C++ Generic Programming Exercise

Richard Kroesen (774056)
1-8-2024 Aarhus

1. Introduction

This report outlines the implementation and testing of a two-dimensional matrix class, Matrix<T>, using templates and concepts in C++. The assignment required defining a specific Imatrix for integers, and subsequently generalizing this to a template-based Matrix<T> capable of handling various data types.

The primary goals included implementing essential matrix operations, ensuring efficient memory management, and supporting a broad range of element types, including user-defined types like a minimal Chess_piece class.

The report is written in practical and direct way, more details could be found in the program itself, and/or in the README of the repository.

2. Program Implementation

In this chapter, the program examples are given and the design considerations.

2.1 Program Examples

Matrix – Creationg & Summing

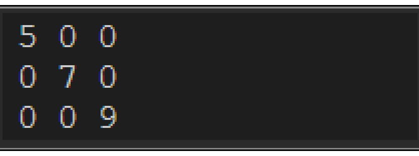
```cpp
MATRIX::Matrix<int> matrix1(3, 3);
MATRIX::Matrix<int> matrix2(3, 3);

matrix1(0, 0) = 1;
matrix1(1, 1) = 2;
matrix1(2, 2) = 3;

matrix2(0, 0) = 4;
matrix2(1, 1) = 5;
matrix2(2, 2) = 6;

MATRIX::Matrix<int> matrix3 = matrix1 + matrix2;
for (size_t i = 0; i < 3; ++i) {
    for (size_t j = 0; j < 3; ++j) {
        std::cout << matrix3(i, j) << " ";
    }
    std::cout << std::endl;
}
```

Output:



```
5 0 0
0 7 0
0 0 9
```

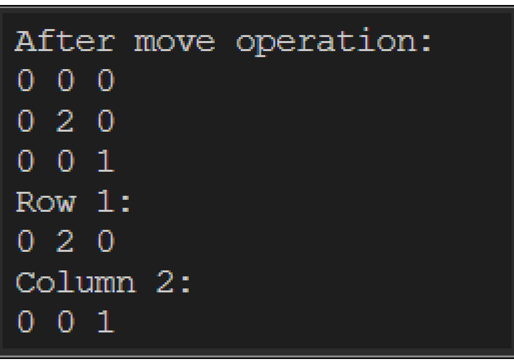
Matrix Move & Retrieve:

```cpp
matrix1.move(0, 0, 2, 2);
```

```cpp
std::cout << "After move operation:" << std::endl;
for (size_t i = 0; i < 3; ++i) {
    for (size_t j = 0; j < 3; ++j) {
        std::cout << matrix1(i, j) << " ";
    }
    std::cout << std::endl;
}
auto row = matrix1.row(1);
std::cout << "Row 1:" << std::endl;
for (const auto& val : row) {
    std::cout << val << " ";
}
std::cout << std::endl;

auto column = matrix1.column(2);
std::cout << "Column 2:" << std::endl;
for (const auto& val : column) {
    std::cout << val << " ";
}
std::cout << std::endl;
```

Output:

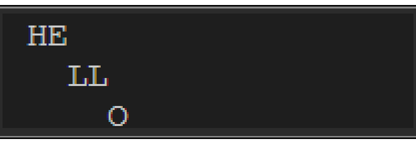


```
After move operation:
0 0 0
0 2 0
0 0 1
Row 1:
0 2 0
Column 2:
0 0 1
```

String Matrix:

```cpp
MATRIX::Matrix<std::string> matrix4(3, 6);
matrix4(0, 0) = "HE";
matrix4(1, 2) = "LL";
matrix4(2, 4) = "O";
matrix4.print();
```

Output:

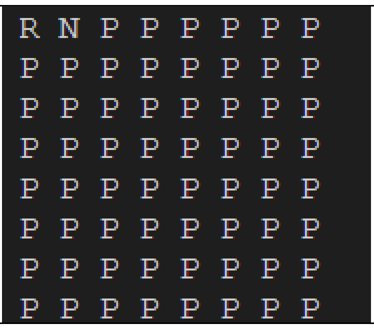


```
HE
    LL
        O
```

Chess Matrix:

```cpp
MATRIX::Matrix<MATRIX::Chess_piece> chess_board(8, 8);
chess_board(0, 0) = MATRIX::Chess_piece(MATRIX::Type::Rook);
chess_board(0, 1) = MATRIX::Chess_piece(MATRIX::Type::Knight);
chess_board.print();
```

Output:



```
R N P P P P P P
P P P P P P P P
P P P P P P P P
P P P P P P P P
P P P P P P P P
P P P P P P P P
P P P P P P P P
P P P P P P P P
```

Table 1 Code Examples of Matrix implementation

2.2 Design Considerations

The following decisions were made during the design and implementation:

- ***Useage of Templates and Concepts***: The matrix class is a template based class, which can handle any type of T. Concepts are used as a constraining method, for mathematical operations.
- ***Memory Safety***: The matrix uses std::vector to manage dynamic memory useage, in this way the low-level memory management is avoided. This prevent undefined behavior and memory leakage risks.
- Exception Handling: The class is able to throw out of range for invalid indices and invalid arguments for dimension mismatch.
- ***Generic Operations***: The matrix is able to do the basic operations and is using operator semantics of C++ language.
- **Applicational Example**: A minimal version of chess board is created to demonstrate the matrix class’s ability to handle other datatypes.

3. Conclusion

This exercise demonstrates the power and flexibility of C++'s generic programming capabilities through the implementation of a flexible and efficient two-dimensional matrix class, Matrix<T>. By leveraging templates and concepts, the Matrix class supports a wide range of element types, including integers, strings, and user-defined types like Chess_piece.

This is functionally tested and given as examples in this report.

Overall, this assignment illustrates how modern C++ features can be utilized to create highly flexible and efficient data structures. The implementation not only meets the requirements but also highlights the advantages of generic programming in creating reusable and robust code. The accompanying README and code examples provide further insights and detailed usage instructions, ensuring clarity and ease of understanding for future reference and application.

Appendix A - Compiler & Build details

- **Operating System**: Windows 11 Pro, version 23H2, build 22631.3880.
- **Compiler**: I used the GCC 13.2.0 with MinGW, and I enforced using C++ 20 standard in the configuration.
- **Standard**: C++ 20
- **Build-Type:** in release