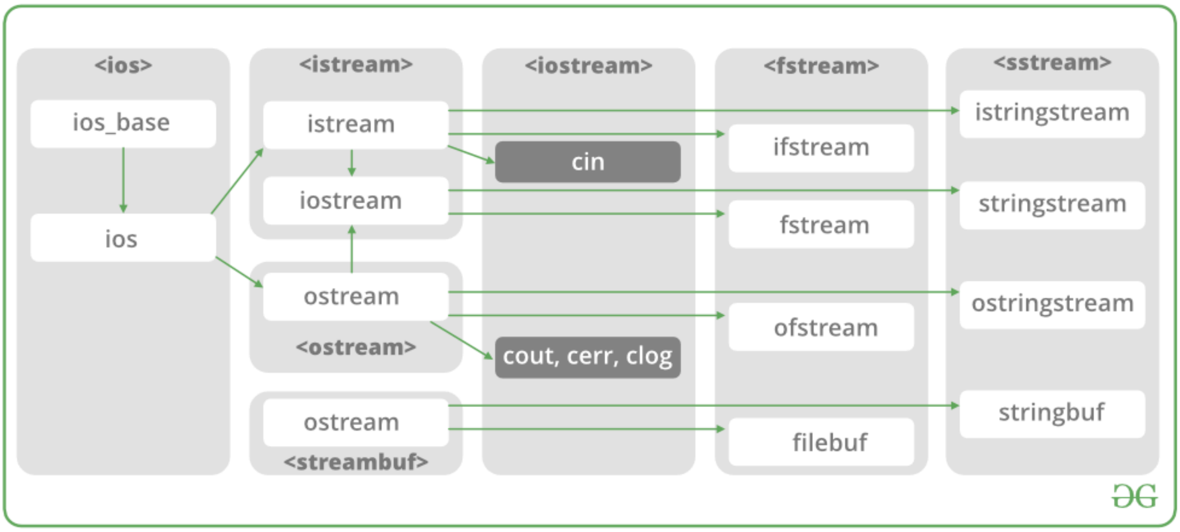
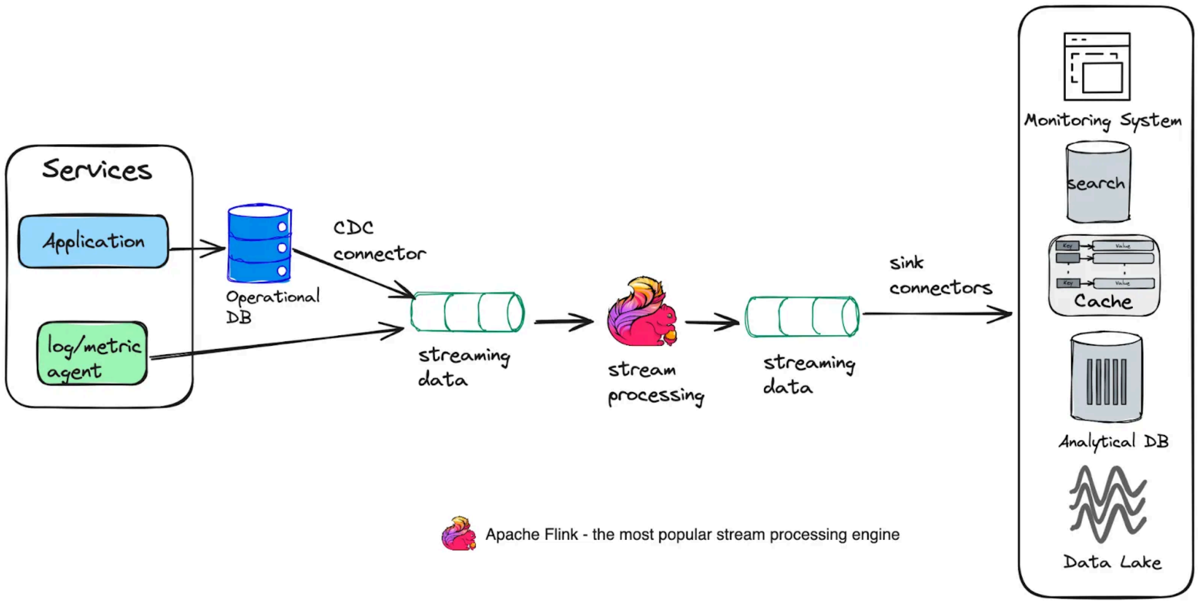
Streams and File Handling



✅ 1. Introduction to Streams in C++



Apache Flink - the most popular stream processing engine

What Are Streams?

A **stream** is a sequence of **bytes** used for **reading (input)** or **writing (output)** data. It acts as a medium between the program and data sources like **keyboard, files, network sockets, and memory buffers**.

💡 **Think of a stream like a pipeline**:

- **Input Stream (`istream`)** → Brings data into the program (e.g., `cin`, file reading).
- **Output Stream (`ostream`)** → Sends data out of the program (e.g., `cout`, file writing).

Stream Type	Description	Example
Input Stream	Data flows **into** the program	Reading from a file or keyboard
Output Stream	Data flows **out of** the program	Writing to console or file

✅ 1. Files and Streams in C++

C++ provides file handling support via **fstream** library:

```
#include <fstream>
```

Three classes are provided:

Class	Description
ifstream	Input file stream (read from files)
ofstream	Output file stream (write to files)
fstream	Input + Output (read and write)

✅ 2. How Do Streams Work?

Streams **buffer** the data, meaning they store it temporarily before reading/writing.

🔶 Steps of Stream Operation

1. **Open the stream** (connect to a source/destination like a file, keyboard, etc.).
2. **Perform the read/write operation.**
3. **Close the stream** (release resources).

Example of Stream Operation:

```
ofstream fout("data.txt");   // Creates a stream 'fout' linked to file
fout << "Hello, World!";     // Writes data to file
fout.close();                // Closes the stream
```

✅ 3. Input/Output Stream Classes

📌 Hierarchy of IOStream classes

```
        ios
       /   \
  istream    ostream
     |          |
 ifstream    ofstream
      \        /
       fstream
```

File Modes

When opening a file in C++, we use different file open modes to specify the kind of operations we want to perform on the file. These modes are provided by the `fstream` class and can be used with `ifstream`, `ofstream`, or `fstream`. Below is a list of all file modes and examples of how they can be used:

---

1. `ios::in` (Input Mode)

- **Purpose**: Open file for reading.
- **Example**:

  ```
  ifstream file("example.txt", ios::binary)  // Open file for reading
  if (file.is_open()) {
      string line;
      while (getline(file, line)) {
          cout << line << endl;
      }
  ```

```
    file.close();
}
```

2. `ios::out` (Output Mode)

- **Purpose**: Open file for writing.
- **Example**:

```
ofstream file("example.txt", ios::out | ios::binary);  // Open file for writing
if (file.is_open()) {
    file << "Hello, world!" << endl;
    file.close();
}
```

3. `ios::app` (Append Mode)

- **Purpose**: Open file for appending (writing at the end).
- **Example**:

```
ofstream file("example.txt", ios::app);  // Open file for appending
if (file.is_open()) {
    file << "Appending text to the file." << endl;
    file.close();
}
```

4. `ios::binary` (Binary Mode)

- **Purpose**: Open file in binary mode (not text).
- **Example**:

```
ofstream file("example.bin", ios::binary);  // Open file in binary mode
int num = 12345;
file.write(reinterpret_cast<char*>(&num), sizeof(num));
file.close();
```

5. `ios::trunc` (Truncate Mode)

- **Purpose**: If the file already exists, it will be truncated to zero length. If the file does not exist, it will be created.
- **Example**:

```
ofstream file("example.txt", ios::trunc);  // Truncate the file if it exists
file << "This is a new text." << endl;
file.close();
```

6. `ios::ate` (At End Mode)

- **Purpose**: Open file and move the write pointer to the end. Useful when opening a file for both reading and writing but starting at the end of the file.
- **Example**:

```
ofstream file("example.txt", ios::ate);  // Open file and move to end
file << "This text will be appended." << endl;
file.close();
```

7. `ios::in | ios::out` (Reading and Writing)

- **Purpose**: Open file for both reading and writing. The file must exist.
- **Example**:

```
fstream file("example.txt", ios::in | ios::out);  // Open file for both reading and writing
if (file.is_open()) {
    string content;
    file >> content;
    cout << "Content read from file: " << content << endl;
    file.seekp(0, ios::beg);  // Move pointer to the beginning for writing
    file << "Updated content" << endl;
    file.close();
}
```

8. `ios::in | ios::app` (Reading and Appending)

- **Purpose**: Open file for reading and appending (writing at the end).
- **Example**:

```
fstream file("example.txt", ios::in | ios::app);  // Open file for reading and appending
if (file.is_open()) {
    string line;
    while (getline(file, line)) {
        cout << line << endl;
    }
    file << "New line added at the end." << endl;
    file.close();
}
```

9. `ios::out | ios::binary` (Writing in Binary)

- **Purpose**: Open file for writing in binary mode. If the file exists, it will be truncated.
- **Example**:

```
ofstream file("example.bin", ios::out | ios::binary);  // Open file for writing in binary mode
int num = 12345;
file.write(reinterpret_cast<char*>(&num), sizeof(num));
file.close();
```

10. `ios::in | ios::binary` (Reading in Binary Mode)

- **Purpose**: Open file for reading in binary mode.
- **Example**:

```
ifstream file("example.bin", ios::in | ios::binary);  // Open file for reading in binary mode
if (file.is_open()) {
    int num;
    file.read(reinterpret_cast<char*>(&num), sizeof(num));
    cout << "Read number: " << num << endl;
    file.close();
}
```

11. `ios::in | ios::out | ios::binary` (Reading and Writing in Binary Mode)

- **Purpose**: Open file for both reading and writing in binary mode.
- **Example**:

```
fstream file("example.bin", ios::in | ios::out | ios::binary);  // Open file for reading and writing in binary mode
if (file.is_open()) {
    int num;
    file.read(reinterpret_cast<char*>(&num), sizeof(num));  // Read binary data
    cout << "Read number: " << num << endl;

    num = 67890;
    file.seekp(0, ios::beg);  // Move pointer to the beginning
    file.write(reinterpret_cast<char*>(&num), sizeof(num));  // Write binary data
    file.close();
}
```

12. `ios::in | ios::out | ios::app` (Reading, Writing, and Appending)

- **Purpose**: Open file for both reading, writing, and appending.
- **Example**:

```
fstream file("example.txt", ios::in | ios::out | ios::app);  // Open file for reading, writing, and appending
if (file.is_open()) {
    string content;
    file >> content;
    cout << "Content read: " << content << endl;
    file.seekp(0, ios::end);  // Move to end for appending
    file << "Appended text" << endl;
    file.close();
}
```

Functions and Operations

File handling in C++ is performed using file streams (`ifstream`, `ofstream`, and `fstream`). These classes offer several functions to handle files efficiently. Here's a list of all the commonly used file handling functions:

1. File Opening Functions

- **`open()`**: Opens a file in a specified mode.
  - Syntax: `fileStream.open("filename", mode)`
  - Modes:
    - `ios::in`: Open for reading.
    - `ios::out`: Open for writing.
    - `ios::app`: Open for appending (writing at the end).
    - `ios::trunc`: Truncate the file (erase contents before writing).
    - `ios::binary`: Open in binary mode.

  Example:

  ```
  ifstream inFile;
  inFile.open("data.txt", ios::in);
  ```

2. File Status Checking Functions

- **`is_open()`**: Checks if the file is open.
  - Syntax: `fileStream.is_open()`
  - Returns `true` if the file is successfully opened, otherwise `false`.

  Example:

```
if (!inFile.is_open()) {
    cout << "Error opening the file!" << endl;
}
```

- `fail()`: Checks if the last file operation failed.
  - Syntax: `fileStream.fail()`
  - Returns `true` if the last I/O operation failed.

  Example:

  ```
  if (inFile.fail()) {
      cout << "Error reading the file!" << endl;
  }
  ```

- `eof()`: Checks if the end of the file is reached.
  - Syntax: `fileStream.eof()`
  - Returns `true` if the end of the file has been reached.

  Example:

  ```
  if (inFile.eof()) {
      cout << "End of file reached." << endl;
  }
  ```

- `bad()`: Checks for serious errors such as a hardware failure.
  - Syntax: `fileStream.bad()`
  - Returns `true` if a serious error occurred.

  Example:

  ```
  if (inFile.bad()) {
      cout << "A serious error occurred!" << endl;
  }
  ```

- `good()`: Checks if the file stream is in a good state.
  - Syntax: `fileStream.good()`

    - Returns `true` if the stream is in a good state.

Example:

```
if (inFile.good()) {
    cout << "The file is in a good state!" << endl;
}
```

3. File Reading Functions

- **`getline()`**: Reads a line from the file into a string.
    - Syntax: `getline(fileStream, stringVariable)`
    - Used to read an entire line from the file.

Example:

```
string line;
getline(inFile, line);
```

- **`get()`**: Reads one character from the file.
    - Syntax: `fileStream.get()`
    - Returns the next character from the file or EOF if the end of the file is reached.

Example:

```
char ch;
inFile.get(ch);
```

- **`read()`**: Reads binary data from the file into a buffer.
    - Syntax: `fileStream.read(buffer, size)`
    - Reads `size` bytes from the file into the buffer.

Example:

```
char buffer[100];
inFile.read(buffer, sizeof(buffer));
```

4. File Writing Functions

- `write()`: Writes binary data to the file.
  - Syntax: `fileStream.write(buffer, size)`
  - Writes `size` bytes from the buffer to the file.

  Example:

  ```
  char buffer[] = "Hello, World!";
  outFile.write(buffer, sizeof(buffer));
  ```

- `put()`: Writes a single character to the file.
  - Syntax: `fileStream.put(char)`
  - Writes a single character to the file.

  Example:

  ```
  outFile.put('A');
  ```

- `<<` **(stream insertion operator)**: Used to write data to the file (mostly for text).
  - Syntax: `fileStream << data`
  - Used to write formatted text to the file.

  Example:

  ```
  outFile << "Hello, World!" << endl;
  ```

---

5. File Closing Function

- `close()`: Closes the file stream.
  - Syntax: `fileStream.close()`
  - Closes the file and releases any resources associated with it.

  Example:

  ```
  inFile.close();
  ```

```
outFile.close();
```

6. Seek Functions

- **`seekg()`**: Moves the read pointer to a specific position in the file (for input).
    - Syntax: `fileStream.seekg(position, direction)`
    - `position`: The position to move the pointer to.
    - `direction`: Can be `ios::beg`, `ios::cur`, or `ios::end` to specify relative to beginning, current position, or end.

  Example:

  ```
  inFile.seekg(0, ios::beg);  // Move to the beginning of the file
  ```

- **`seekp()`**: Moves the write pointer to a specific position in the file (for output).
    - Syntax: `fileStream.seekp(position, direction)`
    - `position` and `direction` work similarly as `seekg()`.

  Example:

  ```
  outFile.seekp(0, ios::beg);  // Move to the beginning of the file
  ```

- **`tellg()`**: Returns the current position of the read pointer.
    - Syntax: `fileStream.tellg()`
    - Returns the current position of the input pointer in the file.

  Example:

  ```
  cout << "Current position: " << inFile.tellg() << endl;
  ```

- **`tellp()`**: Returns the current position of the write pointer.
    - Syntax: `fileStream.tellp()`
    - Returns the current position of the output pointer in the file.

  Example:

```
cout << "Current position: " << outFile.tellp() << endl;
```

7. File Type Checking Functions

- `is_open()`: Checks if the file is open.
  - Syntax: `fileStream.is_open()`
  - Returns `true` if the file is open.

  Example:

  ```
  if (inFile.is_open()) {
      cout << "File is open!" << endl;
  }
  ```

- `flush()`: Forces the buffer to be written to the file.
  - Syntax: `fileStream.flush()`
  - Ensures that all buffered output is written to the file.

  Example:

  ```
  outFile.flush();
  ```

File Handling in C++: Binary Files

In C++, **file handling** allows reading from and writing to files. Binary files are different from text files in that they store data in raw, machine-readable format, unlike text files, which store data as human-readable text.

What is a Binary File?

- A **binary file** stores data in binary format (0s and 1s), making it more efficient for storing structured data like numbers, objects, or even images.
- Text files, on the other hand, store data in a human-readable format (ASCII/UTF-8 encoding).

Why Use Binary Files?

- **Efficiency**: Binary files are more compact and can be read and written faster, as they avoid the need for text encoding/decoding.
- **Exact Representation**: Binary files represent data exactly as it is in memory (like integers, floats, or structs), avoiding issues with formatting and conversions that may happen with text files.

How to Open a Binary File?

In C++, you use the `fstream` class to handle binary files. You can open a file in **binary mode** using `ios::binary`.

Opening Files in Binary Mode

- **Reading**: `ifstream file("filename", ios::binary);`
- **Writing**: `ofstream file("filename", ios::binary);`
- **Reading & Writing**: `fstream file("filename", ios::in | ios::out | ios::binary);`

Basic Operations

- **Write**: Store data in binary format.
- **Read**: Retrieve data from a binary file.

Important Functions for Binary File Handling

1. **Opening a Binary File**
    - **Syntax**: `fstream file("filename", ios::binary);`
    - **Example**:

    ```
    ofstream outFile("data.bin", ios::binary);
    if (!outFile) {
        cout << "Error opening file!" << endl;
        return;
    }
    ```

2. **Writing Data to a Binary File**
    - **Syntax**: `write()`
        - *write(const char buffer, size_t size)*

- **Example**:

  ```
  int num = 10;
  outFile.write(reinterpret_cast<char*>(&num), sizeof(num));
  ```

3. **Reading Data from a Binary File**
   - **Syntax**: `read()`
     - *read(char buffer, size_t size)*
   - **Example**:

     ```
     int num;
     inFile.read(reinterpret_cast<char*>(&num), sizeof(num));
     cout << "Read number: " << num << endl;
     ```

4. **Checking if File is Open**
   - **Syntax**: `is_open()`
   - **Example**:

     ```
     if (!file.is_open()) {
         cout << "Failed to open file!" << endl;
     }
     ```

5. **File Status Functions** (Checking after operations)
   - **eof()**: Check if end of file is reached.
   - **fail()**: Check if a file operation failed.
   - **good()**: Check if the file stream is in a good state.
   - **bad()**: Check if a serious error occurred.

Writing a Simple Struct to a Binary File

Example: Saving and Reading a `struct` to/from a Binary File

Let’s define a `struct` for storing a student's details and write it to a binary file.

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

struct Student {
    int id;
    char name[50];
    double grade;
};

int main() {
    // Create a Student object
    Student student = {1, "John Doe", 90.5};

    // Open the binary file for writing
    ofstream outFile("students.bin", ios::binary);
    if (!outFile) {
        cout << "Error opening file!" << endl;
        return -1;
    }

    // Write the student object to the file
    outFile.write(reinterpret_cast<char*>(&student), sizeof(student));

    // Close the file after writing
    outFile.close();

    // Open the binary file for reading
    ifstream inFile("students.bin", ios::binary);
    if (!inFile) {
        cout << "Error opening file!" << endl;
        return -1;
    }

    // Read the student object from the file
    Student readStudent;
    inFile.read(reinterpret_cast<char*>(&readStudent), sizeof(readStudent));
    // Display the data
    cout << "Student ID: " << readStudent.id << endl;
    cout << "Student Name: " << readStudent.name << endl;
    cout << "Student Grade: " << readStudent.grade << endl;

    // Close the file after reading
    inFile.close();

    return 0;
}
```

Output:

```
Student ID: 1
Student Name: John Doe
Student Grade: 90.5
```

Why Use `reinterpret_cast` for Binary Files?

- **Converting Between Types**: When writing and reading data to/from a binary file, you often need to cast between types (like from a `struct` to a `char*`) to store and retrieve raw data.
- `reinterpret_cast` allows you to treat a block of memory as a different type, which is what's needed for reading and writing complex data structures.

Reading and Writing Complex Objects (Classes) to Binary Files

You can also save and load class objects to binary files, but it's important to handle pointers, dynamic memory, and non-trivial constructors and destructors appropriately.

Example: Saving and Loading a Class Object

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

class Employee {
```

```
public:
    int id;
    double salary;

    Employee() : id(0), salary(0.0) {}
    Employee(int id, double salary) : id(id), salary(salary) {}

    // Function to display employee info
    void display() const {
        cout << "ID: " << id << ", Salary: " << salary << endl;
    }
};

int main() {
    Employee emp(1, 50000.0);

    // Writing the employee object to a binary file
    ofstream outFile("employee.bin", ios::binary);
    if (!outFile) {
        cout << "Error opening file!" << endl;
        return -1;
    }
    outFile.write(reinterpret_cast<char*>(&emp), sizeof(emp));
    outFile.close();

    // Reading the employee object back from the binary file
    Employee readEmp;
    ifstream inFile("employee.bin", ios::binary);
    if (!inFile) {
        cout << "Error opening file!" << endl;
        return -1;
    }
    inFile.read(reinterpret_cast<char*>(&readEmp), sizeof(readEmp));
    inFile.close();

    // Display the read employee info
    readEmp.display();
    return 0;
}
```

Output:

```
ID: 1, Salary: 50000
```

Student Task 1:

Console-Based Employee Management System using an **OOP approach** with:

✔ **Encapsulation** (Fully encapsulated `Employee` class)

✔ **Abstraction** (Abstract base class `EmployeeOperations`)

✔ **Modularity** (Separate `EmployeeManager` class for business logic)

✔ **Industry-Standard Practices**

🛠 Features Implemented

1️⃣ Add Employee

2️⃣ Get All Employees

3️⃣ Get Employee by ID

4️⃣ Update Employee Details by ID

5️⃣ Delete Employee by ID

6️⃣ Get Employees by Salary Range

7️⃣ Get Employees by Department

8️⃣ Get Employees by City

9️⃣ Exit

Solution

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <iomanip>
```

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

// ========================== Fully Encapsulated Employee Class
==========================
class Employee {
private:
    int id;
    char name[50];
    float salary;
    char department[30];
    char city[30];

public:
    // Constructor
    Employee() : id(0), salary(0.0f) {}

    // Getter & Setter Methods
    void setEmployee(int empId, const string &empName, float empSalary, c
onst string &dept, const string &empCity) {
        id = empId;
        strncpy(name, empName.c_str(), sizeof(name) - 1);
        salary = empSalary;
        strncpy(department, dept.c_str(), sizeof(department) - 1);
        strncpy(city, empCity.c_str(), sizeof(city) - 1);
    }

    int getId() const { return id; }
    float getSalary() const { return salary; }
    string getDepartment() const { return string(department); }
    string getCity() const { return string(city); }

    void display() const {
        cout << left << setw(10) << id << setw(20) << name << setw(10) << s
alary << setw(15) << department << setw(15) << city << endl;
    }

    // Read/Write functions for binary file
    void writeToFile(ofstream &out) const {
        out.write(reinterpret_cast<const char *>(this), sizeof(Employee));
    }

    void readFromFile(ifstream &in) {
        in.read(reinterpret_cast<char *>(this), sizeof(Employee));
    }
};

// ========================== Abstract Class for Employee Operations ==========================
class EmployeeOperations {
public:
    virtual void addEmployee() = 0;
    virtual void getAllEmployees() = 0;
    virtual void getEmployeeById(int empId) = 0;
    virtual void updateEmployeeById(int empId) = 0;
    virtual void deleteEmployeeById(int empId) = 0;
    virtual void getEmployeesBySalaryRange(float minSalary, float maxSalary) = 0;
    virtual void getEmployeesByDepartment(const string &dept) = 0;
    virtual void getEmployeesByCity(const string &city) = 0;
};

// ========================== EmployeeManager (Abstract Implementation) ==========================
class EmployeeManager : public EmployeeOperations {
private:
    const string fileName = "employees.dat";

public:
    void addEmployee() override {
        Employee emp;
        int id;
        string name, dept, city;
        float salary;

        cout << "Enter Employee ID: ";
        cin >> id;
        cin.ignore();
        cout << "Enter Name: ";
        getline(cin, name);
        cout << "Enter Salary: ";
        cin >> salary;
        cin.ignore();
        cout << "Enter Department: ";
        getline(cin, dept);
        cout << "Enter City: ";
        getline(cin, city);

        emp.setEmployee(id, name, salary, dept, city);

        ofstream outFile(fileName, ios::binary | ios::app);
        if (outFile) {
            emp.writeToFile(outFile);
            cout << "Employee added successfully.\n";
        } else {
            cout << "Error opening file.\n";
        }
        outFile.close();
    }

    void getAllEmployees() override {
        ifstream inFile(fileName, ios::binary);
        if (!inFile) {
            cout << "No records found.\n";
            return;
        }
        Employee emp;
        cout << left << setw(10) << "ID" << setw(20) << "Name" << setw(10)
<< "Salary" << setw(15) << "Department" << setw(15) << "City" << endl;
        cout << string(70, '-') << endl;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
{
            emp.display();
        }
        inFile.close();
    }

    void getEmployeeById(int empId) override {
        ifstream inFile(fileName, ios::binary);
        if (!inFile) {
            cout << "No records found.\n";
            return;
        }
        Employee emp;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
{
            if (emp.getId() == empId) {
                emp.display();
                return;
            }
        }
        cout << "Employee not found.\n";
        inFile.close();
    }

    void updateEmployeeById(int empId) override {
        fstream file(fileName, ios::binary | ios::in | ios::out);
        if (!file) {
            cout << "No records found.\n";
            return;
        }
        Employee emp;
        while (file.read(reinterpret_cast<char *>(&emp), sizeof(Employee))) {
            if (emp.getId() == empId) {
                cout << "Enter new Salary: ";
                float newSalary;
                cin >> newSalary;
                emp.setEmployee(emp.getId(), emp.getDepartment(), newSalary,
emp.getDepartment(), emp.getCity());

                file.seekp(-static_cast<int>(sizeof(Employee)), ios::cur);
                file.write(reinterpret_cast<char *>(&emp), sizeof(Employee));
                cout << "Employee updated successfully.\n";
                file.close();
                return;
            }
        }
        cout << "Employee not found.\n";
        file.close();
    }

    void deleteEmployeeById(int empId) override {
        ifstream inFile(fileName, ios::binary);
        ofstream tempFile("temp.dat", ios::binary);
        Employee emp;
        bool found = false;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
{
            if (emp.getId() == empId) {
                found = true;
            } else {
                emp.writeToFile(tempFile);
            }
        }
        inFile.close();
        tempFile.close();
        remove(fileName.c_str());
        rename("temp.dat", fileName.c_str());

        if (found)
            cout << "Employee deleted successfully.\n";
        else
            cout << "Employee not found.\n";
    }

    void getEmployeesBySalaryRange(float minSalary, float maxSalary) overr
ide {
        ifstream inFile(fileName, ios::binary);
        Employee emp;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
```

```
{
            if (emp.getSalary() >= minSalary && emp.getSalary() <= maxSalary)
                emp.display();
        }
        inFile.close();
    }

    void getEmployeesByDepartment(const string &dept) override {
        ifstream inFile(fileName, ios::binary);
        Employee emp;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
{
            if (emp.getDepartment() == dept)
                emp.display();
        }
        inFile.close();
    }

    void getEmployeesByCity(const string &city) override {
        ifstream inFile(fileName, ios::binary);
        Employee emp;
        while (inFile.read(reinterpret_cast<char *>(&emp), sizeof(Employee)))
{
            if (emp.getCity() == city)
                emp.display();
        }
        inFile.close();
    }
};

// ========================== UI Class (Main Menu) ==========
================
int main() {
    EmployeeManager manager;
    int choice, id;
    float minSalary, maxSalary;
    string dept, city;
    do {
        cout << "\nEmployee Management System\n";
        cout << "1. Add Employee\n2. Get All Employees\n3. Get Employee by ID\n4. Update Employee\n5. Delete Employee\n";
        cout << "6. Get Employees by Salary Range\n7. Get Employees by Department\n8. Get Employees by City\n9. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;

        switch (choice) {
            case 1: manager.addEmployee(); break;
            case 2: manager.getAllEmployees(); break;
            case 3: cout << "Enter ID: "; cin >> id; manager.getEmployeeById(id); break;
            case 4: cout << "Enter ID: "; cin >> id; manager.updateEmployeeById(id); break;
            case 5: cout << "Enter ID: "; cin >> id; manager.deleteEmployeeById(id); break;
            case 6: cout << "Enter Salary Range: "; cin >> minSalary >> maxSalary; manager.getEmployeesBySalaryRange(minSalary, maxSalary); break;
            case 7: cout << "Enter Department: "; cin >> dept; manager.getEmployeesByDepartment(dept); break;
            case 8: cout << "Enter City: "; cin >> city; manager.getEmployeesByCity(city); break;
        }
    } while (choice != 9);
}
```

Advantages of Using Binary Files

- **Faster**: Binary files are more compact, hence faster to read and write.
- **Exact Storage**: They store data in the exact format as it's held in memory, meaning no conversion is needed (e.g., for integers, floats, or structures).
- **Efficient**: Especially useful for storing large amounts of data like images, databases, etc.

Common Pitfalls

- **Platform Dependence**: Binary files can be platform-dependent due to differences in how different systems store data (e.g., endianess).
- **Human Unreadable**: Unlike text files, binary files are not human-readable.
- **Complexity**: Writing and reading binary files may require additional work, such as managing memory properly.