5.1 File Streams

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

Lecture 41: File Streams in C++

- Unit 5: File Handling and Exception Handling
- Topic 5.1: File Streams
- Object Oriented Programming with C++

Lecture 41: File Streams in C++
- Unit 5: File Handling and Exception Handling
- Topic 5.1: File Streams
- Object Oriented Programming with C++

Welcome to Lecture 41. Today we are beginning Unit 5, which focuses on File Handling and Exception Handling. We will start by exploring how C++ interacts with files using file streams, allowing us to permanently store data beyond the execution of our program.

Why Do We Need File Handling?

- Volatile vs. Non-Volatile Memory:
- - Standard variables and objects are stored in RAM (volatile memory) and are lost when the program terminates.
- Persistence:
- - File handling allows data to be stored on a hard drive (non-volatile memory) for future retrieval and manipulation.
- Data Sharing:
- - Files allow programs to output data that can be read by other programs or users.

Why Do We Need File Handling?
- Volatile vs. Non-Volatile Memory:
- - Standard variables and objects are stored in RAM (volatile memory) and are lost when the program terminates.
- Persistence:
- - File handling allows data to be stored on a hard drive (non-volatile memory) for future retrieval and manipulation.
- Data Sharing:
- - Files allow programs to output data that can be read by other programs or users.

Up until now, every time we run our programs, we enter data, process it, output it, and when the program ends, everything vanishes. The variables were in RAM. File handling gives our programs 'memory' across different runs. We can save configurations, user data, or processing results. This is crucial for almost any real-world application, from games saving your progress to databases storing user records.

The C++ Stream Concept

- What is a Stream?
- - A stream is an abstraction that represents a device on which input and output operations are performed.
- - It can be thought of as a flow of characters (or bytes).
- Source and Destination:
- - Input stream: Flow of data INTO the program (e.g., from keyboard or file).
- - Output stream: Flow of data OUT OF the program (e.g., to screen or file).

The C++ Stream Concept
- What is a Stream?
- - A stream is an abstraction that represents a device on which input and output operations are performed.
- - It can be thought of as a flow of characters (or bytes).
- Source and Destination:
- - Input stream: Flow of data INTO the program (e.g., from keyboard or file).
- - Output stream: Flow of data OUT OF the program (e.g., to screen or file).

C++ treats all I/O operations through the concept of 'streams'. Imagine a stream of water flowing; similarly, a data stream is a sequence of bytes flowing into or out of your program. You're already familiar with this! `cin` is an input stream from the keyboard, and `cout` is an output stream to the console. File streams work exactly the same way, just connected to a file on your disk instead of the console.

The ¡fstream¿ Library

- Header File:
- - Must `#include <fstream>` to work with file streams.
- Key Classes provided:
- 1. `ofstream`: Stream class to write on files (Output File Stream).
- 2. `ifstream`: Stream class to read from files (Input File Stream).
- 3. `fstream`: Stream class to both read and write from/to files.

The ¡fstream¿ Library
- Header File:
- - Must #include <fstream> to work with file streams.
- Key Classes provided:
- 1. ofstream: Stream class to write on files (Output File Stream).
- 2. ifstream: Stream class to read from files (Input File Stream).
- 3. fstream: Stream class to both read and write from/to files.

To use file streams, you must include the `<fstream>` header file at the top of your program. This header gives you access to three main classes. Remember the 'o' stands for output, the 'i' for input. If you only need to read, use `ifstream` to protect against accidental writes. If you only need to write, use `ofstream`. If you need to do both simultaneously, `fstream` is your tool.

The Lifecycle of File Handling

- Every file operation in C++ follows these general steps:
- 1. Declare a file stream object (ifstream, ofstream, or fstream).
- 2. Open the file (connecting the object to the physical file).
- 3. Check if the file opened successfully.
- 4. Process the file (Read from it or Write to it).
- 5. Close the file (disconnect and release resources).

The Lifecycle of File Handling
- Every file operation in C++ follows these general steps:
- 1. Declare a file stream object (ifstream, ofstream, or fstream).
- 2. Open the file (connecting the object to the physical file).
- 3. Check if the file opened successfully.
- 4. Process the file (Read from it or Write to it).
- 5. Close the file (disconnect and release resources).

Think of this like making a phone call. First, you need a phone (declare object). Then you dial the number (open the file). You check if they answered (check for success). You talk (read/write). Finally, you hang up (close the file). If you skip hanging up, you might lock the line for others. We will look at each of these steps in detail.

Opening Files

- Method 1: Using the Constructor
- - `ofstream outFile("data.txt");`
- Method 2: Using the open() function
- - `ofstream outFile;`
- - `outFile.open("data.txt");`
- File Modes (Optional 2nd argument):
- - `ios::out` (Write - default for ofstream)
- - `ios::in` (Read - default for ifstream)
- - `ios::app` (Append - write at the end)
- - `ios::trunc` (Truncate - clear contents before writing)

Opening Files
- Method 1: Using the Constructor
- - ofstream outFile("data.txt");
- Method 2: Using the open() function
- - ofstream outFile;
- - outFile.open("data.txt");
- File Modes (Optional 2nd argument):
- - ios::out (Write - default for ofstream)
- - ios::in (Read - default for ifstream)
- - ios::app (Append - write at the end)
- - ios::trunc (Truncate - clear contents before writing)

You can open a file immediately when you create the stream object using its constructor, or you can create an empty stream and open it later using the `.open()` method. The second argument to open is the 'mode'. By default, an `ofstream` will truncate (erase) an existing file when it opens it. If you want to add to an existing file, you must explicitly use `ios::app`.

Checking if a File Opened Successfully

- Why check?
- - The file might not exist (when reading).
- - You might lack permissions.
- - The disk might be full.
- How to check:
- - `is_open()` method: Returns true if open, false otherwise.
- - Using the stream object as a boolean: `if(outFile) { ... }` or `if(!outFile) { ... }`

Checking if a File Opened Successfully
- Why check?
- - The file might not exist (when reading).
- - You might lack permissions.
- - The disk might be full.
- How to check:
- - is_open() method: Returns true if open, false otherwise.
- - Using the stream object as a boolean: if(outFile) { ... } or if(!outFile) { ... }

Always check if a file opened successfully! If you try to read from a file that doesn't exist, your program will fail silently or crash if you don't handle it. The `is_open()` function is the most explicit and recommended way to verify the file is ready for processing.

Example: Writing to a File (ofstream)

- #include ¡iostream¿
- #include ¡fstream¿
- using namespace std;
-
- int main() {
- ofstream outFile("student.txt"); // Create and open
- if (outFile.is_open()) {
- outFile ¡¡ "John Doe\n"; // Write to file
- outFile ¡¡ 20 ¡¡ "\n"; // Write an integer
- outFile.close(); // Always close!
- cout ¡¡ "Data saved.\n";
- } else {
- cout ¡¡ "Error opening file.\n";
- }
- return 0;
- }

Example: Writing to a File (ofstream)
#include ¡iostream¿
#include ¡fstream¿
using namespace std;
int main() {
ofstream outFile("student.txt"); // Create and open
if (outFile.is_open()) {
outFile ¡¡ "John Doe\n"; // Write to file
outFile ¡¡ 20 ¡¡ "\n"; // Write an integer
outFile.close(); // Always close!
cout ¡¡ "Data saved.\n";
} else {
cout ¡¡ "Error opening file.\n";
}
return 0;
}

Notice how similar writing to a file is to writing to the console. We just replace `cout` with our `ofstream` object `outFile`. The stream insertion operator << knows how to format standard data types like strings and integers into text suitable for the file.

Closing Files

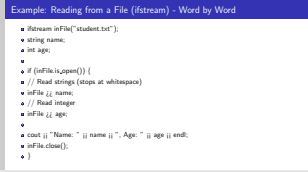
- The close() Method:
- - `stream_object.close();`
- Why is it necessary?
- - Flushes the buffer: Ensures all data temporarily held in memory is actually written to the disk.
- - Releases OS locks: Allows other programs to access the file.
- - Free resources: The operating system limits how many files a program can have open at once.

Closing Files
- The close() Method:
- - stream_object.close();
- Why is it necessary?
- - Flushes the buffer: Ensures all data temporarily held in memory is actually written to the disk.
- - Releases OS locks: Allows other programs to access the file.
- - Free resources: The operating system limits how many files a program can have open at once.

When you write to a file, the OS often stores that data in a temporary memory buffer to improve performance. It only actually writes to the hard drive when the buffer is full or when you close the file. If your program crashes before closing, you might lose data. Therefore, `close()` is critical for data integrity.

Example: Reading from a File (ifstream) - Word by Word

- ifstream inFile("student.txt");
- string name;
- int age;
-
- if (inFile.is_open()) {
- // Read strings (stops at whitespace)
- inFile ¿¿ name;
- // Read integer
- inFile ¿¿ age;
-
- cout ¡¡ "Name: " ¡¡ name ¡¡ ", Age: " ¡¡ age ¡¡ endl;
- inFile.close();
- }

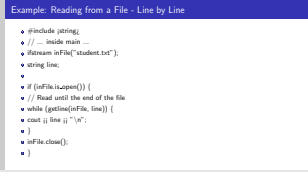


To read, we use `ifstream` and the stream extraction operator >>. Just like `cin`, the >> operator stops reading when it hits whitespace (spaces, tabs, newlines). If our file contained 'John Doe', `inFile >> name` would only read 'John'. This is important to remember when dealing with text containing spaces.

Example: Reading from a File - Line by Line

```
#include ¡string¿
// ... inside main ...
ifstream inFile("student.txt");
string line;

if (inFile.is_open()) {
// Read until the end of the file
while (getline(inFile, line)) {
cout ¡¡ line ¡¡ "\n";
}
inFile.close();
}
```

To solve the whitespace issue, we use the `getline()` function. `getline(stream, string_var)` reads an entire line up to the newline character and stores it in the string variable. Putting this in a `while` loop is the standard idiomatic way in C++ to process a text file line by line until the end of the file (EOF) is reached.

Appending to a File

- Problem: Default `ofstream` overwrites the existing file.
- Solution: Use `ios::app` mode.
-
- Code:
- `ofstream outFile("log.txt", ios::app);`
- `if(outFile.is_open()) {`
- `outFile << "New log entry\n";`
- `outFile.close();`
- `}`

Appending to a File
Problem: Default ofstream overwrites the existing file.
Solution: Use ios::app mode.
Code:
ofstream outFile("log.txt", ios::app);
if(outFile.is_open()) {
outFile << "New log entry\n";
outFile.close();
}

If you are creating a log file or adding new student records, you don't want to delete the old ones! By passing `ios::app` (append) as the second parameter to the constructor or the open method, the internal file pointer is moved to the end of the file before any writing occurs.

Detecting the End of File (EOF)

- The EOF State:
- - When a stream attempts to read past the last character of a file, the EOF flag is set.
- Using `.eof()` method:
- - Returns `true` if the EOF flag is set.
- Preferred approach:
- - Evaluate the stream operation directly in a loop condition (e.g., `while(inFile >> data)`). This checks for EOF and other errors simultaneously.

Detecting the End of File (EOF)
- The EOF State:
- - When a stream attempts to read past the last character of a file, the EOF flag is set.
- Using `.eof()` method:
- - Returns `true` if the EOF flag is set.
- Preferred approach:
- - Evaluate the stream operation directly in a loop condition (e.g., `while(inFile >> data)`). This checks for EOF and other errors simultaneously.

While C++ provides an `eof()` method, it is a common beginner mistake to write `while(!inFile.eof())`. This often results in processing the last line twice because the EOF flag isn't set until AFTER a read fails. The preferred, robust method is to make the read operation itself the condition of the while loop, as seen in the getline example.

Summary of Today's Lecture

- File handling uses streams: `<fstream>`, `ifstream`, `ofstream`.
- Standard process: Declare, Open, Check, Process, Close.
- << is used for writing to `ofstream`.
- >> is used for reading tokens from `ifstream`.
- `getline()` is used for reading lines from `ifstream`.
- Always check `is_open()` and remember to `close()` files.

Summary of Today's Lecture
- File handling uses streams: <fstream>, ifstream, ofstream.
- Standard process: Declare, Open, Check, Process, Close.
- << is used for writing to ofstream.
- >> is used for reading tokens from ifstream.
- getline() is used for reading lines from ifstream.
- Always check is_open() and remember to close() files.

To summarize, file handling allows our data to survive past the lifetime of the program. We treat files just like the console, using streams and the insertion/extraction operators. Always remember defensive programming: check if the file actually opened, and be polite to the operating system by closing the file when you are done.