747993_SHAANREHSI_A5

I used:
https://www.youtube.com/watch?v=fwl3BYvl1CA&list=PLDBYedoMzOlzgiSvL2_CKCg-gMh2vtzvz to help me with this assignment.

1. **Define a 2D matrix of ints, Imatrix, with the following operations:**

   Default construction: all elements get the default value 0.
   Assignment operator and copy and move constructors.
   Subscripting: m(x,y) is the x,y element. You should be able to assign to this x,y element in the matrix.
   +, *, /, -, and %, yielding a new Imatrix.
   Move(x,y): place the value from location x to location y and set x to 0
   Row(n): return a vector<int> with the values from the nth row
   Column: return a vector<int> with the values from the nth column

To make sure that the matrix doesn't leak memory and to handle range errors more robustly, I added a destructor and updated the subscripting operator to throw an std::out_of_range exception if an index is out of bounds. Additionally, I added checks in the arithmetic operator functions to ensure that the matrices involved in the operations have compatible dimensions.

I also used the subscripting operator to check for out-of-range indices and throw an std::out_of_range exception if needed. This ensures that access to elements outside the matrix dimensions is not allowed.

In the main function, I wrapped the operations within a try-catch block to handle potential exceptions that might be thrown during matrix operations.

Design Decisions:

- I used std::vector to manage the matrix data to ensure active memory allocation and deallocation.

- I used concepts to restrict the matrix operations to integral types to avoid potential issues with floating-point arithmetic.

- I used std::invalid_argument exceptions for situations where the matrix dimensions are not compatible for arithmetic operations and for division and modulo by zero.

- I used the std::swap function in the ‘Move’ function to be able to swap two elements in the matrix.

- I added exception handling in the main function to catch and display any exceptions thrown during matrix operations.

These design decisions help to make sure that the Imatrix class is safer, less prone to errors, and better manages memory, making it more reliable for various matrix operations.

```
Matrix 1:
1 2 3
4 5 6
7 8 9

Matrix 2:
9 8 7
6 5 4
3 2 1

Result of addition:
10 10 10
10 10 10
10 10 10

Result of subtraction:
-8 -6 -4
-2 0 2
4 6 8

Result of multiplication:
30 24 18
84 69 54
138 114 90

Result of division:
0 1 1
2 2 3
3 4 4

Result of modulo:
1 2 0
1 2 0
1 2 0
```

Experiments

```
Matrix 3:
1.1 2.2 3.3
4.4 5.5 6.6
7.7 8.8 9.9

Result of double addition:
2.2 4.4 6.6
8.8 11 13.2
15.4 17.6 19.8
```

Testing Non-Integral Types: I used a new matrix using double values and performed an addition operation. This experiment tests the flexibility of Imatrix class to work with different types.

Performance Testing: I made larger matrices (large_matrix1 and large_matrix2) and measured the time taken for matrix multiplication. This experiment checks the performance of the implementation for larger inputs. large_matrix1 and large_matrix2 of size 1000x1000 are created. The time taken to perform matrix multiplication using these large matrices is measured using the <chrono> library. The duration is displayed to show if the implementation for larger inputs is efficient

Overall, these design choices enhance the functionality and flexibility of the Imatrix class by allowing it to work with various data types, and they also provide performance insights for larger matrix operations.

Advantages:

Abstraction and Reusability: The Imatrix class abstracts away the details of matrix manipulation and provides a reusable component. This can save time and effort when working with matrices in various projects.

Organisation of Code: The class provides a structured way to organise matrix-related functionality, making it easier to manage and extend over time.

Modularity: By summarising matrix operations within the class, modular design is created that makes it easier to understand, test, and maintain code.

Type Safety: The use of concepts and type requirements ensures that matrix operations are limited to integral types. This helps catch potential errors early and provides a level of type safety.

Memory Management: The use of std::vector for managing matrix data ensures automatic memory allocation and deallocation, helping to prevent memory leaks and manage memory efficiently.

Exception Handling: The matrix class includes exception handling for cases of out-of-range indices and invalid operations, making the code more robust and user-friendly.

Disadvantages:
Limited Flexibility: The class is tailored for integral types and doesn't handle other numeric types, which can limit its applicability in cases requiring floating-point or custom numeric types.

Performance Overhead: The use of std::vector for matrix storage can introduce some overhead compared to more specialised storage methods for larger matrices. Also, the implementation of matrix operations might not be as optimised as dedicated linear algebra libraries.

Complexity: The matrix class implementation includes multiple templated functions, which can make the code more complex and harder to understand for newcomers to the codebase.

2. Generalise your Imatrix to take the element type, T, as a parameter, e.g., your Imatrix should be equivalent to Matrix<int>.

> **With regards to the operations:**
>
> **Default construct: all elements gets a default value T{}**
> **= and copy and move constructors.**
> **Subscripting: m(x,y) is the x,y element that supports assignment to said element.**
> **+, *, /, - yield a new Matrix**
> **Have % defined if the elements are like integers.**
> **Move: place the value from location x to location y and leave x in the default state.**
> **All operations must be generic (of course), so define concepts as needed. You can find a list of the std concepts here: https://en.cppreference.com/w/cpp/concepts.**
>
> **Note: Concepts are not required! If you are using C++17, just implement the matrix without any guards, but try to see the error message when using an unsupported operation, e.g. % for non-integers.**
> **Again, make sure your matrix doesn't leak memory and it throws an exception if an operation would involve a range error.**
>
> **Define a minimal Chess_piece type.**
>
> **Define and test Matrix<int> (this should yield identical code to Imatrix), Matrix<string>, and Matrix<Chess_piece> (the chess piece may require you to add and/or move some of your concept requirements).**

Design Choices:

Class Template: The Matrix class is implemented as a class template. This allows the matrix to work with different types while providing the same set of operations. The type T is a template parameter, and the class is instantiated with specific types, like int, std::string, or Chess_piece.

Concepts: The concept Arithmetic is used to ensure that the template type T is an arithmetic type (like int, float, etc.) when performing operations like +, -, *, /. This helps prevent accidental misuse of the operations with unsupported types.

Constructor: The constructor of the Matrix class takes the number of rows and columns as parameters. This allows the user to specify the dimensions of the matrix during instantiation.

Data Storage: The matrix data is stored as a one-dimensional std::vector<T>. The 2D indexing is emulated by mapping the 2D indices (x, y) to a 1D index (x * cols_ + y). This approach provides better memory locality and ease of resizing compared to using nested vectors.

Bounds Checking: The operator() for element access checks whether the provided indices are within bounds before accessing the data. If the indices are out of range, an exception of type std::out_of_range is thrown.

Matrix Operations: In the provided example, the operator= is defined to illustrate how matrix addition could be implemented. Similar operators (-, *, /) are implemented similarly. These operations check for compatibility of matrix sizes before performing the operation.

Custom Type: A custom Chess_piece type is introduced to demonstrate how the Matrix class can work with non-arithmetic types. The Chess_piece class is simple, holding a character symbol representing a chess piece.

Error Handling: The implementation includes basic error handling using try and catch. Exceptions are thrown for out-of-range indices and incompatible matrix sizes.

Main Function: The main function demonstrates the use of the Matrix class with different types (int, std::string, Chess_piece). It initialises matrices with values and performs basic operations.

Overall, the designs I used aim to create a versatile matrix class that can be easily extended to work with different data types. The use of concepts helps ensure that the operations are supported for the specific element type, providing better compile-time checks and more informative error messages.