5.1 File Streams

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

Lecture 37: File Streams in C++

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

Lecture 37: File Streams in C++
- Unit 5: File Handling and Exception Handling
- Topic 5.1: File Streams
- Classes: fstream, ifstream, ofstream
- Opening, Closing, Reading & Writing Files

Welcome to Lecture 37. Today we begin Unit 5, which focuses on interacting with files and handling errors. Up until now, all the data in our programs was stored in RAM, meaning it was lost as soon as the program terminated. Today, we learn how to persist data to the hard drive using file streams.

Why File Handling?

- Data persistence: Data stored in memory (variables, arrays) is volatile and lost upon program exit.
- Files allow us to store data permanently on secondary storage devices (hard drives, SSDs).
- Enables processing of large datasets that cannot fit entirely in memory.
- Allows data sharing between different programs or different runs of the same program.

Why File Handling?
- Data persistence: Data stored in memory (variables, arrays) is volatile and lost upon program exit.
- Files allow us to store data permanently on secondary storage devices (hard drives, SSDs).
- Enables processing of large datasets that cannot fit entirely in memory.
- Allows data sharing between different programs or different runs of the same program.

Ask the students: 'What happens to your game progress if you close a game without saving?' It's lost. That's because the data was only in memory. File handling allows us to save that state. In enterprise applications, we read configurations, process logs, and save user data, all requiring file I/O operations.

Understanding 'Streams' in C++

- A stream is an abstraction that represents a flow of data.
- It acts as an interface between the program and I/O devices.
- Input Stream: Data flows from a source (keyboard, file) into the program.
- Output Stream: Data flows from the program to a destination (screen, file).
- You already know streams: `cin` (standard input stream) and `cout` (standard output stream).

Understanding 'Streams' in C++
- A stream is an abstraction that represents a flow of data.
- It acts as an interface between the program and I/O devices.
- Input Stream: Data flows from a source (keyboard, file) into the program.
- Output Stream: Data flows from the program to a destination (screen, file).
- You already know streams: cin (standard input stream) and cout (standard output stream).

In C++, we don't directly talk to the hard drive or the keyboard. We talk to a 'stream'. Think of a stream as a water pipe. You pour data into one end, and it comes out the other. You are already familiar with iostream; today we apply the exact same concepts to files using fstream.

C++ File Stream Classes

- To perform file processing in C++, we use classes from the `<fstream>` library.
- `ofstream`: Output File Stream. Used to write data to files.
- `ifstream`: Input File Stream. Used to read data from files.
- `fstream`: File Stream. Can be used for both reading and writing.
- These classes inherit from `ostream`, `istream`, and `iostream` respectively.

C++ File Stream Classes
- To perform file processing in C++, we use classes from the <fstream> library.
- ofstream: Output File Stream. Used to write data to files.
- ifstream: Input File Stream. Used to read data from files.
- fstream: File Stream. Can be used for both reading and writing.
- These classes inherit from ostream, istream, and iostream respectively.

To use these classes, you must `#include <fstream>` at the top of your program. Notice the naming convention: 'o' for output, 'i' for input. If you need to do both, just use fstream. Because they inherit from standard stream classes, you will use the same << and >> operators you use with cout and cin.

The Life Cycle of File Handling

- Every file operation in C++ follows three fundamental steps:
- 1. Open the file: Establish a connection between the stream object and the physical file on the disk.
- 2. Process the file: Read data from the file into variables, or write data from variables to the file.
- 3. Close the file: Disconnect the stream from the file, ensuring all data is saved and system resources are released.

The Life Cycle of File Handling
- Every file operation in C++ follows three fundamental steps:
- 1. Open the file: Establish a connection between the stream object and the physical file on the disk.
- 2. Process the file: Read data from the file into variables, or write data from variables to the file.
- 3. Close the file: Disconnect the stream from the file, ensuring all data is saved and system resources are released.

Emphasize step 3. Forgetting to close a file can lead to data loss or file corruption because the data might still be sitting in a memory buffer waiting to be written to the disk. Always close what you open.

Step 1: Opening a File

- Files can be opened in two ways:
- 1. Using the constructor during object creation:
- std::ofstream outFile("data.txt");
- 2. Using the `open()` member function:
- std::ofstream outFile;
- outFile.open("data.txt");

Step 1: Opening a File
- Files can be opened in two ways:
- 1. Using the constructor during object creation:
- std::ofstream outFile("data.txt");
- 2. Using the open() member function:
- std::ofstream outFile;
- outFile.open("data.txt");

Using the constructor is generally preferred as it initializes the object and opens the file in one step, adhering to RAII (Resource Acquisition Is Initialization) principles. The open() method is useful if you create the stream object first and need to open a file later, perhaps based on user input.

File Opening Modes

- Modes determine how the file is accessed. They are defined in the `ios` class.
- `ios::in`: Open for input (reading). Default for `ifstream`.
- `ios::out`: Open for output (writing). Default for `ofstream`. Overwrites existing file.
- `ios::app`: Append. All output is appended to the end of the file.
- `ios::trunc`: Truncate. Discard file contents if it exists.
- `ios::binary`: Open file in binary mode (instead of text mode).

File Opening Modes
- Modes determine how the file is accessed. They are defined in the ios class.
- ios::in: Open for input (reading). Default for ifstream.
- ios::out: Open for output (writing). Default for ofstream. Overwrites existing file.
- ios::app: Append. All output is appended to the end of the file.
- ios::trunc: Truncate. Discard file contents if it exists.
- ios::binary: Open file in binary mode (instead of text mode).

You can combine modes using the bitwise OR operator |. For example, `ios::out | ios::app` opens a file for writing but appends to the end instead of overwriting. Note that ofstream defaults to `ios::out | ios::trunc`, meaning if you don't specify app, the old data is deleted!

Verifying the File Opened Successfully

- Always check if a file was successfully opened before trying to read or write.
- Files might fail to open if they don't exist (for reading), lack permissions, or the disk is full.
- Using `is_open()` method:
- if (myFile.is_open()) { / *proceed* / }
- Using the stream object itself as a boolean:
- if (!myFile) { cout ¡¡ "Error opening file"; }

Verifying the File Opened Successfully
- Always check if a file was successfully opened before trying to read or write.
- Files might fail to open if they don't exist (for reading), lack permissions, or the disk is full.
- Using is_open() method:
- if (myFile.is_open()) { / proceed / }
- Using the stream object itself as a boolean:
- if (!myFile) { cout ¡¡ "Error opening file"; }

This is a common source of bugs for beginners. You write a program, it silently fails to open the file, and does nothing. Always validate! The `!myFile` syntax relies on an overloaded boolean conversion operator in the stream class that returns false if any error flags are set.

Step 2: Writing to a File (ofstream)

- Use the stream insertion operator <<, just like with cout.
- Example:
- std::ofstream out("test.txt");
- if (out) {
- out ¡¡ "Hello, File!" ¡¡ std::endl;
- out ¡¡ 123 ¡¡ " " ¡¡ 45.6;
- }

Step 2: Writing to a File (ofstream)
- Use the stream insertion operator <<, just like with cout.
- Example:
- std::ofstream out("test.txt");
- if (out) {
- out ¡¡ "Hello, File!" ¡¡ std::endl;
- out ¡¡ 123 ¡¡ " " ¡¡ 45.6;
- }

Notice how familiar this looks. If you know how to print to the screen with cout, you know how to write to a file. The ofstream object simply directs the flow of characters to the disk instead of the console.

Step 2: Reading from a File (`ifstream`)

- Use the stream extraction operator >>, just like with `cin`.
- Reads data word-by-word (whitespace delimited).
- Example:
- std::ifstream in("data.txt");
- std::string word;
- int number;
- in ¿¿ word ¿¿ number;
- // Reads the first string and first integer from the file.

Step 2: Reading from a File (ifstream)
- Use the stream extraction operator >>, just like with cin.
- Reads data word-by-word (whitespace delimited).
- Example:
- std::ifstream in("data.txt");
- std::string word;
- int number;
- in ¿¿ word ¿¿ number;
- // Reads the first string and first integer from the file.

The >> operator skips leading whitespace (spaces, tabs, newlines) and reads until it hits the next whitespace. This is great for structured data like lists of numbers, but not great if you want to read a whole sentence that contains spaces.

Reading Files Line-by-Line

- To read an entire line including spaces, use the `std::getline()` function.
- Syntax: `getline(inputStream, stringVariable);`
- Example:
- std::ifstream in("poem.txt");
- std::string line;
- while (std::getline(in, line)) {
- std::cout ¡¡ line ¡¡ std::endl;
- }

Reading Files Line-by-Line
- To read an entire line including spaces, use the std::getline() function.
- Syntax: getline(inputStream, stringVariable);
- Example:
- std::ifstream in("poem.txt");
- std::string line;
- while (std::getline(in, line)) {
- std::cout ¡¡ line ¡¡ std::endl;
- }

This is the most robust way to read text files. The `getline()` function returns the stream itself, which evaluates to `true` as long as a line was successfully read. The `while` loop automatically terminates when it hits the end of the file (EOF).

End of File (EOF) and Error States

- Stream objects maintain state flags to indicate their status:
- `good()`: Returns true if no error flags are set.
- `eof()`: Returns true if the End-Of-File has been reached.
- `fail()`: Returns true if a logical error occurred (e.g., tried to read an int, but found a letter).
- `bad()`: Returns true if a critical I/O error occurred (e.g., disk disconnected).

End of File (EOF) and Error States
- Stream objects maintain state flags to indicate their status:
- good(): Returns true if no error flags are set.
- eof(): Returns true if the End-Of-File has been reached.
- fail(): Returns true if a logical error occurred (e.g., tried to read an int, but found a letter).
- bad(): Returns true if a critical I/O error occurred (e.g., disk disconnected).

While you can check `.eof()` explicitly in a loop, the preferred idiom in C++ is to check the state of the stream itself, e.g., `while (file >> data)`. This checks for both EOF and read failures simultaneously.

Step 3: Closing a File

- Use the `close()` member function: `myFile.close();`
- Why close manually?
- - Flushes output buffers to the disk (ensures data is written).
- - Releases operating system file locks.
- - Frees memory resources.
- Note: C++ file stream destructors automatically call `close()` when the object goes out of scope.

Step 3: Closing a File
- Use the close() member function: myFile.close();
- Why close manually?
- - Flushes output buffers to the disk (ensures data is written).
- - Releases operating system file locks.
- - Frees memory resources.
- Note: C++ file stream destructors automatically call close() when the object goes out of scope.

Even though destructors handle closing automatically when a function ends, it's considered best practice to explicitly close files as soon as you are done with them. If your program crashes before the destructor is called, some data in the buffer might not be written.

Complete Code Example: Writing

```
#include ¡iostream¿
#include ¡fstream¿
using namespace std;
int main() {
ofstream file("records.txt");
if (!file) { cerr ¡¡ "Error!\n"; return 1; }
file ¡¡ "Alice 95\n";
file ¡¡ "Bob 82\n";
file.close();
cout ¡¡ "Saved.\n";
return 0;
}
```


Complete Code Example: Writing
#include ¡iostream¿
#include ¡fstream¿
using namespace std;
int main() {
ofstream file("records.txt");
if (!file) { cerr ¡¡ "Error!\n"; return 1; }
file ¡¡ "Alice 95\n";
file ¡¡ "Bob 82\n";
file.close();
cout ¡¡ "Saved.\n";
return 0;
}

Walk the students through this code line by line. Highlight the inclusion of `<fstream>`, the creation of the `ofstream` object, the validation check, the writing process using `<<`, and the explicit `close()`. Mention `cerr` is the standard error stream.

Complete Code Example: Reading

- #include ¡iostream¿
- #include ¡fstream¿
- #include ¡string¿
- using namespace std;
- int main() {
- ifstream file("records.txt");
- if (!file) { cerr ¡¡ "Error!\n"; return 1; }
- string name;
- int score;
- while (file ¿¿ name ¿¿ score) {
- cout ¡¡ name ¡¡ ": " ¡¡ score ¡¡ "\n";
- }
- file.close();
- return 0;
- }

Complete Code Example: Reading
- #include ¡iostream¿
- #include ¡fstream¿
- #include ¡string¿
- using namespace std;
- int main() {
- ifstream file("records.txt");
- if (!file) { cerr ¡¡ "Error!\n"; return 1; }
- string name;
- int score;
- while (file ¿¿ name ¿¿ score) {
- cout ¡¡ name ¡¡ ": " ¡¡ score ¡¡ "\n";
- }
- file.close();
- return 0;
- }

In this read example, pay attention to the `while` loop condition. `file >> name >> score` attempts to read two pieces of data. If successful, it returns a stream in a 'good' state, meaning true. Once EOF is reached, it returns false, gracefully exiting the loop.

Summary & Best Practices

- Always `#include <fstream>` for file operations.
- Always verify that a file opened successfully using `!stream` or `stream.is_open()`.
- Use `std::getline()` for text files when whitespace matters.
- Use `while (file >> var)` instead of `while (!file.eof())` for safer loops.
- Explicitly `close()` files to prevent resource leaks and flush buffers.
- Prefer passing file streams to functions by reference (`std::ifstream&`).

Recap the main points. The point about passing streams by reference is important: stream objects cannot be copied because they represent a unique connection to a physical resource on the OS.