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.

THEORY

In C++, **stream computation** refers to the flow of data, typically to or from peripheral devices such as 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 utilize file streams to manage data transfer between our program and external files stored on storage devices.

File Handling enables programs to permanently store information on storage devices, such as hard drives, and retrieve it later. This functionality is essential for applications that require saving user data, configurations, or logs.

The C++ Standard Library offers the <fstream> header, which provides classes for file stream operations. These classes include ofstream (for output file streams), ifstream (for input file streams), and fstream (for both input and output file streams).

• **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 readingif (inFile.is_open()) { from my_data.txt:" << endl;
string line; line until end of file } inFile.close(); // Close
the file
} else { reading." << endl; }
cout << "\nReading data
while (getline(inFile, line)) { // Read line by
cout << line << endl;
cerr << "Error: Unable to open file for
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.

1.)Write a program in C++ to write and read the contents of a text file note.txt using file streams. The program should accept a sentence from the user and store it in the file. Then, it should read thecontent back and display it on the screen.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
string sentence;
// Writing to the file
ofstream outFile("note.txt");
if (!outFile) {
cout << "Error opening file for writing!" << endl;
return 1;
}
cout << "Enter a sentence: ";
getline(cin, sentence);
outFile << sentence;
outFile.close();
// Reading from the file
ifstream inFile("note.txt");
if (!inFile) {
cout << "Error opening file for reading!" << endl;
return 1;
}
cout << "\nContents of the file:\n";
string line;
while (getline(inFile, line)) {
cout << line << endl;
}
inFile.close();
return 0;
}
```

2.)Write a program in C++ to define a class Book with data members book id, title, and price. Accept data for 5 books from the user and store them in a file books.txt. Then read the data from the file and display details of books whose price is below Rs. 500.

```
#include <iostream>
#include <fstream>
using namespace std;class Book {
int bookId;
string title;
float price;
public:
void input() {
cout << "Enter Book ID: ";
cin >> bookId;
```

```
cin.ignore(); // to clear newline from input buffer
cout << "Enter Title: ";
getline(cin, title);
cout << "Enter Price: ";
cin >> price;
}
void display() const {
cout << "Book ID: " << bookId << endl;
cout << "Title: " << title << endl;
cout << "Price: Rs. " << price << endl;
cout << "-----------------------" << endl;
}
float getPrice() const {
return price;
}
// File write helper
void writeToFile(ofstream &fout) {
fout << bookId << endl;
fout << title << endl;
fout << price << endl;
}
// File read helper
void readFromFile(ifstream &fin) {
fin >> bookId;
fin.ignore(); // ignore newline
getline(fin, title);
fin >> price;
}
};
int main() {
Book b;
ofstream fout("books.txt");
if (!fout) {
cerr << "Error opening file for writing!" << endl;
return 1;
}
cout << "Enter details for 5 books:" << endl;
for (int i = 0; i < 5; i++) {cout << "\nBook " << i + 1 << ":" << endl;
b.input();
b.writeToFile(fout);
}
fout.close();
ifstream fin("books.txt");
if (!fin) {
cerr << "Error opening file for reading!" << endl;
return 1;
}
cout << "\nBooks with price below Rs. 500:\n";
```

```
cout << "------------------------" << endl;
while (!fin.eof()) {
b.readFromFile(fin);
if (fin && b.getPrice() < 500) {
b.display();
}
}
fin.close();
return 0;
}
```

Q3.)Write a program in C++ to open an existing text file info.txt, count the number of words and lines, and display the results.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
ifstream fin("info.txt");
if (!fin) {
cerr << "Error: Could not open file 'info.txt'" << endl;
return 1;
}
string line;
int lineCount = 0;
int wordCount = 0;
while (getline(fin, line)) {
lineCount++;
int length = line.length();
bool inWord = false;
for (int i = 0; i < length; i++) {
if (isspace(line[i])) {
inWord = false;
} else if (!inWord) {wordCount++;
inWord = true;
}
}
}
fin.close();
cout << "Number of lines: " << lineCount << endl;
cout << "Number of words: " << wordCount << endl;
return 0;
}
```

DISCUSSION:

File handling in C++ utilizes streams to read data from files and write data to files. Special classes like ifstream, ofstream, and fstream facilitate these operations. This enables programs to save data permanently, retrieve stored information, and exchange data effortlessly. File handling is analogous to console input/output but operates with files, enhancing the utility and flexibility of programs. It involves opening a file, executing read/write actions, and subsequently closing the file appropriately.

CONCLUSSION:

File handling with streams in C++ is a powerful feature that simplifies the process of reading from and writing to files. This capability allows programs to store and retrieve data even after the program's execution has ended. By utilizing file streams, developers can efficiently manage data, making their programs more useful and adaptable. A solid understanding of file handling is crucial for developing applications that necessitate persistent data storage or data exchange.