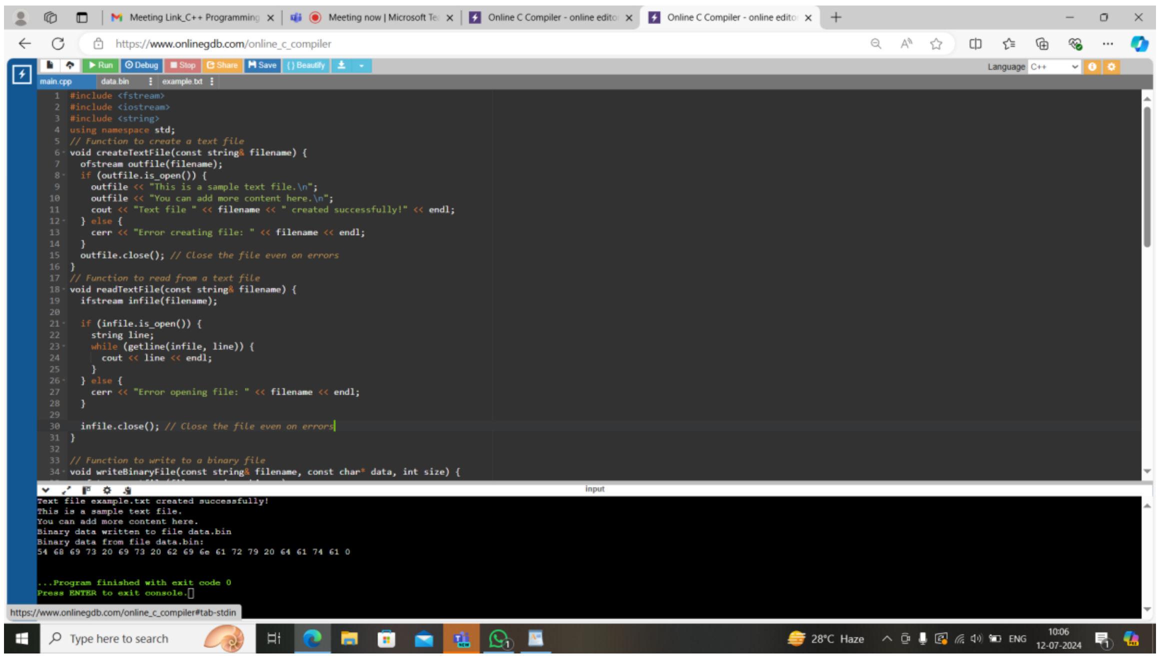
FILES OPERATIONS :-

```
#include <fstream>

#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()) {

        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()) {

        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;
}
```

OUTPUT :-



```cpp
#include <fstream>
#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()) {
    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()) {
    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) {
```

```
Text file example.txt created successfully!
This is a sample text file.
You can add more content here.
Binary data written to file data.bin
Binary data from file data.bin:
54 68 69 73 20 69 73 20 62 69 6e 61 72 79 20 64 61 74 61 0


...Program finished with exit code 0
Press ENTER to exit console.
```

63File Handling Practice Problems

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

Text Files:

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.

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.

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

#include <iostream>

#include <fstream>

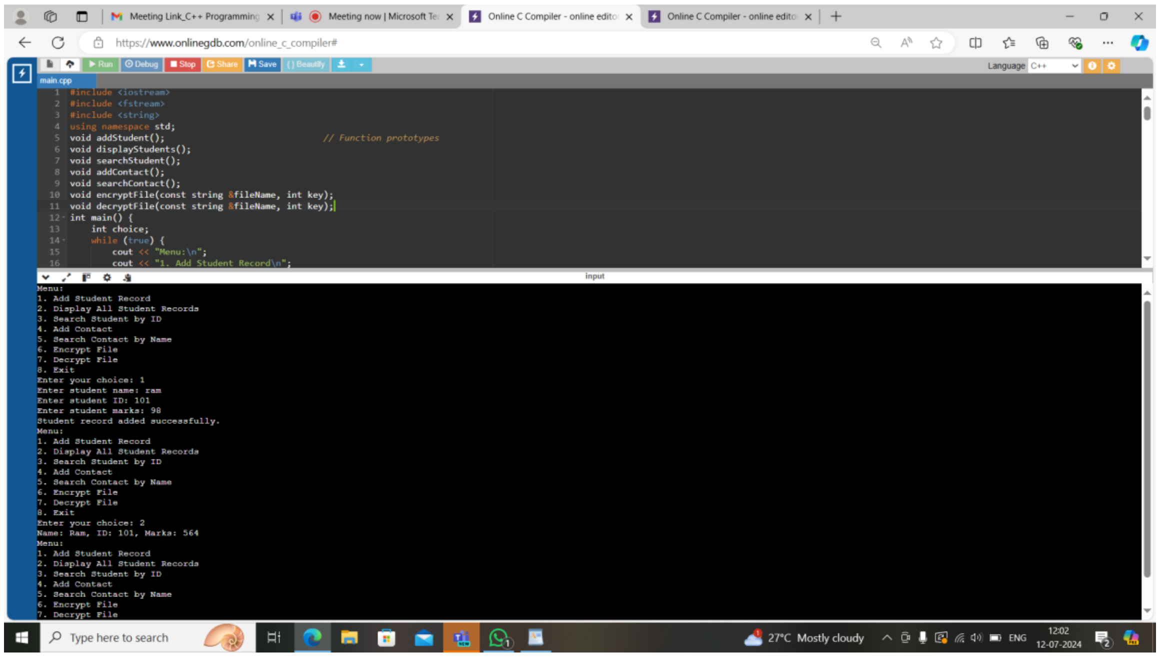
#include <string>

```
using namespace std;
void addStudent();                                          // Function prototypes
void displayStudents();
void searchStudent();
void addContact();
void searchContact();
void encryptFile(const string &fileName, int key);
void decryptFile(const string &fileName, int key);
int main() {
    int choice;
    while (true) {
        cout << "Menu:\n";
        cout << "1. Add Student Record\n";
        cout << "2. Display All Student Records\n";
        cout << "3. Search Student by ID\n";
        cout << "4. Add Contact\n";
        cout << "5. Search Contact by Name\n";
        cout << "6. Encrypt File\n";
        cout << "7. Decrypt File\n";
        cout << "8. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                addStudent();
                break;
            case 2:
                displayStudents();
                break;
            case 3:
                searchStudent();
                break;
            case 4:
                addContact();
                break;
            case 5:
                searchContact();
                break;
            case 6: {
                string fileName;
                int key;
                cout << "Enter file name to encrypt: ";
                cin >> fileName;
                cout << "Enter encryption key: ";
                cin >> key;
                encryptFile(fileName, key);
                break;
            }
            case 7: {
                string fileName;
                int key;
                cout << "Enter file name to decrypt: ";
                cin >> fileName;
                cout << "Enter decryption key: ";
                cin >> key;
                decryptFile(fileName, key);
```

```
                break;
            }
            case 8:
                return 0;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }
    return 0;
}
void addStudent() {                                              // Function to add a new student record
    ofstream outFile("students.txt", ios::app);
    string name, id;
    int marks;
    cout << "Enter student name: ";
    cin >> name;
    cout << "Enter student ID: ";
    cin >> id;
    cout << "Enter student marks: ";
    cin >> marks;
    outFile << name << " " << id << " " << marks << endl;
    outFile.close();
    cout << "Student record added successfully.\n";
}
void displayStudents() {                                     // Function to display all student records
    ifstream inFile("students.txt");
    string name, id;
    int marks;
    while (inFile >> name >> id >> marks) {
        cout << "Name: " << name << ", ID: " << id << ", Marks: " << marks << endl;
    }
    inFile.close();
}
void searchStudent() {                                        // Function to search for a student
by ID
    ifstream inFile("students.txt");
    string name, id, searchId;
    int marks;
    bool found = false;
    cout << "Enter student ID to search: ";
    cin >> searchId;
    while (inFile >> name >> id >> marks) {
        if (id == searchId) {
            cout << "Name: " << name << ", ID: " << id << ", Marks: " << marks << endl;
            found = true;
            break;
        }
    }
    inFile.close();
    if (!found) {
        cout << "Student with ID " << searchId << " not found.\n";
    }
}
void addContact() {                                                  // Function to add a new
contact
    ofstream outFile("contacts.txt", ios::app);
    string name, phoneNumber;
    cout << "Enter contact name: ";
    cin >> name;
    cout << "Enter contact phone number: ";
    cin >> phoneNumber;
    outFile << name << " " << phoneNumber << endl;
    outFile.close();
    cout << "Contact added successfully.\n";
}

// Function to search for a contact by name
void searchContact() {
    ifstream inFile("contacts.txt");
    string name, phoneNumber, searchName;
    bool found = false;
    cout << "Enter contact name to search: ";
    cin >> searchName;
    while (inFile >> name >> phoneNumber) {
        if (name == searchName) {
            cout << "Name: " << name << ", Phone Number: " << phoneNumber << endl;
            found = true;
            break;
        }
    }
    inFile.close();
    if (!found) {
        cout << "Contact with name " << searchName << " not found.\n";
    }
```

```
}

void encryptFile(const string &fileName, int key) {                          // Function to encrypt a file using
Caesar cipher

        ifstream inFile(fileName);

        ofstream outFile(fileName + ".enc");

        char ch;

        while (inFile.get(ch)) {

                outFile.put(ch + key);

        }

        inFile.close();

        outFile.close();

        cout << "File encrypted successfully.\n";

}

void decryptFile(const string &fileName, int key) {                          // Function to decrypt a file using
Caesar cipher

        ifstream inFile(fileName);

        ofstream outFile(fileName + ".dec");

        char ch;

        while (inFile.get(ch)) {

                outFile.put(ch - key);

        }

        inFile.close();

        outFile.close();

        cout << "File decrypted successfully.\n";

}
```

OUTPUT :-



```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void addStudent();                                   // Function prototypes
void displayStudents();
void searchStudent();
void addContact();
void searchContact();
void encryptFile(const string &fileName, int key);
void decryptFile(const string &fileName, int key);
int main() {
    int choice;
    while (true) {
        cout << "Menu:\n";
        cout << "1. Add Student Record\n";
```

```
Menu:
1. Add Student Record
2. Display All Student Records
3. Search Student by ID
4. Add Contact
5. Search Contact by Name
6. Encrypt File
7. Decrypt File
8. Exit
Enter your choice: 1
Enter student name: ram
Enter student ID: 101
Enter student marks: 98
Student record added successfully.
Menu:
1. Add Student Record
2. Display All Student Records
3. Search Student by ID
4. Add Contact
5. Search Contact by Name
6. Encrypt File
7. Decrypt File
8. Exit
Enter your choice: 2
Name: Ram, ID: 101, Marks: 564
Menu:
1. Add Student Record
2. Display All Student Records
3. Search Student by ID
4. Add Contact
5. Search Contact by Name
6. Encrypt File
7. Decrypt File
```

Binary Files:

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.

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;

void copyImage(const string &sourceFile, const string &destFile);                    // Function prototypes

void addItem();
void displayItems();

void updateQuantity();

void saveHighScore();

void displayHighScore();

struct Item {

    char name[50];

    double price;

    int quantity;

};

struct HighScore {

    char playerName[50];

    int score;

};

int main() {

    int choice;

    while (true) {

        cout << "Menu:\n";

        cout << "1. Copy Image\n";

        cout << "2. Add New Inventory Item\n";

        cout << "3. Display All Inventory Items\n";

        cout << "4. Update Item Quantity\n";

        cout << "5. Save High Score\n";

        cout << "6. Display High Score\n";

        cout << "7. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;


        switch (choice) {
```

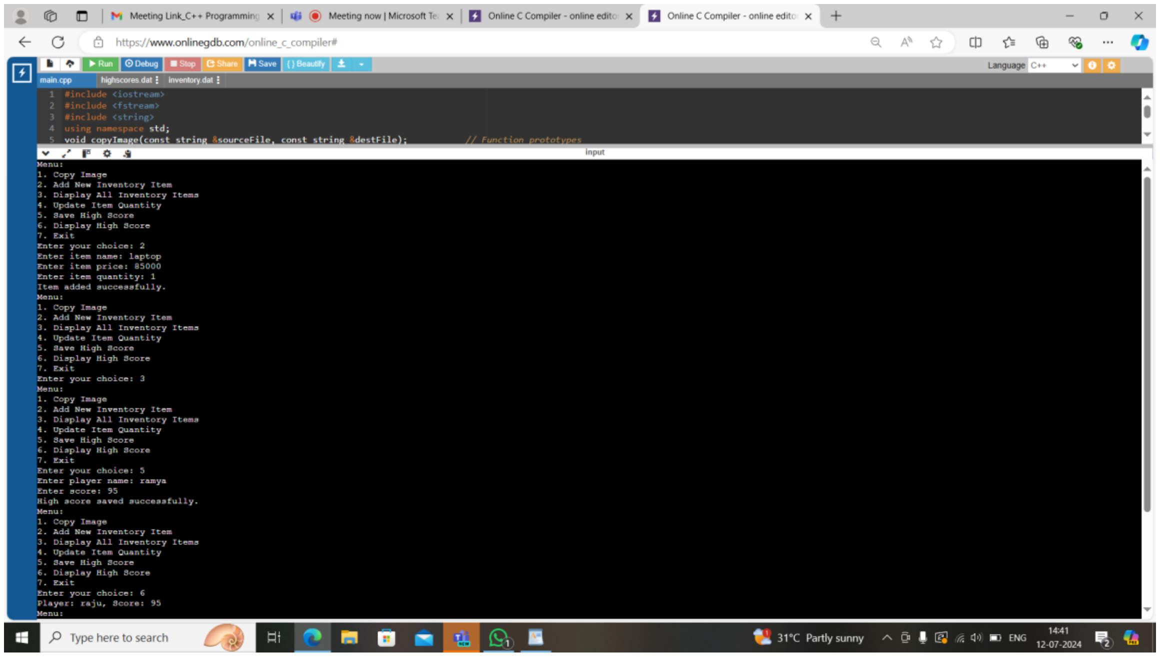
```
case 1: {
    string sourceFile, destFile;
    cout << "Enter source image file name: ";
    cin >> sourceFile;
    cout << "Enter destination image file name: ";
    cin >> destFile;
    copyImage(sourceFile, destFile);
    break;
}
case 2:
    addItem();
    break;
case 3:
    displayItems();
    break;
case 4:
    updateQuantity();
    break;
case 5:
    saveHighScore();
    break;
case 6:
    displayHighScore();
    break;
case 7:
    return 0;
default:
    cout << "Invalid choice. Please try again.\n";
```

```
        }
    }

    return 0;
}
void copyImage(const string &sourceFile, const string &destFile) {          // Function prototypes
    ifstream inFile(sourceFile, ios::binary);
    ofstream outFile(destFile, ios::binary);
    if (!inFile) {
        cout << "Error opening source file.\n";
        return;
    }
    if (!outFile) {
        cout << "Error creating destination file.\n";
        return;
    }
    outFile << inFile.rdbuf();
    cout << "Image copied successfully.\n";
}
void addItem() {                              // Function to add a new inventory item
    ofstream outFile("inventory.dat", ios::binary | ios::app);
    Item item;
    cout << "Enter item name: ";
    cin.ignore();
    cin.getline(item.name, 50);
    cout << "Enter item price: ";
    cin >> item.price;
    cout << "Enter item quantity: ";
```

```
        cin >> item.quantity;

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

        outFile.close();

        cout << "Item added successfully.\n";

}


// Function to display all inventory items

void displayItems() {

        ifstream inFile("inventory.dat", ios::binary);

        Item item;

        while (inFile.read(reinterpret_cast<char*>(&item), sizeof(item))) {

                cout << "Name: " << item.name << ", Price: " << item.price << ", Quantity: " << item.quantity
<< endl;

        }

        inFile.close();

}


// Function to update the quantity of an existing item

void updateQuantity() {

        fstream file("inventory.dat", ios::binary | ios::in | ios::out);

        Item item;

        char searchName[50];

        cout << "Enter the name of the item to update: ";

        cin.ignore();

        cin.getline(searchName, 50);

        bool found = false;

        while (file.read(reinterpret_cast<char*>(&item), sizeof(item))) {

                if ((item.name, searchName) == 0) {
```

```
                cout << "Enter new quantity: ";
                cin >> item.quantity;
                file.seekp(-sizeof(item), ios::cur);
                file.write(reinterpret_cast<char*>(&item), sizeof(item));
                found = true;
                cout << "Item quantity updated successfully.\n";
                break;
            }
        }
        if (!found) {
            cout << "Item not found.\n";
        }
        file.close();
}
void saveHighScore() {                                          // Function to save a new high score
        ofstream outFile("highscores.dat", ios::binary | ios::app);
        HighScore score;
        cout << "Enter player name: ";
        cin.ignore();
        cin.getline(score.playerName, 50);
        cout << "Enter score: ";
        cin >> score.score;
        outFile.write(reinterpret_cast<char*>(&score), sizeof(score));
        outFile.close();
        cout << "High score saved successfully.\n";
}
void displayHighScore() {                                            // Function to display the current high score
        ifstream inFile("highscores.dat", ios::binary);
        HighScore score, highScore;
        bool first = true;
        while (inFile.read(reinterpret_cast<char*>(&score), sizeof(score))) {
            if (first || score.score > highScore.score) {
                highScore = score;
                first = false;
            }
        }
        if (!first) {
            cout << "Player: " << highScore.playerName << ", Score: " << highScore.score << endl;
        } else {
            cout << "No high scores recorded.\n";
        }
        inFile.close();
    }
```

OUTPUT :-



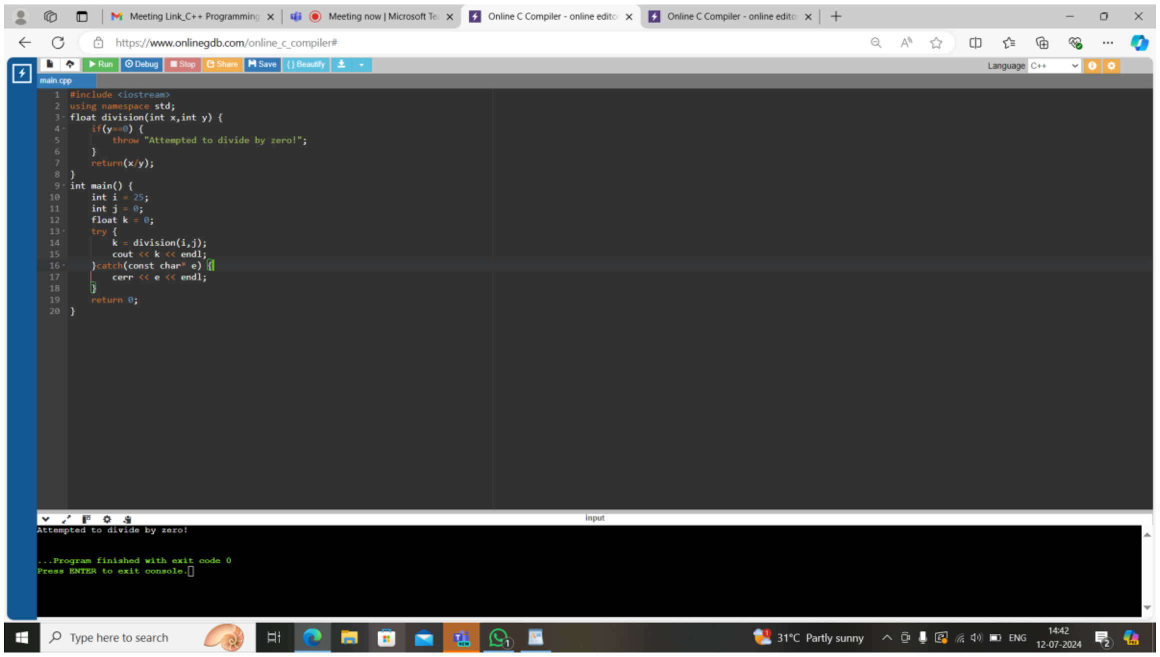
```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void copyImage(const string &sourceFile, const string &destFile);          // Function prototypes
```

```
Menu:
1. Copy Image
2. Add New Inventory Item
3. Display All Inventory Items
4. Update Item Quantity
5. Save High Score
6. Display High Score
7. Exit
Enter your choice: 2
Enter item name: laptop
Enter item price: 85000
Enter item quantity: 1
Item added successfully.
Menu:
1. Copy Image
2. Add New Inventory Item
3. Display All Inventory Items
4. Update Item Quantity
5. Save High Score
6. Display High Score
7. Exit
Enter your choice: 3
Menu:
1. Copy Image
2. Add New Inventory Item
3. Display All Inventory Items
4. Update Item Quantity
5. Save High Score
6. Display High Score
7. Exit
Enter your choice: 5
Enter player name: ramya
Enter score: 95
High score saved successfully.
Menu:
1. Copy Image
2. Add New Inventory Item
3. Display All Inventory Items
4. Update Item Quantity
5. Save High Score
6. Display High Score
7. Exit
Enter your choice: 6
Player: raju, Score: 95
Menu:
```

TRY/CATCH (DIVISION) :-

```
#include <iostream>
using namespace std;
float division(int x,int y) {
    if(y==0) {
        throw "Attempted to divide by zero!";
    }
    return(x/y);
}
int main() {
    int i = 25;
    int j = 0;
    float k = 0;
    try {
        k = division(i,j);
        cout << k << endl;
    }catch(const char* e) {
        cerr << e << endl;
    }
    return 0;
}
```

OUTPUT :-



```cpp
#include <iostream>
using namespace std;
float division(int x,int y) {
    if(y==0) {
        throw "Attempted to divide by zero!";
    }
    return(x/y);
}
int main() {
    int i = 25;
    int j = 0;
    float k = 0;
    try {
        k = division(i,j);
        cout << k << endl;
    }catch(const char* e) {
        cerr << e << endl;
    }
    return 0;
}
```

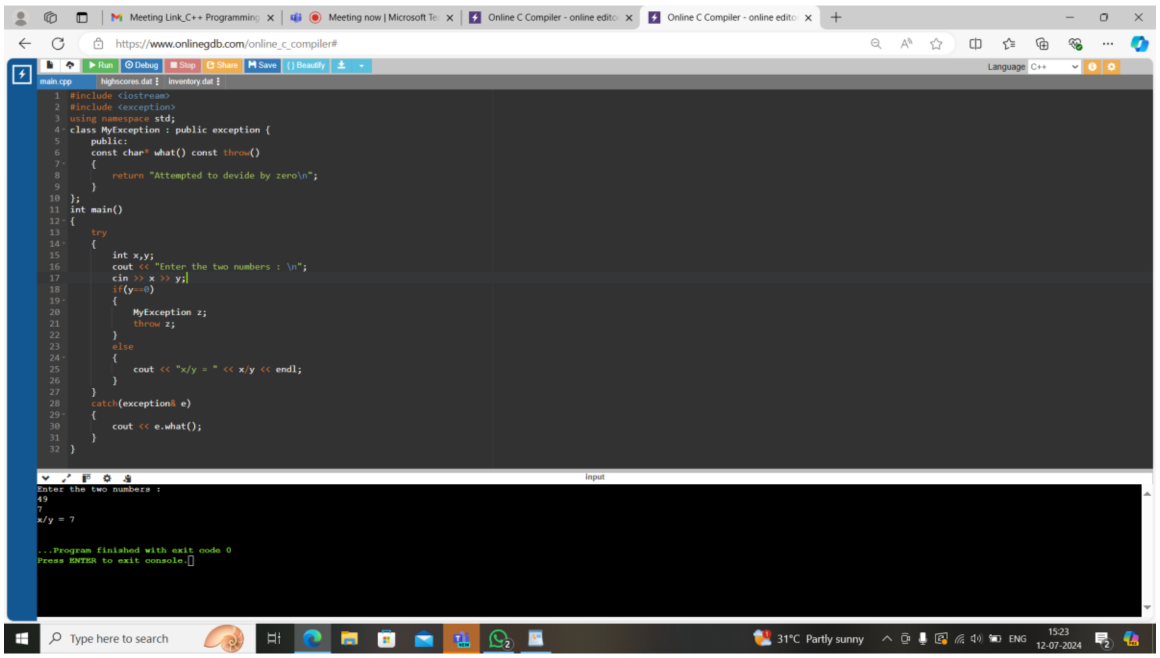
```
Attempted to divide by zero!


...Program finished with exit code 0
Press ENTER to exit console.
```

USER DEFINED EXCEPTION :-

```cpp
#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception {

        public:

        const char* what() const throw()

        {

                return "Attempted to devide 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(exception& e)
        {
            cout << e.what();
        }
}
```

OUTPUT :-



```cpp
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception {
    public:
    const char* what() const throw()
    {
        return "Attempted to devide 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(exception& e)
    {
        cout << e.what();
    }
}
```

```
Enter the two numbers :
49
7
x/y = 7


...Program finished with exit code 0
Press ENTER to exit console.
```

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

Using exceptions in C++ offers several advantages and disadvantages compared to traditional error codes:

Advantages of Exceptions :-

1. Separation of Error Handling from Normal Flow : Exceptions allow you to separate the code that detects errors from the code that handles them, improving code readability and maintainability.

2. Error Propagation : Exceptions can propagate up the call stack automatically until they are caught, which means error handling doesn't need to be explicitly managed at every level of the call hierarchy.

3. Centralized Error Handling : Exceptions can be caught at a higher level where it makes sense to handle them, rather than handling errors immediately where they occur. This can lead to more centralized and consistent error handling logic.

4. Easier Debugging : Exceptions can provide more detailed error information, including stack traces (if supported), which can be very helpful in debugging and diagnosing issues.

5. Focus on Normal Path : Code that uses exceptions tends to focus more on the normal path of execution, making the code cleaner and easier to understand without error handling clutter.

Disadvantages of Exceptions :-

1. Performance Overhead : Throwing and catching exceptions can be more costly in terms of performance compared to simple error code checks, especially in scenarios where exceptions are thrown frequently.

2. Resource Management : Exceptions can make resource management more complex, as you need to ensure that resources are properly cleaned up in both normal and exceptional paths.

3. Compatibility and Interoperability : Exceptions might not be compatible with certain environments or languages, especially in mixed-language programming or embedded systems.

4. Code Size : Exception handling code can increase the size of the executable, as the compiler typically generates additional code to handle exceptions.

5. Misuse and Abuse : Improper use of exceptions, such as using them for control flow rather than error handling, can lead to less maintainable and harder-to-understand code.

Considerations for Choosing Between Exceptions and Error Codes :-

Complexity of the Application : For simple applications with straightforward error conditions, error codes might be sufficient and simpler to manage. For more complex applications with many potential error conditions, exceptions might provide better structure and clarity.

Performance Requirements : If performance is critical and exceptions would impose too much overhead, error codes might be preferable.

Team Expertise and Codebase : Consider the expertise of your development team and the existing codebase conventions. If your team is experienced with exceptions and the codebase is already using them effectively, it might be easier to stick with exceptions.

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

To ensure that exception classes provide informative error messages for debugging in C++, you can follow these best practices:

1. Include Relevant Information in the Message :

- Ensure that the exception message contains all pertinent information about the error. This might include details such as what caused the exception, the context in which it occurred, and any relevant state information.

2. Use Descriptive Error Messages :

- Choose clear and descriptive messages that convey the nature of the problem. Avoid overly generic messages like "Error occurred" or "Exception caught".

3. Include Contextual Information :

- Include information that helps in identifying the location and cause of the exception. This could involve mentioning the function or method where the exception originated, relevant parameters or variables, and any other contextual details.

4. Consider Error Codes or Enums :

- In addition to textual messages, consider using error codes or enumerations that can be used programmatically to identify different types of errors. This can be especially useful when handling exceptions in a structured manner.

5. Avoid Revealing Sensitive Information :

- Be cautious not to include sensitive information (such as passwords, user data, etc.) in exception messages, as these might be exposed in logs or error reports.

6. Use `what()` Method Override :

- In C++, exceptions should typically override the `what()` method inherited from `std::exception`. This method should return a `const char*` describing the exception. Ensure that this description is informative and clear.

Here's an example of how an exception class might be designed to provide informative error messages:

```
#include <stdexcept>

#include <string>

class MyException : public std::exception {

private:

    std::string message;

public:

    MyException(const std::string& msg) : message(msg) {}


    // Override what() to provide a descriptive message

    virtual const char* what() const noexcept override {

        return message.c_str();

    }

};

void someFunction() {

    // Example of throwing the exception

    throw MyException("Error: Unable to perform operation X due to invalid input.");

}

int main() {

    try {
```

```cpp
        someFunction();

    } catch (const MyException& e) {

        // Catch and handle the exception, printing the error message

        std::cerr << "Caught exception: " << e.what() << std::endl;

    }

    return 0;

}
```

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

Optimizing exception handling performance in C++ is crucial for performance-critical applications where microseconds matter. Here are some strategies to achieve better performance:

1. Minimize Exception Usage

- **Prefer Error Codes:** Instead of relying on exceptions for error handling, use error codes or return values where appropriate. This eliminates the overhead associated with exception handling altogether.

- **Exceptions for Exceptional Cases:** Use exceptions only for exceptional circumstances, such as unrecoverable errors or exceptional conditions that should not normally occur during program execution.

2. Use noexcept Specification

noexcept : Use `noexcept` specification for functions that are not supposed to throw exceptions. This allows the compiler to optimize more aggressively because it knows that no exceptions will be propagated from these functions.

```cpp

void func() noexcept {

    // Function body

}
```

3. Avoid Unnecessary Catch Blocks

Specificity in Catch Blocks: Catch exceptions by their most derived type whenever possible. This allows the compiler to generate more efficient code for handling specific exception types.

```cpp

try {
    // Code that might throw
} catch (const std::exception& ex) {
    // Handle std::exception and its derived types
} catch (const MyException& ex) {
    // Handle MyException and its derived types
}
```

Avoid Catch-All Blocks : Minimize the use of `catch (...)` as it catches all exceptions, including those not derived from `std::exception`. This can lead to less efficient handling due to the need for more dynamic checks.

4. Optimize Stack Unwinding

Minimize Deep Exception Nesting : Deeply nested try-catch blocks can impact performance, especially during stack unwinding. Refactor code to reduce nested try blocks where possible.

5. Compile-Time Optimization

Enable Compiler Optimizations : Use compiler optimizations (`-O2`, `-O3` in GCC/Clang) to allow the compiler to inline functions more aggressively and optimize exception handling code paths.

6. Profile and Benchmark

Measure Performance: Use profiling tools to identify performance bottlenecks related to exception handling. Benchmark different error handling strategies to determine the most efficient approach for your specific application.

7. Custom Allocation Strategies

Custom `new` and `delete`:If your application frequently throws exceptions, consider implementing custom `new` and `delete` operators to optimize memory allocation and deallocation, which can improve overall exception handling performance.

8. Exception Safety Guarantees

Choose the Right Level of Exception Safety:** Understand and implement the appropriate level of exception safety (basic, strong, or no-throw) depending on your application's requirements. This can prevent unnecessary overhead in exception handling.

9. Library and Framework Considerations

Choose Efficient Libraries : When selecting third-party libraries or frameworks, consider their exception handling strategies and performance implications. Opt for libraries that align with your performance goals.

10. Use of RAII :-

Resource Acquisition Is Initialization (RAII) :RAII can simplify exception handling by tying resource management to object lifetimes. This approach can lead to cleaner, more efficient code compared to manual resource management.

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

Designing a hierarchy of exception classes in C++ can greatly enhance code maintainability and reusability by providing a structured way to handle different types of errors or exceptional situations. Here's a guideline on how to design such a hierarchy:

1. Base Exception Class

Start by defining a base exception class that serves as the root of your hierarchy. This class should inherit from `std::exception` (which is the standard base class for exceptions in C++) and provide additional functionalities as needed.

```
#include <exception>

#include <string>

class BaseException : public std::exception {

private:

    std::string message_;

public:

    explicit BaseException(const std::string& message) : message_(message) {}

    const char* what() const noexcept override {

        return message_.c_str();

    }

    virtual ~BaseException() noexcept = default;

};
```

2. Specific Exception Classes

Derive specific exception classes from the `BaseException` class to represent different categories of errors or exceptional conditions in your application. Each derived class can include additional data members or methods specific to the type of exception it represents.

```
class FileIOException : public BaseException {

public:

    explicit FileIOException(const std::string& message) : BaseException(message) {}
};
class NetworkException : public BaseException {
public:
    explicit NetworkException(const std::string& message) : BaseException(message) {}
};
// More specific exception classes as needed
```

3. Use Case Example

Consider how these exception classes might be used in a function

```cpp
#include <fstream>
void readFile(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        throw FileIOException("Failed to open file: " + filename);
    }
    // Read file contents...
}
```

Here, `FileIOException` is thrown when there's an issue related to file operations, providing a clear indication of what went wrong.

4. Handling Exceptions

When handling exceptions, you can catch specific exceptions based on their types. This allows for more granular error handling and recovery strategies:

```
try {
    readFile("example.txt");
} catch (const FileIOException& e) {
    // Handle file IO exception
    std::cerr << "File IO error: " << e.what() << std::endl;
} catch (const NetworkException& e) {
    // Handle network-related exception
    std::cerr << "Network error: " << e.what() << std::endl;
} catch (const std::exception& e) {
    // Catch all other exceptions
    std::cerr << "Error: " << e.what() << std::endl;
}
```

Code Maintainability : With a hierarchy of exception classes, it's easier to understand the types of exceptions that can be thrown and how to handle them.

Code Reusability : By subclassing from a base exception class, you can reuse common exception handling logic across different parts of your application.

Granular Error Handling : Allows for specific error messages and handling strategies based on the type of exception, improving application robustness.

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

In C++, exceptions are a powerful mechanism for error handling, but there are scenarios where using them might not be appropriate or desirable:

1. Embedded Systems and Performance-Critical Applications :

- In environments where every CPU cycle counts, the overhead of exception handling (both in terms of code size and runtime performance) can be significant. In such cases, error codes or alternative error handling strategies (like return codes) are often preferred because they are more predictable and can be optimized more easily by the compiler.

2. Compatibility with C Libraries :

- When integrating with existing C libraries or APIs that do not support exceptions, using exceptions for error handling can complicate interoperability and error propagation. In these cases, adhering to the error handling mechanisms of the C library (typically error codes or function return values indicating errors) is often necessary for consistency and ease of maintenance.

3. Resource-Constrained Environments :

- Some embedded systems or specialized environments have limited memory resources. Exception handling mechanisms in C++ might require additional memory overhead for the exception handling tables and unwinding, which can be prohibitive in such resource-constrained environments.

4. Code Maintainability and Familiarity :

- In some codebases or teams, there might be a preference or policy to avoid exceptions due to concerns about code readability, maintainability, or familiarity. Some developers find error codes or explicit error handling through return values or out-parameters clearer and easier to reason about, especially in complex codebases.

5. Real-Time Systems :

- Systems that require deterministic behavior and strict timing constraints (like real-time operating systems) often avoid exceptions because they introduce non-deterministic behavior. Error handling in such systems is typically done using error codes or similar mechanisms to ensure precise control over execution flow.

6. Specific Project or Team Guidelines :

- Sometimes, the decision to use or avoid exceptions is based on project-specific guidelines or coding standards. Teams may have decided to use a consistent error handling strategy across the codebase for better consistency and maintainability.

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

The program should:

Read data from a text file.

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

Perform calculations based on the valid data.

Implement exception handling for the following error scenarios:

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

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

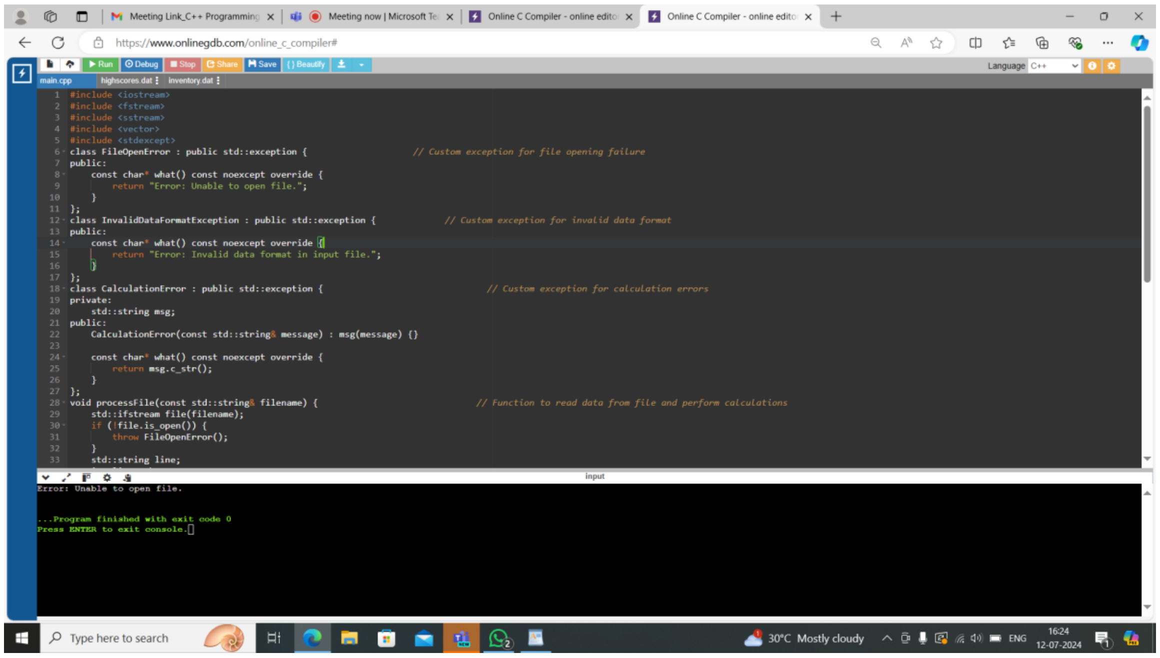
Calculation errors: Throw a custom exception named CalculationError with a descriptive message if any calculation fails (e.g., division by zero).

```
#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <stdexcept>

class FileOpenError : public std::exception {                    // Custom exception for file
opening failure
```

```
public:

        const char* what() const noexcept override {

                return "Error: Unable to open file.";

        }

};

class InvalidDataFormatException : public std::exception {                          // Custom exception for
invalid data format

public:

        const char* what() const noexcept override {

                return "Error: Invalid data format in input file.";

        }

};

class CalculationError : public std::exception {                                                    // Custom
exception for calculation errors

private:

        std::string msg;

public:

        CalculationError(const std::string& message) : msg(message) {}


        const char* what() const noexcept override {

                return msg.c_str();

        }

};

void processFile(const std::string& filename) {                                                   // Function to read
data from file and perform calculations

        std::ifstream file(filename);

        if (!file.is_open()) {

                throw FileOpenError();

        }
        std::string line;

        int lineNumber = 0;

        while (std::getline(file, line)) {

                lineNumber++;

                std::istringstream iss(line);                                                    //
Example: Assuming we expect two integers per line separated by space

                int num1, num2;

                if (!(iss >> num1 >> num2)) {

                        throw InvalidDataFormatException();

                }

                try {
// Perform some calculation

                        if (num2 == 0) {

                                throw CalculationError("Error: Division by zero.");

                        }

                        double result = static_cast<double>(num1) / num2;

                        std::cout << "Result of calculation on line " << lineNumber << ": " << result << std::endl;

                } catch (const CalculationError& e) {

                        std::cerr << e.what() << std::endl;

                }

        }

        file.close();

}

int main() {

        std::string filename = "data.txt";

        try {

                processFile(filename);

        } catch (const FileOpenError& e) {
```

```
        std::cerr << e.what() << std::endl;
    } catch (const InvalidDataFormatException& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

OUTPUT :-



```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <stdexcept>
class FileOpenError : public std::exception {                  // Custom exception for file opening failure
public:
    const char* what() const noexcept override {
        return "Error: Unable to open file.";
    }
};
class InvalidDataFormatException : public std::exception {            // Custom exception for invalid data format
public:
    const char* what() const noexcept override {
        return "Error: Invalid data format in input file.";
    }
};
class CalculationError : public std::exception {                             // Custom exception for calculation errors
private:
    std::string msg;
public:
    CalculationError(const std::string& message) : msg(message) {}

    const char* what() const noexcept override {
        return msg.c_str();
    }
};
void processFile(const std::string& filename) {                          // Function to read data from file and perform calculations
    std::ifstream file(filename);
    if (!file.is_open()) {
        throw FileOpenError();
    }
    std::string line;
```

```
Error: Unable to open file.


...Program finished with exit code 0
Press ENTER to exit console.
```