read a record. Immediately after, we check the stream state. We specifically handle the EOF case—maybe the user asked for record number 100, but only 50 exist. Notice we call clear() if we hit EOF so the stream is usable again. If it's a different failure, we log it. Only if the read succeeds completely do we proceed to the update logic.

Summary

- Updating binary files involves: opening in read/write mode, calculating offsets, reading, modifying, seeking back, and rewriting.
- seekg() and seekp() are essential for random access.
- Never assume file I/O success. Check stream states (good, fail, eof, bad).
- Use clear() to reset stream flags after an EOF or recoverable error.

Summary
- Updating binary files involves: opening in read/write mode, calculating offsets, reading, modifying, seeking back, and rewriting.
- seekg() and seekp() are essential for random access.
- Never assume file I/O success. Check stream states (good, fail, eof, bad).
- Use clear() to reset stream flags after an EOF or recoverable error.

To wrap up, today we learned the complete lifecycle of updating a binary file in place, saving processing time and disk I/O. We also learned how to use the C++ standard library's stream state bits to defend our application against hardware and logical I/O errors.