5.1 File Streams

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

Lecture 45: File Streams in C++

- Unit 5: File Handling and Exception Handling
- Topic: 5.1 File Streams
- Classes: fstream, ifstream, ofstream
- Operations: Opening, Closing, Reading & Writing

Lecture 45: File Streams in C++
- Unit 5: File Handling and Exception Handling
- Topic: 5.1 File Streams
- Classes: fstream, ifstream, ofstream
- Operations: Opening, Closing, Reading & Writing

Welcome to Lecture 45. Today marks the beginning of Unit 5. Up to this point in the course, our programs have operated entirely in RAM, meaning any data entered by the user or computed by our application vanishes the moment the program ends. Today, we bridge the gap between volatile memory and permanent storage by introducing File Streams.

The Need for File Handling

- Volatile Memory vs. Persistent Storage: Standard variables live in RAM and die with the process.
- Data Persistence: Saving state across multiple runs of an application.
- Data Volume: Handling datasets too large to type in manually via cin.
- Data Sharing: Outputting data in a format that other programs (like Excel, databases) can read.

The Need for File Handling
- Volatile Memory vs. Persistent Storage: Standard variables live in RAM and die with the process.
- Data Persistence: Saving state across multiple runs of an application.
- Data Volume: Handling datasets too large to type in manually via cin.
- Data Sharing: Outputting data in a format that other programs (like Excel, databases) can read.

Why do we need files? Imagine writing a game. If you can't save your progress, the game is unplayable. Imagine a banking application; you wouldn't want your account balance to reset to zero every time the server restarts. File handling allows our programs to interact with the non-volatile storage (like a Hard Drive or SSD) to read external data and save output permanently. It is a fundamental concept in software development.

C++ Stream Concept (Review & Extension)

- A 'stream' is an abstraction representing a flow of data.
- Standard streams: cin (input stream), cout (output stream).
- File streams operate identically, but the source/destination is a file instead of the console.
- Requires including the ¡fstream¿ standard library header.

C++ Stream Concept (Review & Extension)
- A 'stream' is an abstraction representing a flow of data.
- Standard streams: cin (input stream), cout (output stream).
- File streams operate identically, but the source/destination is a file instead of the console.
- Requires including the ¡fstream¿ standard library header.

You are already familiar with streams. You've been using 'cin' and 'cout' since day one. 'cout' is a stream that flows from your program to the screen. 'cin' flows from the keyboard to your program. A file stream is exactly the same abstraction. The only difference is the endpoint. Instead of the screen, an output file stream flows to a file on your disk. Because the interface is the same, you already know how to write to files: using the insertion and extraction operators (¡¡ and ¿¿). Just remember to #include ¡fstream¿.

The File Stream Classes

- Defined in the ¡fstream¿ header.
- ifstream (Input File Stream): Used exclusively for reading data from a file.
- ofstream (Output File Stream): Used exclusively for writing data to a file.
- fstream (File Stream): Used for both reading and writing to a single file.

The File Stream Classes
- Defined in the ¡fstream¿ header.
- ifstream (Input File Stream): Used exclusively for reading data from a file.
- ofstream (Output File Stream): Used exclusively for writing data to a file.
- fstream (File Stream): Used for both reading and writing to a single file.

C++ provides three main classes for file operations, mirroring the standard streams. 'ifstream' is like 'cin' for files—it only reads. 'ofstream' is like 'cout' for files—it only writes. 'fstream' is a bidirectional stream, inheriting from both, allowing read and write operations. For most basic sequential file processing, you will use ifstream and ofstream to keep your intentions clear and code secure.

The 4-Step File Processing Workflow

- 1. Include Header: #include ¡fstream¿
- 2. Create a Stream Object: Instantiate ifstream, ofstream, or fstream.
- 3. Open the File: Link the stream object to a physical file on the disk.
- 4. Process Data: Read or write using standard operators (¡¡, ¿¿) or functions.
- 5. Close the File: Sever the link and release system resources.

The 4-Step File Processing Workflow
- 1. Include Header: #include ¡fstream¿
- 2. Create a Stream Object: Instantiate ifstream, ofstream, or fstream.
- 3. Open the File: Link the stream object to a physical file on the disk.
- 4. Process Data: Read or write using standard operators (¡¡, ¿¿) or functions.
- 5. Close the File: Sever the link and release system resources.

Working with files in C++ always follows this consistent pattern. Think of it like making a phone call. First, you get your phone (create object). Then, you dial the number (open the file). You have your conversation (read/write data). Finally, you hang up (close the file). Skipping the last step—closing the file—can lead to data corruption or memory leaks, much like leaving the phone off the hook.

Opening a File (Output)

- Using default constructor and `.open()` method:
- std::ofstream outFile;
- outFile.open("data.txt");
- Using parameterized constructor (preferred shorthand):
- std::ofstream outFile("data.txt");

Opening a File (Output)
Using default constructor and .open() method:
std::ofstream outFile;
outFile.open("data.txt");
Using parameterized constructor (preferred shorthand):
std::ofstream outFile("data.txt");

To open a file, we create an instance of our stream class. Here, we want to write, so we use ofstream. We can instantiate the object and then call the .open() method, passing the file path as a string. However, C++ provides a more concise way: the parameterized constructor allows us to pass the filename directly when creating the object, opening the file immediately. This is the idiomatic C++ approach.

File Open Modes

- Modes define HOW a file is opened. Accessed via std::ios namespace.
- ios::in : Open for reading (default for ifstream).
- ios::out : Open for writing (default for ofstream). OVERWRITES existing data.
- ios::app : Append. Writing starts at the end of the existing file.
- ios::trunc : Truncate. Discards existing file content (default with ios::out).
- ios::binary : Open file in binary mode instead of text mode.

File Open Modes
- Modes define HOW a file is opened. Accessed via std::ios namespace.
- ios::in : Open for reading (default for ifstream).
- ios::out : Open for writing (default for ofstream). OVERWRITES existing data.
- ios::app : Append. Writing starts at the end of the existing file.
- ios::trunc : Truncate. Discards existing file content (default with ios::out).
- ios::binary : Open file in binary mode instead of text mode.

When opening a file, C++ applies a 'mode'. By default, ofstream uses ios::out — ios::trunc. This means if 'data.txt' already exists, its contents are completely wiped out when you open it! If you want to add to a log file instead of erasing it, you must explicitly use ios::app. You can combine modes using the bitwise OR operator (—). For instance, fstream myFile("data.txt", ios::in — ios::out);

Validating the File Open Operation

- File operations can fail (e.g., file not found, permission denied).
- Always check if a file opened successfully before attempting to process it.
- Method 1: The `.is_open()` function returns true if the file is ready.
- Method 2: Stream objects can be evaluated as booleans (e.g., if(!myFile))

Validating the File Open Operation
- File operations can fail (e.g., file not found, permission denied).
- Always check if a file opened successfully before attempting to process it.
- Method 1: The .is_open() function returns true if the file is ready.
- Method 2: Stream objects can be evaluated as booleans (e.g., if(!myFile))

Never assume a file opened successfully. The file might not exist, the disk might be full, or the operating system might deny permission. Trying to read from a closed or failed stream will cause subtle logical errors in your loops. You must always check the state immediately after attempting to open. Using if(!myFile) is a common, succinct C++ idiom that checks if the stream's error flags have been set.

Writing to a Text File

- #include ¡iostream¿
- #include ¡fstream¿
- using namespace std;
- int main() {
- ofstream outFile("scores.txt");
- if (!outFile) { cerr ¡¡ "Error!"; return 1; }
- outFile ¡¡ "Alice " ¡¡ 95 ¡¡ endl;
- outFile ¡¡ "Bob " ¡¡ 88 ¡¡ endl;
- outFile.close();
- return 0;
- }

Writing to a Text File
- #include ¡iostream¿
- #include ¡fstream¿
- using namespace std;
- int main() {
- ofstream outFile("scores.txt");
- if (!outFile) { cerr ¡¡ "Error!"; return 1; }
- outFile ¡¡ "Alice " ¡¡ 95 ¡¡ endl;
- outFile ¡¡ "Bob " ¡¡ 88 ¡¡ endl;
- outFile.close();
- return 0;
- }

Let's look at a complete example of writing. We include fstream. We create an ofstream object named outFile. We immediately check if it opened. If so, we use the exact same insertion operator (¡¡) that we use for cout. The integer 95 is automatically converted to text characters by the stream. Finally, we explicitly close the file.

Reading from a Text File

- ifstream inFile("scores.txt");
- if (!inFile) { cerr ¡¡ "File not found!"; return 1; }
- string name;
- int score;
- while (inFile ¿¿ name ¿¿ score) {
- cout ¡¡ "Read: " ¡¡ name ¡¡ " got " ¡¡ score ¡¡ endl;
- }
- inFile.close();

Reading from a Text File
- ifstream inFile("scores.txt");
- if (!inFile) { cerr ¡¡ "File not found!"; return 1; }
- string name;
- int score;
- while (inFile ¿¿ name ¿¿ score) {
- cout ¡¡ "Read: " ¡¡ name ¡¡ " got " ¡¡ score ¡¡ endl;
- }
- inFile.close();

Reading is slightly more complex because we usually want to read until the file ends. Here, we use ifstream. The extraction operator (¿¿) reads data delimited by whitespace (spaces, tabs, newlines). The expression (inFile ¿¿ name ¿¿ score) returns the stream object itself. In a boolean context (like the while loop condition), the stream evaluates to 'false' if it fails to read (like hitting the end of the file). This is the safest and most robust way to read a file sequentially.

Closing Files: Why does it matter?

- The `.close()` method.
- 1. Flushes the buffer: Data is often held in RAM temporarily for performance. Closing forces it to write to disk.
- 2. Releases OS resources: Operating systems limit the number of open files.
- 3. Unlocks the file: Allows other programs to access it.
- Note: Destructors will close files automatically when the object goes out of scope, but explicit closing is a best practice.

Closing Files: Why does it matter?
- The .close() method.
- 1. Flushes the buffer: Data is often held in RAM temporarily for performance. Closing forces it to write to disk.
- 2. Releases OS resources: Operating systems limit the number of open files.
- 3. Unlocks the file: Allows other programs to access it.
- Note: Destructors will close files automatically when the object goes out of scope, but explicit closing is a best practice.

Why do we call .close()? Disk I/O is slow. C++ mitigates this by buffering—storing up a chunk of text in fast RAM and writing it to the disk all at once. If your program crashes before the buffer is flushed, you lose data. Closing the file flushes this buffer. While C++ stream destructors automatically close files when the function ends, explicit closing makes your intentions clear and frees up system resources immediately, which is crucial in large applications.

Reading Strings with Spaces: getline()

- The ¿¿ operator stops at whitespace.
- To read an entire line, use std::getline(stream, string_variable).
- Example:
- string line;
- ifstream myFile("story.txt");
- while (getline(myFile, line)) {
- cout ¡¡ line ¡¡ endl;
- }

Reading Strings with Spaces: getline()
- The ¿¿ operator stops at whitespace.
- To read an entire line, use std::getline(stream, string_variable).
- Example:
- string line;
- ifstream myFile("story.txt");
- while (getline(myFile, line)) {
- cout ¡¡ line ¡¡ endl;
- }

A common pitfall: if a name in our previous example was 'John Doe', (inFile ¿¿ name) would only read 'John'. 'Doe' would be mistakenly parsed as the integer score, causing a silent failure. When your data contains spaces, or when you want to process a file line-by-line regardless of spaces, use the global std::getline() function provided by the ¡string¿ header. It reads until it hits a newline character.

Understanding End-of-File (EOF)

- Files have an invisible 'EOF' marker at the end.
- The stream's `.eof()` method returns true ONLY AFTER an attempt is made to read past this marker.
- Anti-pattern: while(!inFile.eof()) { ... } often results in processing the last line twice.
- Best Practice: Use the read operation itself as the loop condition.

Understanding End-of-File (EOF)
- Files have an invisible 'EOF' marker at the end.
- The stream's .eof() method returns true ONLY AFTER an attempt is made to read past this marker.
- Anti-pattern: while(!inFile.eof()) { ... } often results in processing the last line twice.
- Best Practice: Use the read operation itself as the loop condition.

A very common mistake for beginners is using while(!inFile.eof()). The problem is that the EOF flag is not set when you read the last byte of data; it is only set when you try to read AGAIN and there is nothing left. This usually causes the loop body to execute one extra time with stale data. Always use the read operation as the condition, e.g., while(getline(inFile, str)), which attempts the read and immediately evaluates if it succeeded.

Lecture Summary

- Include ¡fstream¿ for file handling.
- Use ofstream to write, ifstream to read.
- Constructor parameters or .open() link the stream to a file.
- Modes (ios::app, ios::out) dictate file interaction rules.
- Always check if a file opened properly.
- Read loops evaluate the stream state.
- Always explicitly .close() your files.

Lecture Summary
- Include ¡fstream¿ for file handling.
- Use ofstream to write, ifstream to read.
- Constructor parameters or .open() link the stream to a file.
- Modes (ios::app, ios::out) dictate file interaction rules.
- Always check if a file opened properly.
- Read loops evaluate the stream state.
- Always explicitly .close() your files.

To summarize, file streams in C++ are a powerful but straightforward extension of the console streams you already know. Remember the lifecycle: include, instantiate, open, validate, process, and close. Pay special attention to how you construct your reading loops to avoid EOF bugs, and always ensure your files are closed to preserve data integrity.