DAY – 15 Assignment and LABs

// Files contains codes on - > File Handling, File Handli.(Assignment - 1), Exception Handling , Exception Handling (Assignment - 2) .

// Code - 1 File Handling

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

class Student {
public:
    struct stu {
        char name[20];
        int roll;
    } s;

    void put_data();
    void get_data();
};

void Student::put_data() {
    ofstream outfile("hit.txt", ios::app | ios::binary);
    if (!outfile) {
        cerr << "Error opening file for writing" << endl;
        return;
    }
    cout << "Enter student name: ";
    cin >> s.name;
    cout << "Enter student roll number: ";
    cin >> s.roll;
    outfile.write(reinterpret_cast<char*>(&s), sizeof(s));
    outfile.close();
}

void Student::get_data() {
    int temp;
    cout << "Enter roll no.: ";
    cin >> temp;
    ifstream file("hit.txt", ios::in | ios::binary);
    if (!file) {
        cerr << "Error opening file for reading" << endl;
        return;
    }
    file.seekg(0, ios::beg);
    bool found = false;
    while (file.read(reinterpret_cast<char*>(&s), sizeof(s))) {
        if (temp == s.roll) {
            cout << "Student name is: " << s.name << "\n";
            cout << "Student roll no is: " << s.roll << endl;
            found = true;
            break;
        }
    }
    if (!found) {
        cout << "Student with roll no " << temp << " not found." << endl;
    }
    file.close();
}

int main() {
    Student student;
    student.get_data();

    return 0;
} */
```

// Code - 2 Implementation of code on File Handling

```
/* #include <fstream>  // Header for file stream operations.
#include <iostream>
#include <string>
using namespace std;

// Function to create a text file
```

```
void createTextFile(const string& filename) {
  ofstream outfile(filename);

  if (outfile.is_open()) {   // Checks if the file is successfully opened using outfile.is_open().
    outfile << "This is a sample text file.\n";
    outfile << "You can add more content here.\n";
    cout << "Text file " << filename << " created successfully!" << endl;
  } else {
    cerr << "Error creating file: " << filename << endl;
  }
  outfile.close(); // Close the file even on errors
}

// Function to read from a text file
void readTextFile(const string& filename) {
  ifstream infile(filename);

  if (infile.is_open()) {   // Checks if the file is successfully opened using infile.is_open().
    string line;
    while (getline(infile, line)) {
      cout << line << endl;
    }
  } else {
    cerr << "Error opening file: " << filename << endl;
  }
  infile.close(); // Close the file even on errors
}
// Function to write to a binary file
void writeBinaryFile(const string& filename, const char* data, int size) {
  ofstream outfile(filename, ios::binary);

  if (outfile.is_open()) {
    outfile.write(data, size);
    cout << "Binary data written to file " << filename << endl;
  } else {
    cerr << "Error creating binary file: " << filename << endl;
  }

  outfile.close(); // Close the file even on errors
}

// Function to read from a binary file
void readBinaryFile(const string& filename, int size) {
  char buffer[size];

  ifstream infile(filename, ios::binary);

  if (infile.is_open()) {
    infile.read(buffer, size);
    cout << "Binary data from file " << filename << ":" << endl;

    for (int i = 0; i < size; ++i) {

      cout << hex << static_cast<int>(buffer[i]) << " ";

    }

    cout << endl;

  } else {

    cerr << "Error opening binary file: " << filename << endl;

  }


  infile.close(); // Close the file even on errors

}


int main() {

  string textFilename = "example.txt";

  string binaryFilename = "data.bin";


  // Create a text file

  createTextFile(textFilename);


  // Read from the text file

  readTextFile(textFilename);


  // Sample data for binary file

  char binaryData[] = "This is binary data";
  // Write to a binary file
  writeBinaryFile(binaryFilename, binaryData, sizeof(binaryData));

  // Read from the binary file (adjust size based on written data)
  readBinaryFile(binaryFilename, sizeof(binaryData));

  return 0;
} */
```

// Assignment 1 : Implement the following problem statements tasks.

/* A. File Handling Practice Problems

This set of problems will help you practice the concepts of file handling in C++ covered in the provided code.

1. Text Files:

a. Student Records: Create a program that allows users to enter student information (name, ID, marks) and store them in a text file. The program should allow users to:

Add new student records.

Display all student records from the file.

Search for a specific student by ID and display their details. */

```
#include <iostream>
#include <fstream> // The ifstream and ofstream are both classes in C++ provided by the <fstream>
#include <string>
```

```
using namespace std;

struct Student {
    string name;
    int id;
    float marks;
};

void addStudentRecord() {
    ofstream outFile("student_records.txt", ios::app);
    if (!outFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Student newStudent;
    cout << "Enter name: ";
    getline(cin, newStudent.name);
    cout << "Enter ID: ";
    cin >> newStudent.id;
    cout << "Enter marks: ";
    cin >> newStudent.marks;

    outFile << newStudent.name << " " << newStudent.id << " " << newStudent.marks << endl;
    outFile.close();
}

void displayAllStudentRecords() {
    ifstream inFile("student_records.txt"); // When we use to open file for reading we use ifstream
    if (!inFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Student student;
    while (inFile >> student.name >> student.id >> student.marks) {
        cout << "Name: " << student.name << ", ID: " << student.id << ", Marks: " << student.marks << endl;
    }

    inFile.close();
}

void searchStudentByID(int searchID) {
    ifstream inFile("student_records.txt");
    if (!inFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Student student;
    bool found = false;
    while (inFile >> student.name >> student.id >> student.marks) {
        if (student.id == searchID) {
            cout << "Name: " << student.name << ", ID: " << student.id << ", Marks: " << student.marks << endl;
            found = true;
            break;
        }
    }

    if (!found) {
        cout << "Student with ID " << searchID << " not found." << endl;
    }

    inFile.close();
}

int main() {
    int choice;
    int searchID;

    do {
```

```
cout << "\nMenu:\n";
cout << "1. Add new student record\n";
cout << "2. Display all student records\n";
cout << "3. Search for student by ID\n";
cout << "4. Exit\n";

cout << "Enter your choice: ";
cin >> choice;
cin.ignore(); // Ignore newline character left by cin

switch (choice) {
    case 1:
        addStudentRecord();
        break;
    case 2:
        displayAllStudentRecords();
        break;
    case 3:
        cout << "Enter ID to search: ";
        cin >> searchID;
        searchStudentByID(searchID);
        break;
    case 4:
        cout << "Exiting program.\n";
        break;
```

```
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 4);

    return 0;
}
```

/* b. Phonebook:

Develop a program that functions as a simple phonebook. Users can:

Add new contacts (name, phone number) to the file.

Search for a contact by name and display their phone number. */

```
/* #include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct Contact {
    string name;
    string phoneNumber;
};
```

```
void addContact() {
    ofstream outFile("phonebook.txt", ios::app);
    if (!outFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Contact newContact;
    cout << "Enter name: ";
    getline(cin, newContact.name);
    cout << "Enter phone number: ";
    getline(cin, newContact.phoneNumber);

    outFile << newContact.name << " " << newContact.phoneNumber << endl;

    outFile.close();
}

void searchContactByName(const string& searchName) {
    ifstream inFile("phonebook.txt");
    if (!inFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }
    Contact contact;
    bool found = false;
    while (inFile >> contact.name >> contact.phoneNumber) {
        if (contact.name == searchName) {
            cout << "Name: " << contact.name << ", Phone Number: " << contact.phoneNumber <<
endl;
            found = true;
            break;
        }
    }

    if (!found) {
        cout << "Contact with name \"" << searchName << "\" not found." << endl;
    }

    inFile.close();
}

int main() {
    int choice;
    string searchName;

    do {
        cout << "\nMenu:\n";
        cout << "1. Add new contact\n";
        cout << "2. Search for contact by name\n";
        cout << "3. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // Ignore newline character left by cin

        switch (choice) {
            case 1:
                addContact();
                break;
            case 2:
                cout << "Enter name to search: ";
                getline(cin, searchName);
                searchContactByName(searchName);
                break;
            case 3:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 3);

    return 0;
```

```
} */
```

/*File Encryption/Decryption (Optional): Implement a program that encrypts/decrypts a text file using a simple Caesar cipher or another basic encryption method. */

/* 2. Binary Files:

// a. Image Copy: Write a program that copies the contents of an image file (e.g., JPG, PNG) to a new file. Ensure you handle binary data correctly. */

```
/* #include <iostream>
#include <fstream>

using namespace std;

void copyImage(const string& sourceFile, const string& destFile) {
    ifstream inFile(sourceFile, ios::binary);
    if (!inFile) {
        cerr << "Error: Unable to open source image file." << endl;
        return;
    }

    ofstream outFile(destFile, ios::binary);
    if (!outFile) {
        cerr << "Error: Unable to create or open destination image file." << endl;
        inFile.close();
        return;
    }

    // Copy contents from source to destination
    outFile << inFile.rdbuf();

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

    cout << "Image copied successfully." << endl;
}

int main() {
    string sourceFile, destFile;

    cout << "Enter source image file name: ";
    getline(cin, sourceFile);

    cout << "Enter destination image file name: ";
    getline(cin, destFile);

    copyImage(sourceFile, destFile);

    return 0;
```

```
} */
```

/* b. Inventory Management:

Develop a program that manages a store inventory. Users can:

Add new items (name, price, quantity) to a binary file.

Display all items from the inventory.

Update the quantity of an existing item.*/

/* High Score Tracking (Optional): Create a program that keeps track of high scores for a game. Users can:

Save a new high score to a binary file.

Display the current high score. */

```
/* #include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct Item {
    string name;
    float price;
    int quantity;
```

```
};


void addItem() {
    ofstream outFile("inventory.bin", ios::binary | ios::app);
    if (!outFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }


    Item newItem;
    cout << "Enter item name: ";
    getline(cin, newItem.name);
    cout << "Enter price: ";
    cin >> newItem.price;
    cout << "Enter quantity: ";
    cin >> newItem.quantity;


    outFile.write(reinterpret_cast<const char*>(&newItem), sizeof(newItem));


    outFile.close();
}
void displayInventory() {
    ifstream inFile("inventory.bin", ios::binary);
    if (!inFile) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Item item;
    while (inFile.read(reinterpret_cast<char*>(&item), sizeof(item))) {
        cout << "Name: " << item.name << ", Price: " << item.price << ", Quantity: " << item.quantity << endl;
    }

    inFile.close();
}

void updateItemQuantity(const string& itemName, int newQuantity) {
    fstream file("inventory.bin", ios::binary | ios::in | ios::out);
    if (!file) {
        cerr << "Error: Unable to open file." << endl;
        return;
    }

    Item item;
    bool found = false;
    while (file.read(reinterpret_cast<char*>(&item), sizeof(item))) {
        if (item.name == itemName) {
            // Update quantity
            item.quantity = newQuantity;
            // Move file pointer back to update record
            file.seekp(file.tellg() - sizeof(item));
            file.write(reinterpret_cast<const char*>(&item), sizeof(item));
            found = true;
            break;
        }
    }

    if (found) {
        cout << "Item quantity updated successfully." << endl;
    } else {
        cout << "Item \"" << itemName << "\" not found." << endl;
    }

    file.close();
}

int main() {
    int choice;
    string itemName;
    int newQuantity;

    do {
        cout << "\nMenu:\n";
        cout << "1. Add new item\n";

        cout << "2. Display all items\n";

        cout << "3. Update item quantity\n";

        cout << "4. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        cin.ignore(); // Ignore newline character left by cin


        switch (choice) {

            case 1:

                addItem();

                break;

            case 2:

                displayInventory();

                break;

            case 3:

                cout << "Enter item name to update quantity: ";

                getline(cin, itemName);

                cout << "Enter new quantity: ";

                cin >> newQuantity;

                updateItemQuantity(itemName, newQuantity);

                break;

            case 4:

                cout << "Exiting program.\n";

                break;
```

```
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 4);

    return 0;
} */
```

// Concept on (* Exception Handling ***)**

// In c++ we use three keywords to perform exception Handling

/* try , catch, throw

All the exception classes in C++ are derived from the std:"exception class.

std:: exception

std::logic_failure etc.*/

// Code - 1   Implementation of Program on Exception Handling

```
/* #include<iostream>
using namespace std;

float add(int x,int y) {
    return (x + y);
}
```

```
    //return (x/y);
float sub(int x,int y) {
    if (x > y){
    return (x - y);
}else
    return (y - x) ;
}
int mult(int x,int y) {
    return x*y;  // 1/0 = undefined and 0/1 = 0
}

float division(int x,int y) {
    if( y == 0) {
        throw " Attempted to divide by zero";
    }
    return (x/y);  // 1/0 = undefined and 0/1 = 0
}

int main(){
    int i = 25;
    int j = 0;
    float result1,result2,result3,result4 ;

    result1 = add(i,j);
    cout<<result1<<endl;
    result2= sub(i,j);
    cout<<result2<<endl;

    result3= mult(i,j);
    cout<<mult<<endl;

    try {
        result4 = division (i,j);
        cout<<result4<<endl;
    }
    catch (const char* e) {
        cerr <<e<< endl;
    }

    return 0;
} */
```

// Code – 2 On Calculator using Exception Handling

```
/* #include <iostream>
#include<stdexcept>
using namespace std;

class Calculator {
public:
    double add(double num1, double num2) {
        return num1 + num2;
    }

    double subtract(double num1, double num2) {
        return num1 - num2;
    }

    double multiply(double num1, double num2) {
        return num1 * num2;
    }

    double divide(double num1, double num2) {
        if (num2 == 0) {
            throw runtime_error("Cannot divide by zero!");
        }
        return num1 / num2;
    }
};

int main() {
    Calculator calc;

    try {
        double num1, num2;
        cout << "Enter the first number: ";
        cin >> num1;
        cout << "Enter the second number: ";
        cin >> num2;

        cout << "Result: " << calc.add(num1, num2) <<endl;
        cout << "Result: " << calc.subtract(num1, num2) <<endl;
        cout << "Result: " << calc.multiply(num1, num2) << endl;
        cout << "Result: " << calc.divide(num1, num2) << endl;
    } catch (runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
        return 1;
    }

    return 0;
} */
```

// Code – 3   User defined Exception

```
/* #include<iostream>
#include<exception>
using namespace std;

class MyException : public exception{
    public :
```

```
    const char * what()const throw()
    {
        return " Attempted to divide by zero\n";
    }
};
int main(){
    try{
        int x,y;
        cout<< " Enter the two numbers : \n";
        cin>>x>>y;
        if (y == 0)
        {

            MyException z;
            throw z;
        }

    else {
            cout << "x / y = " << x / y << endl;
        }
    } catch (const MyException& e) {
        cerr << "Error: " << e.what() << endl;
    }

    return 0;
```

} */

// Assignment - 2

// Questions:

// 1. What are the advantages and disadvantages of using exceptions in C++ compared to traditional error codes?

/*1. Advantages of using exceptions in C++ compared to traditional error codes:

- Exceptions allow for more robust error handling, as they can propagate up the call stack and be caught by a handler, whereas error codes must be explicitly checked and handled at each level.

- Exceptions can provide more information about the error, such as a message or a stack trace, whereas error codes are often just a simple integer value.

- Exceptions can be used to handle errors in a more centralized way, whereas error codes require each function to handle errors individually.

Disadvantages of using exceptions in C++ compared to traditional error codes:

- Exceptions can be slower and more resource-intensive than error codes, as they require the creation of an exception object and the unwinding of the stack.

- Exceptions can be more difficult to use correctly, as they require a good understanding of the language and the libraries being used.

- Exceptions can make the code more complex and harder to read, as they require additional try-catch blocks and error handling logic. */

// 2. How can you ensure that exception classes provide informative error messages for debugging?

/* We can ensure that exception classes provide informative error messages for debugging, you can :

- Use a descriptive error message that includes information about the error, such as the function that failed and the reason for the failure.

- Include additional information, such as a stack trace or a error code, to help diagnose the problem.

- Use a logging mechanism to log the error message and other relevant information, so that it can be reviewed later.

- Use a centralized error handling mechanism, such as a global error handler, to catch and handle exceptions in a consistent way.

// 3. Discuss strategies for optimizing exception handling performance, especially in performance-critical applications.

1. We can use different strategies for optimizing exception handling performance:

- Use exceptions only for exceptional circumstances, and use error codes or other mechanisms for expected errors.

- Use a lightweight exception class that contains only the necessary information, rather than a heavy-weight class that contains a lot of unnecessary data. */

// 4. How can you design a hierarchy of exception classes for improved code maintainability and reusability?

/* We can design a hierarchy of exception classes for improved code maintainability and reusability:

- Use a base exception class that provides a common interface for all exceptions, and derive specific exception classes from it.

- Use a consistent naming convention for exception classes, such as "Exception" or "Error", to make them easy to recognize.

- Use a consistent set of methods for exception classes, such as "what()" and "why()", to provide a consistent interface for error handling.

- Use a centralized error handling mechanism, such as a global error handler, to catch and handle exceptions in a consistent way. */

// 5. When might it be appropriate to not use exceptions in C++ for error handling? Explain your reasoning.

/* It may be appropriate to not use exceptions in C++ for error handling in the following situations:

- When the error is expected and can be handled locally, such as when a function returns an error code.

- When the error is not severe and can be ignored, such as when a function fails to perform an optional task.

- When the error handling mechanism is already provided by the language or library, such as when using a library that provides its own error handling mechanism.

- When the code is performance-critical and exceptions would introduce too much overhead.

- When the code is simple and exceptions would add unnecessary complexity. */

// A. Develop a C++ program that demonstrates robust exception handling for file operations.

// The program should:

// 1. Read data from a text file.

// 2. Validate the data format (e.g., expecting specific number of values per line).

// 3. Perform calculations based on the valid data.

```
/* #include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <exception>


using namespace std;


class FileException : public exception {
```

```
public:
    const char* what() const throw() {
        return "File operation error";
    }
};

class DataFormatException : public exception {
public:
    const char* what() const throw() {
        return "Data format error";
    }
};

// Function to read data from a file
vector<vector<int>> readDataFromFile(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        throw FileException();
    }

    vector<vector<int>> data;
    string line;
    while (getline(file, line)) {
        istringstream iss(line);
        vector<int> values;
        int value;
        while (iss >> value) {
            values.push_back(value);
        }
        // Assuming we expect exactly 3 values per line for validation
        if (values.size() != 3) {
            throw DataFormatException();
        }
        data.push_back(values);
    }
    file.close();
    return data;
}
// Function to perform calculations on the data
int performCalculations(const vector<vector<int>>& data) {
    int sum = 0;
    for (const auto& row : data) {
        for (int value : row) {
            sum += value;
        }
    }
    return sum;
}
int main() {
    try {
        string filename = "data.txt";

        vector<vector<int>> data = readDataFromFile(filename);

        int result = performCalculations(data);

        cout << "The sum of all values is: " << result << endl;

    } catch (const FileException& e) {

        cerr << "Error: " << e.what() << endl;

    } catch (const DataFormatException& e) {

        cerr << "Error: " << e.what() << endl;

    } catch (const exception& e) {

        cerr << "An unexpected error occurred: " << e.what() << endl;

    }


    return 0;

} */
```

// B. Implement exception handling for the following error scenarios:

// 1. File opening failure: Throw a custom exception named FileOpenError if the file cannot be opened.

```
/* #include <iostream>

#include <fstream>

#include <exception>


using namespace std;
```

```
class FileOpenError : public exception {
public:
    const char* what() const throw() {
        return "File open error: Could not open the specified file";
    }
};

int main() {
    try {
        ifstream file("nonexistent.txt");
        if (!file.is_open()) {
            throw FileOpenError();
        }
        // Read file or perform other operations
    } catch (const FileOpenError& e) {
        cerr << "Error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "An unexpected error occurred: " << e.what() << endl;
    }

    return 0;
} */
```

// 2. Invalid data format: Throw a custom exception named InvalidDataFormatException if a line in the file doesn't match the expected format.

```
/* #include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <exception>

using namespace std;

class InvalidDataFormatException : public exception {
public:
    const char* what() const throw() {
        return "Invalid data format error: Data in file does not match the expected format";
    }
};

vector<vector<int>> readDataFromFile(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        throw runtime_error("File could not be opened");
    }

    vector<vector<int>> data;
    string line;
    while (getline(file, line)) {
        istringstream iss(line);
        vector<int> values;
        int value;
        while (iss >> value) {
            values.push_back(value);
        }
        if (values.size() != 3) {
            throw InvalidDataFormatException();
        }
        data.push_back(values);
    }
    file.close();
    return data;
}

int main() {
    try {
        vector<vector<int>> data = readDataFromFile("data.txt");
        // Process data further
    } catch (const InvalidDataFormatException& e) {
        cerr << "Error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "An unexpected error occurred: " << e.what() << endl;
    }
    return 0;
} */
```

// 3. Calculation errors: Throw a custom exception named CalculationError with a descriptive message if any calculation fails (e.g., division by zero).

```
#include <iostream>
#include <exception>
using namespace std;
class CalculationError : public exception {
    string message;
public:
    CalculationError(const string& msg) : message(msg) {}
    const char* what() const throw() {
        return message.c_str();
    }
};
int divide(int a, int b) {
    if (b == 0) {
        throw CalculationError("Division by zero error");
    }
    return a / b;
}
int main() {
    try {
        int result = divide(10, 0);
        cout << "Result: " << result << endl;
    } catch (const CalculationError& e) {
        cerr << "Error: " << e.what() << endl;
    } catch (const exception& e) {
        cerr << "An unexpected error occurred: " << e.what() << endl;
    }
    return 0;
}
```