TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: - 09

Title: - Stream computation

Submitted by: -

Name: -Gaurab Pandey

Roll NO: - 15

Submitted To: -

Department Of

Checked by: -

Date of submission: -

Objective

To understand and implement basic stream operations in C++ for input/output (I/O) manipulation, including reading from and writing to standard streams (cin, cout) and files (ifstream, ofstream), while applying formatting techniques for efficient data handling.

Theory

Streams in C++ are sequences of bytes that facilitate data flow between the program and I/O devices (keyboard, screen, files). They provide an abstraction for handling input and output operations uniformly.

1.Stream Class Hierarchy

- istream (Input Stream): Base class for input operations (e.g., cin, ifstream).
- ostream (Output Stream): Base class for output operations (e.g., cout, ofstream).
- iostream (Input/Output Stream): Derived from
  both istream and ostream (e.g., fstream).

2.Standard Streams

- cin: Reads input from the keyboard.
- cout: Writes output to the screen.
- cerr/clog: For error and logging messages (unbuffered/buffered).

3.File Streams

- ifstream: Reads data from files.
- ofstream: Writes data to files.
- fstream: Handles both input/output 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 the content back and display it on the screen.

Code:

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    string userInput;
    ofstream outFile;
    ifstream inFile;
    cout << "Enter a sentence to store in the file: ";
    getline(cin, userInput);

    outFile.open("note.txt");
    if (outFile.is_open()) {
        outFile << userInput;
        outFile.close();
        cout << "Data written to file successfully.\n";
    } else {
        cerr << "Error opening file for writing!\n";
        return 1;
    }

    cout << "\nReading from file:\n";
    inFile.open("note.txt");
    if (inFile.is_open()) {
        string fileContent;
        getline(inFile, fileContent);
        cout << "File contents: " << fileContent << endl;
        inFile.close();
    } else {
        cerr << "Error opening file for reading!\n";
        return 1;
    }

    return 0;
}
```

Output:

```
Enter a sentence to store in the file: Hello my name is gaurab
Data written to file successfully.

Reading from file:
File contents: Hello my name is gaurab
```

2. Write a C++ program to create a binary file employee.dat that stores details of employees (emp id,emp name, and salary) using a class Employee. Read the data from the file and display only those employees whose salary is more than Rs. 30,000.

Code:

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class Employee {
    int emp_id;
    char name[50];
    float salary;
public:
    void input() {
        cout << "Enter employee ID: ";
        cin >> emp_id;
        cin.ignore();
        cout << "Enter employee name: ";
        cin.getline(name, 50);
        cout << "Enter salary: Rs.";
        cin >> salary;
    }

    void display() const {
        cout << "ID: " << emp_id << "\tName: " << name << "\tSalary: Rs." << salary << endl;
    }

    float getSalary() const {
        return salary;
    }
};
```

```cpp
int main() {
    fstream file;
    Employee emp;
    int n;


    file.open("employee.dat", ios::out | ios::binary);
    if (!file) {
        cerr << "Error opening file for writing." << endl;
        return 1;
    }

    cout << "How many employees do you want to enter? ";
    cin >> n;
    cin.ignore();

    for (int i = 0; i < n; ++i) {
        cout << "\nEnter details for employee " << i + 1 << ":\n";
        emp.input();
        file.write(reinterpret_cast<char*>(&emp), sizeof(emp));
    }
    file.close();

    file.open("employee.dat", ios::in | ios::binary);
    if (!file) {
        cerr << "Error opening file for reading." << endl;
        return 1;
    }

    cout << "\nEmployees with salary more than Rs.30,000:\n";
    cout << "----------------------------------------\n";
    while (file.read(reinterpret_cast<char*>(&emp), sizeof(emp))) {
        if (emp.getSalary() > 30000) {
            emp.display();
        }
    }
    file.close();

    return 0;
}
```

Output:

```
How many employees do you want to enter? 3

Enter details for employee 1:
Enter employee ID: 1
Enter employee name: ram
Enter salary: Rs.25000

Enter details for employee 2:
Enter employee ID: 2
Enter employee name: sita
Enter salary: Rs.30000

Enter details for employee 3:
Enter employee ID: 3
Enter employee name: gita
Enter salary: Rs.40000

Employees with salary more than Rs.30,000:
---------------------------------------
ID: 3   Name: gita      Salary: Rs.40000
```

Discussion

The lab exercise on stream computation provided a practical exploration of binary file handling in C++ using file streams. We implemented a program to store employee data (ID, name, and salary) in a binary file and subsequently retrieve records meeting specific criteria (salaries exceeding Rs. 30,000). This exercise demonstrated the efficiency of binary file operations for structured data storage, particularly in terms of memory usage and access speed compared to text files. Key technical aspects included the use of reinterpret_cast for type conversion during file I/O operations, proper file stream management (including opening modes and error checking), and the interaction between class objects and persistent storage. The implementation also highlighted the importance of buffer management, particularly when handling mixed data types (integers, character arrays, and floating-point values) within a single record structure.

Conclusion

This lab successfully achieved its objectives of demonstrating binary file operations through a practical employee records system. The exercise reinforced several core concepts of stream computation: the efficiency of binary file I/O for structured data, the integration of class objects with file streams, and the importance of proper resource management in file operations.