

Himalaya College of Engineering

**Advanced C++ Programming Lab Report**

Lab 10: Stream Computation: File Handling

**Prepared By : Monsoon Sapkota** (HCE081BEI022)

**Subject :** Object-Oriented Programming (OOP)

**Program :** Bachelor of Electronics, Communication and Information Engineering

**Institution :** Himalaya College of Engineering

**Date :** July 24, 2025

**OBJECTIVE**

* To understand the concept of file streams and their role in C++ for input/output operations.
* To implement file handling operations such as reading data from files and writing data to files.
* To learn how to use different file modes for various I/O needs.
* To develop programs that can store and retrieve persistent data.

**BACKGROUND THEORY**

In C++, **stream computation** refers to the flow of data, often to or from peripheral devices like the console or files. A **stream** is an abstract representation of a device on which input, and output operations are performed. When dealing with files, we use file streams to manage data transfer between our program and external files on storage.

* **File Handling** allows programs to store information permanently on storage devices (like hard drives) and retrieve it later. This is crucial for applications that need to save user data, configurations, or logs.
* The C++ Standard Library provides the <fstream> header for file stream operations. This header defines classes like ofstream (for output file stream), ifstream (for input file stream), and fstream (for both input and output file stream).
* **Opening a file** connects a file stream object to a physical file, preparing it for reading or writing.
* **Writing to a file** sends data from the program's memory to the file.
* **Reading from a file** retrieves data from the file into the program's memory.
* **Closing a file** disconnects the file stream object from the physical file, ensuring all data is saved and resources are released.

**Key Concepts and Syntax:**

**File Stream Classes:**

* ofstream: Used to create files and write data to them.
* ifstream: Used to open existing files and read data from them.
* fstream: Can be used for both reading from and writing to files.

**Opening a File:**

Files can be opened using the open() member function or directly through the constructor of the file stream class.

#include <fstream> // Required header

// Using constructor ofstream outFile("example.txt"); ifstream inFile("example.txt"); fstream ioFile("data.txt", ios::in | ios::out); // For both read/write

// Using open() function ofstream outFile; outFile.open("output.txt");

ifstream inFile; inFile.open("input.txt");

**File Modes (used with open() or constructor):**

* ios::out: Open for writing. Creates the file if it doesn't exist, overwrites if it does.
* ios::in: Open for reading.
* ios::app: Open for writing, appends data to the end of the file.
* ios::trunc: Truncates (empties) the file if it exists (default for ios::out).
* ios::ate: Sets the initial position at the end of the file.
* ios::binary: Opens the file in binary mode (default is text mode).

**Checking if a File is Open:** C++

if (outFile.is\_open()) {

// File is open, proceed with I/O

} else {

// Error opening file

}

**Writing to a File:**

Uses the insertion operator <<, similar to cout.

C++

outFile << "Hello, File Handling!" << endl; outFile << 12345 << endl;

**Reading from a File:**

Uses the extraction operator >>, similar to cin, or getline() for entire lines.

C++

string line; int number;

inFile >> number; // Reads a single word/number getline(inFile, line); // Reads an entire line

**Closing a File:**

It's good practice to explicitly close files. File stream objects also close files automatically when they go out of scope (destructor is called). C++

outFile.close(); inFile.close();

**Example: Writing to and Reading from a File**

C++

#include <iostream>

#include <fstream> // For file stream operations #include <string> // For string manipulation using namespace std; int main() {

// --- Writing to a file --- ofstream outFile("my\_data.txt"); // Create and open file for writing

if (outFile.is\_open()) { cout << "Writing data to my\_data.txt..." << endl; outFile << "Name: Alice" << endl; outFile << "Age: 30" << endl; outFile << "City: New York" << endl; outFile.close(); // Close the file cout << "Data written successfully." << endl;

} else { cerr << "Error: Unable to open file for writing." << endl; return 1; // Indicate an error

}

// --- Reading from a file --- ifstream inFile("my\_data.txt"); // Open file for reading

if (inFile.is\_open()) { cout << "\nReading data from my\_data.txt:" << endl;

string line; while (getline(inFile, line)) { // Read line by line until end of file cout << line << endl;

} inFile.close(); // Close the file

} else { cerr << "Error: Unable to open file for reading." << endl; return 1; // Indicate an error

}

return 0; // Program finished successfully

}

**Explanation:**

* The ofstream outFile("my\_data.txt"); line creates an output file stream object and attempts to open "my\_data.txt" for writing. If the file doesn't exist, it's created. If it does, its content is cleared.
* We use outFile << ... to write data to the file, just like cout.
* The outFile.close(); ensures all data is saved to the file and the connection is released.
* Similarly, ifstream inFile("my\_data.txt"); opens the same file for reading.
* The while (getline(inFile, line)) loop reads the file line by line until there's nothing left to read.
* inFile.close(); closes the input file stream.

**Purpose and Characteristics of File Streams:**

* **Purpose:** File streams are designed to facilitate communication between a C++ program and files on disk. They provide a standardized way to save program state, retrieve persistent data, log events, and interact with external data sources. They are essential for applications that require data to persist beyond the program's execution.
* **Characteristics:**
  + They act as an abstraction layer, allowing programs to interact with files using familiar << and >> operators, similar to console I/O.
  + They manage low-level details of file operations (e.g., buffering, error handling).
  + fstream objects automatically handle opening and closing resources, though explicit close() calls are good practice.
  + Error flags (e.g., fail(), eof()) can be checked to determine the status of file operations.