CS205 C/ C++ Programming - Project 5: : A Class for Matrices implement by C++

Name: Lv Yue
SID: 11710420

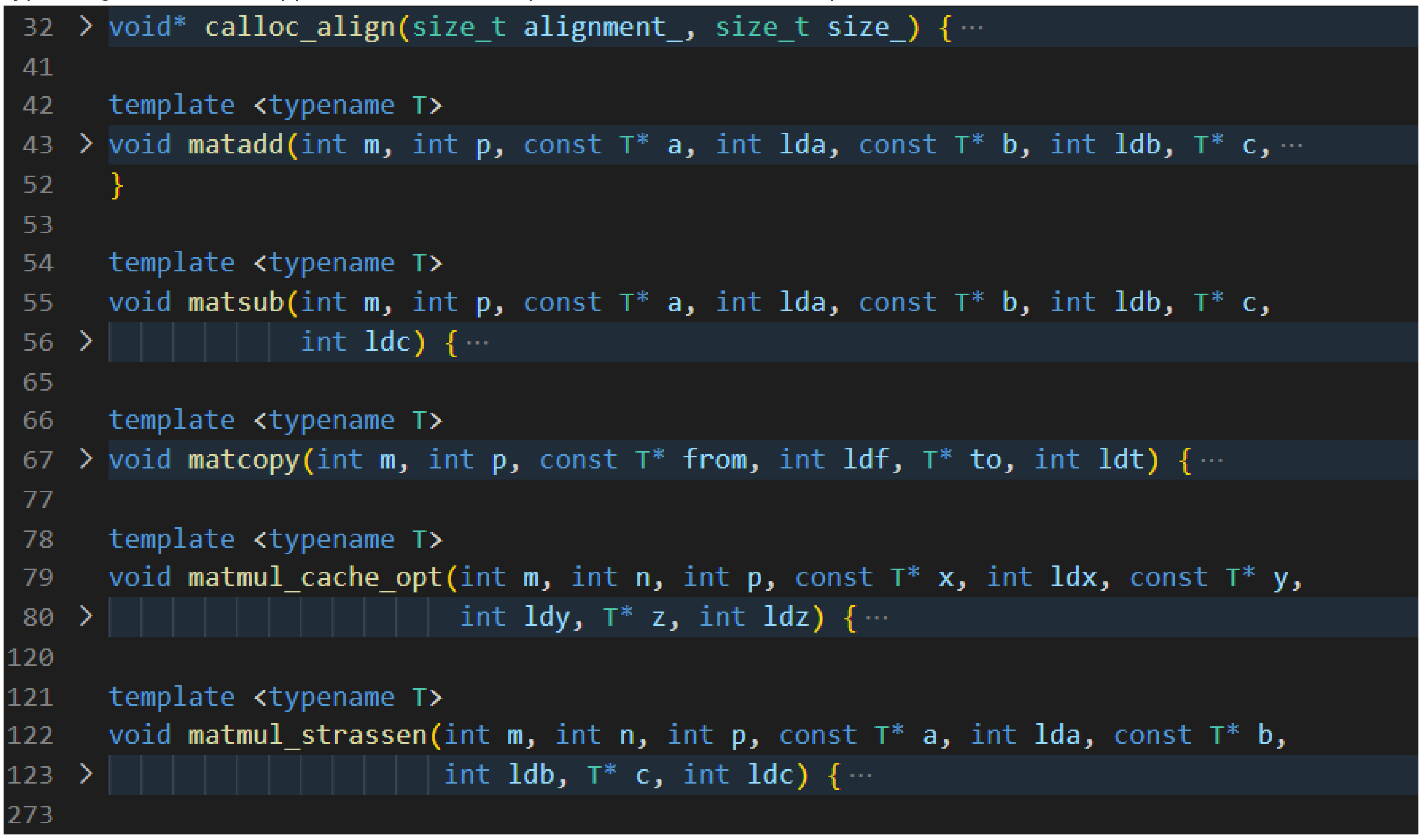
***This project is hosted at https://github.com/JustLittleFive/CppMatrix

Project goal: Implement a matrix class using C++

Core code

There are only 2 files in my implementation.

`fastMul.hpp` contains the fast multiplication of project 4 in C style. It still works efficiently in C++ projects. To support various data types, algorithms and support functions are implemented as function templates.



```cpp
void* calloc_align(size_t alignment_, size_t size_) {···

template <typename T>
void matadd(int m, int p, const T* a, int lda, const T* b, int ldb, T* c,···
}

template <typename T>
void matsub(int m, int p, const T* a, int lda, const T* b, int ldb, T* c,
            int ldc) {···

template <typename T>
void matcopy(int m, int p, const T* from, int ldf, T* to, int ldt) {···

template <typename T>
void matmul_cache_opt(int m, int n, int p, const T* x, int ldx, const T* y,
                      int ldy, T* z, int ldz) {···

template <typename T>
void matmul_strassen(int m, int n, int p, const T* a, int lda, const T* b,
                     int ldb, T* c, int ldc) {···
```

`matrix.cpp` is the main file of this project. In order to support multiple data types and avoid possible memory conflicts caused by union structures, templates are used to define classes and most functions.

First, it implemented a class called `Data` which contains 2 elements: `used` and `__T*`.

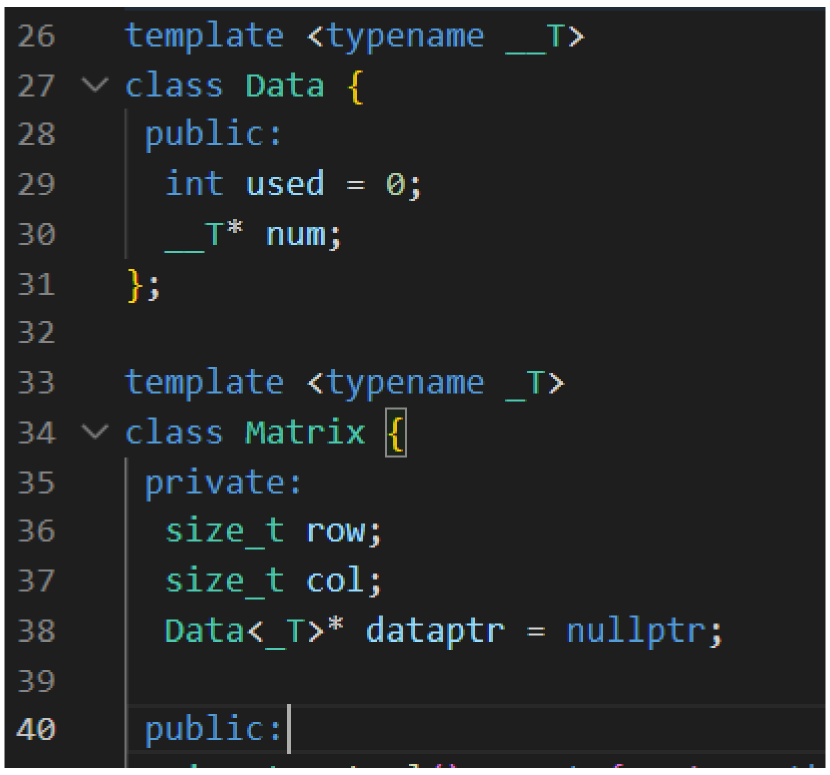
`used` is the element that records how many times this object has been referenced, which is used when creating, copying, and deleting objects.

`__T*` points to the memory where the actual data is stored.

The core class of the project is the `Matrix` class.

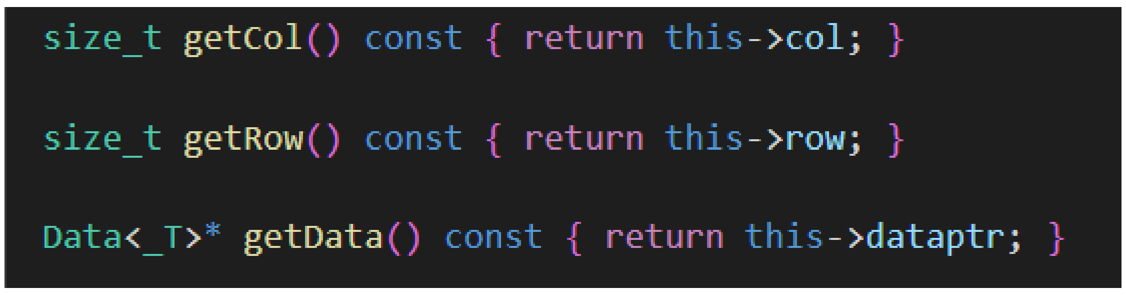
It contains 3 private element: `row`, `col` and `Data` pointer `dataptr`.

```
Hint: In fact, the *Data* class should be a substructure of *Matrix*. However, considering that the class
template will actually create independent and absolute classes while compiling, in order to avoid repeated
creation of the same type of *Data* class, the implementation handles the two separately.
```



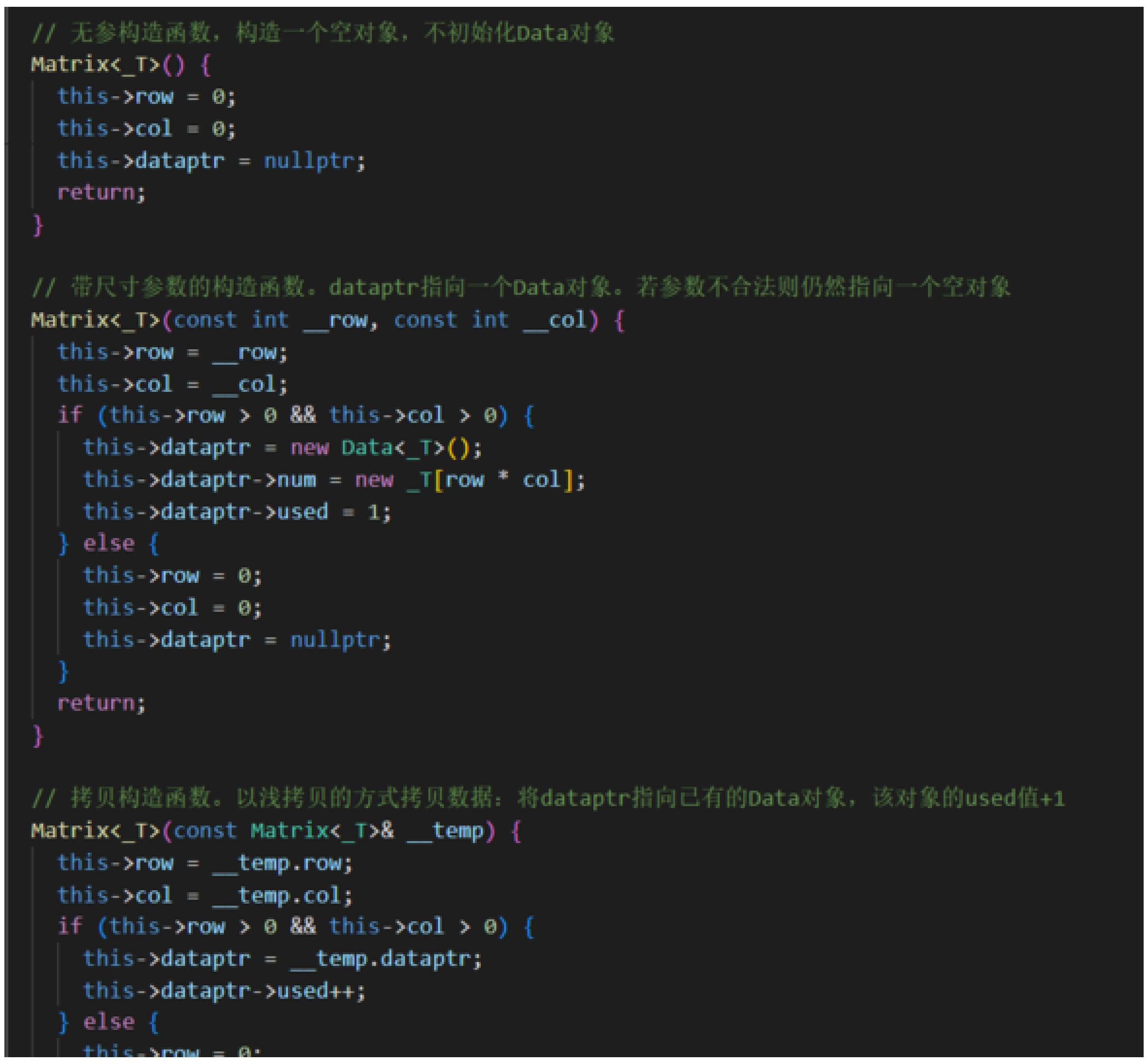
```cpp
template <typename __T>
class Data {
  public:
   int used = 0;
   __T* num;
};

template <typename _T>
class Matrix {
  private:
   size_t row;
   size_t col;
   Data<_T>* dataptr = nullptr;

  public:
```

The first public part of `Matrix` class are the function to access the private elements.



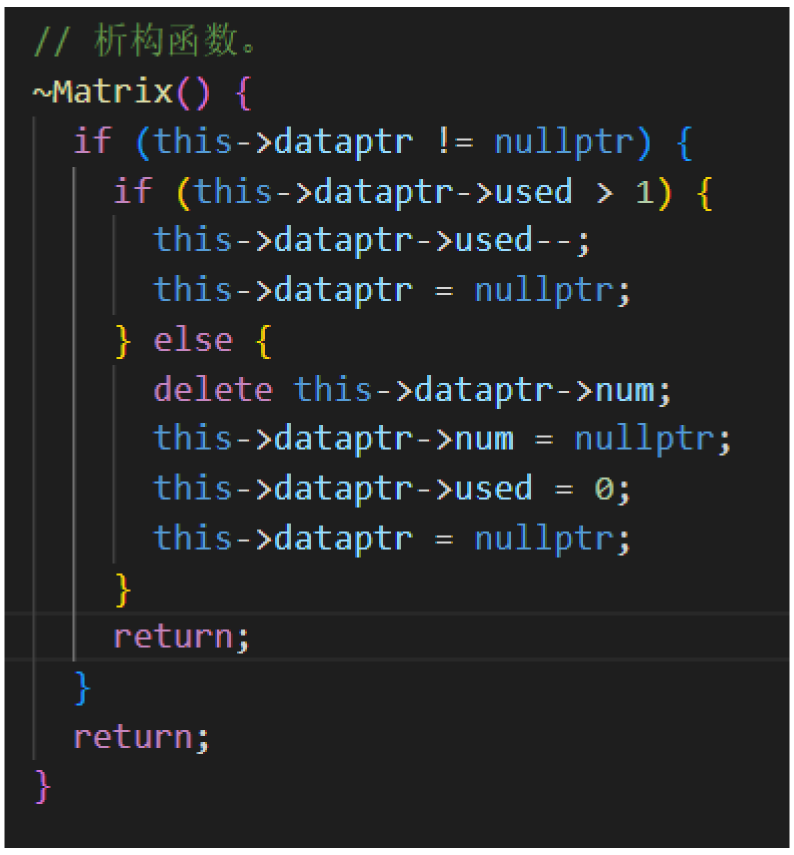
```cpp
size_t getCol() const { return this->col; }

size_t getRow() const { return this->row; }

Data<_T>* getData() const { return this->dataptr; }
```

Next is the constructor with various parameters.



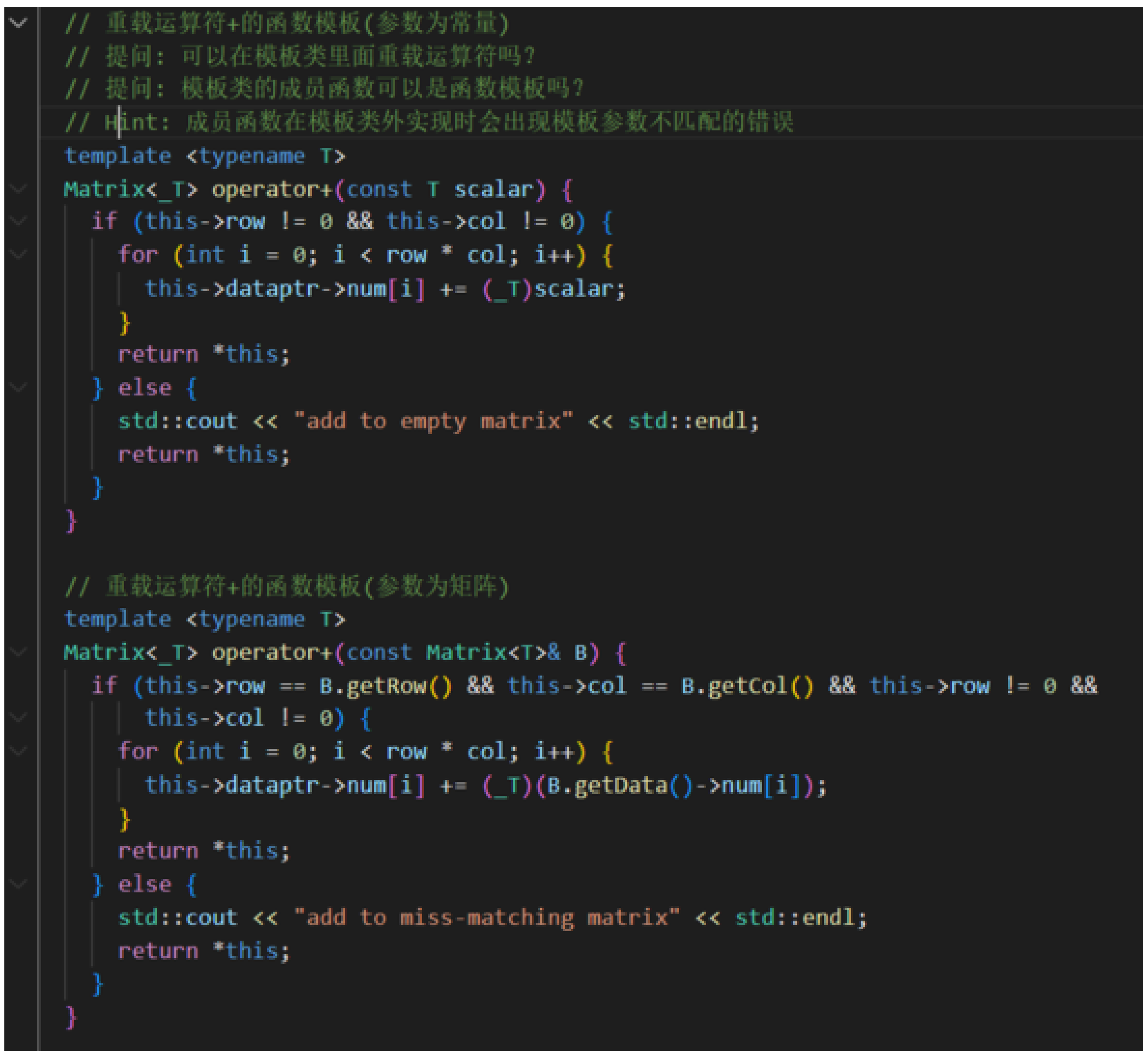
```cpp
// 无参构造函数，构造一个空对象，不初始化Data对象
Matrix<_T>() {
  this->row = 0;
  this->col = 0;
  this->dataptr = nullptr;
  return;
}

// 带尺寸参数的构造函数。dataptr指向一个Data对象。若参数不合法则仍然指向一个空对象
Matrix<_T>(const int __row, const int __col) {
  this->row = __row;
  this->col = __col;
  if (this->row > 0 && this->col > 0) {
    this->dataptr = new Data<_T>();
    this->dataptr->num = new _T[row * col];
    this->dataptr->used = 1;
  } else {
    this->row = 0;
    this->col = 0;
    this->dataptr = nullptr;
  }
  return;
}

// 拷贝构造函数。以浅拷贝的方式拷贝数据：将dataptr指向已有的Data对象，该对象的used值+1
Matrix<_T>(const Matrix<_T>& __temp) {
  this->row = __temp.row;
  this->col = __temp.col;
  if (this->row > 0 && this->col > 0) {
    this->dataptr = __temp.dataptr;
    this->dataptr->used++;
  } else {
```

The destructor. The memory management of the `Data` class is implemented directly here.

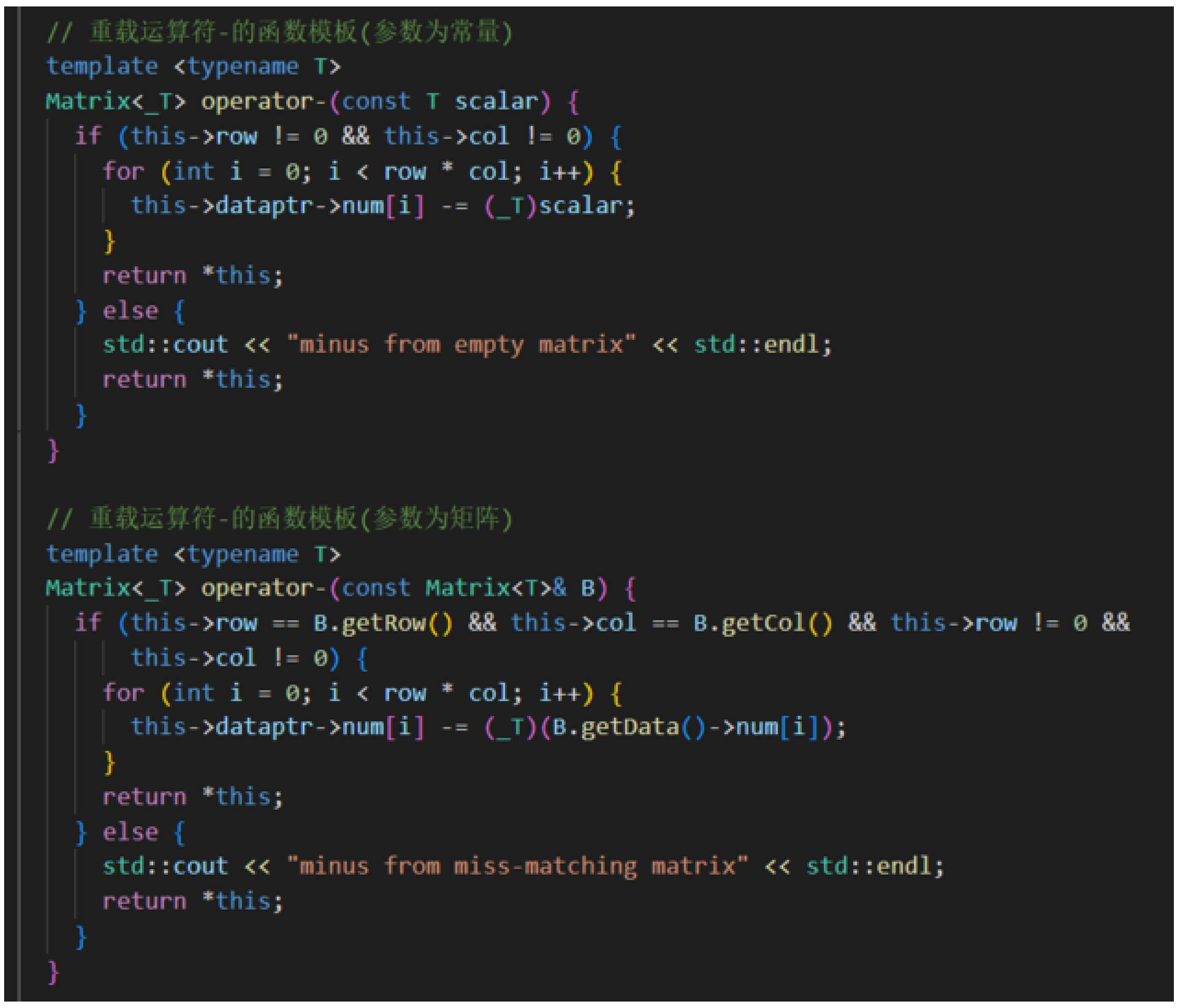


```cpp
// 析构函数。
~Matrix() {
  if (this->dataptr != nullptr) {
    if (this->dataptr->used > 1) {
      this->dataptr->used--;
      this->dataptr = nullptr;
    } else {
      delete this->dataptr->num;
      this->dataptr->num = nullptr;
      this->dataptr->used = 0;
      this->dataptr = nullptr;
    }
    return;
  }
  return;
}
```

Overloaded functions for binary operators, some are function templates for different types of coercion.

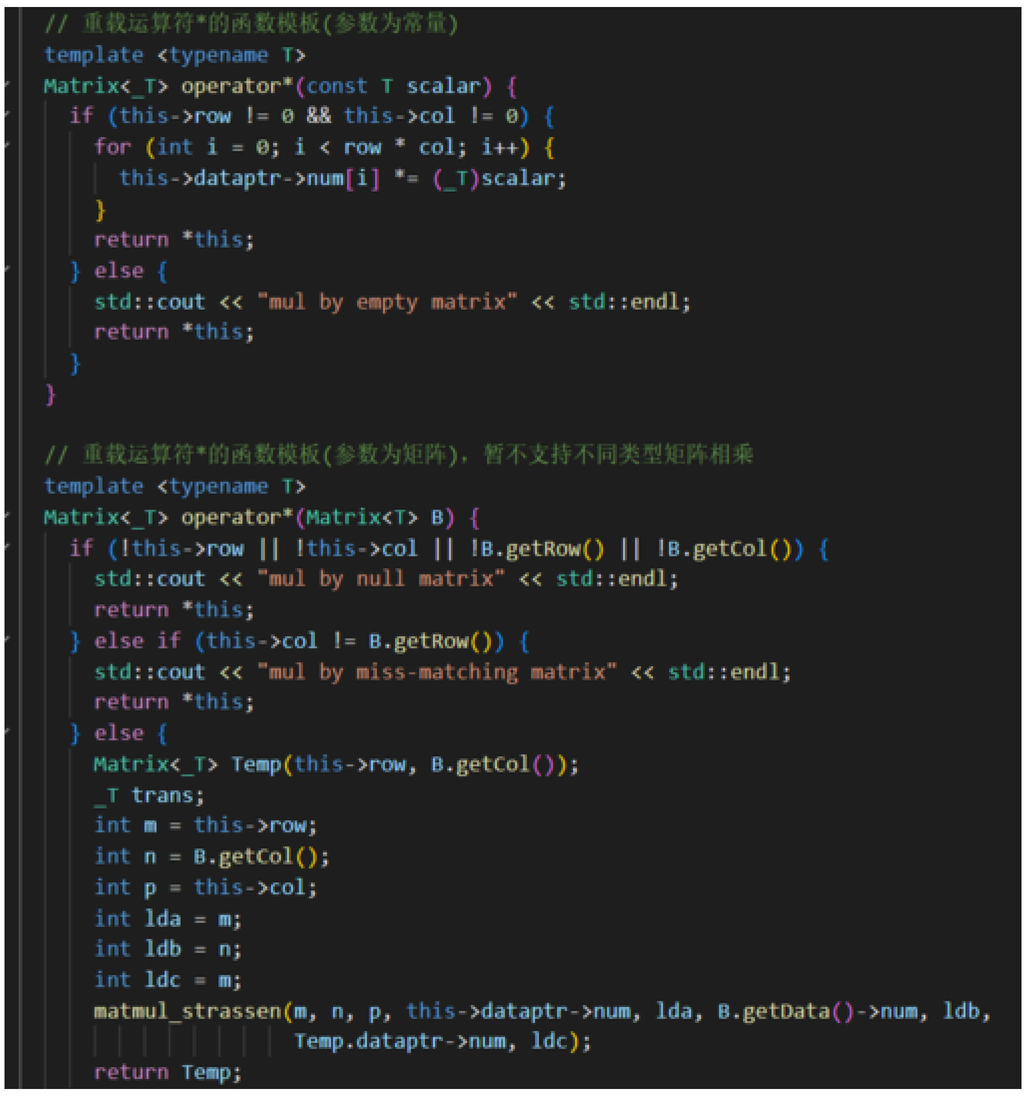


```cpp
// 重载运算符+的函数模板(参数为常量)
// 提问: 可以在模板类里面重载运算符吗?
// 提问: 模板类的成员函数可以是函数模板吗?
// Hint: 成员函数在模板类外实现时会出现模板参数不匹配的错误
template <typename T>
Matrix<_T> operator+(const T scalar) {
  if (this->row != 0 && this->col != 0) {
    for (int i = 0; i < row * col; i++) {
      this->dataptr->num[i] += (_T)scalar;
    }
    return *this;
  } else {
    std::cout << "add to empty matrix" << std::endl;
    return *this;
  }
}

// 重载运算符+的函数模板(参数为矩阵)
template <typename T>
Matrix<_T> operator+(const Matrix<T>& B) {
  if (this->row == B.getRow() && this->col == B.getCol() && this->row != 0 &&
      this->col != 0) {
    for (int i = 0; i < row * col; i++) {
      this->dataptr->num[i] += (_T)(B.getData()->num[i]);
    }
    return *this;
  } else {
    std::cout << "add to miss-matching matrix" << std::endl;
    return *this;
  }
}
```



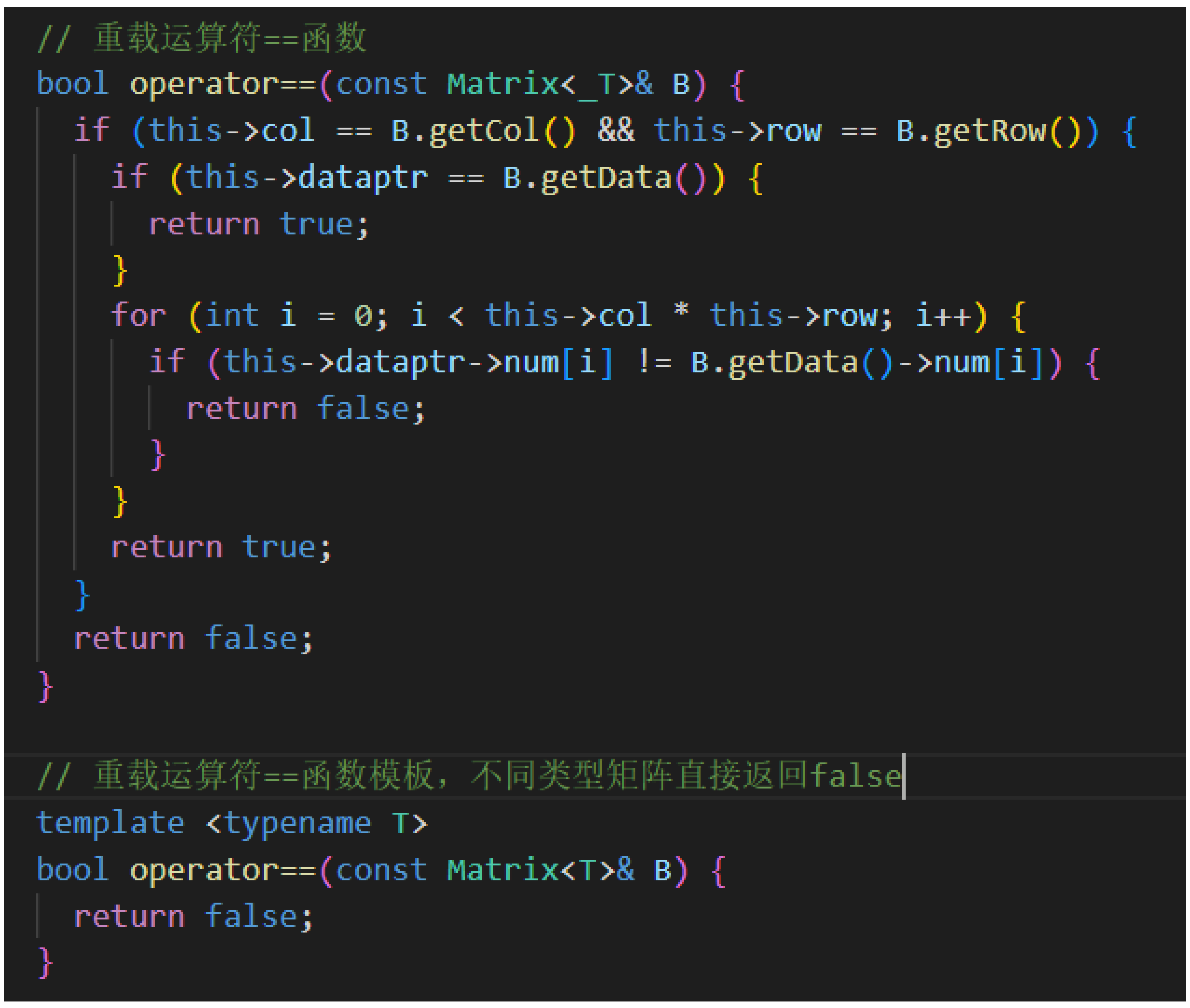
```cpp
// 重载运算符-的函数模板(参数为常量)
template <typename T>
Matrix<_T> operator-(const T scalar) {
  if (this->row != 0 && this->col != 0) {
    for (int i = 0; i < row * col; i++) {
      this->dataptr->num[i] -= (_T)scalar;
    }
    return *this;
  } else {
    std::cout << "minus from empty matrix" << std::endl;
    return *this;
  }
}

// 重载运算符-的函数模板(参数为矩阵)
template <typename T>
Matrix<_T> operator-(const Matrix<T>& B) {
  if (this->row == B.getRow() && this->col == B.getCol() && this->row != 0 &&
      this->col != 0) {
    for (int i = 0; i < row * col; i++) {
      this->dataptr->num[i] -= (_T)(B.getData()->num[i]);
    }
    return *this;
  } else {
    std::cout << "minus from miss-matching matrix" << std::endl;
    return *this;
  }
}
```

Since fast multiplication requires memory alignment, multiplication of different types of matrices is temporarily not supported.



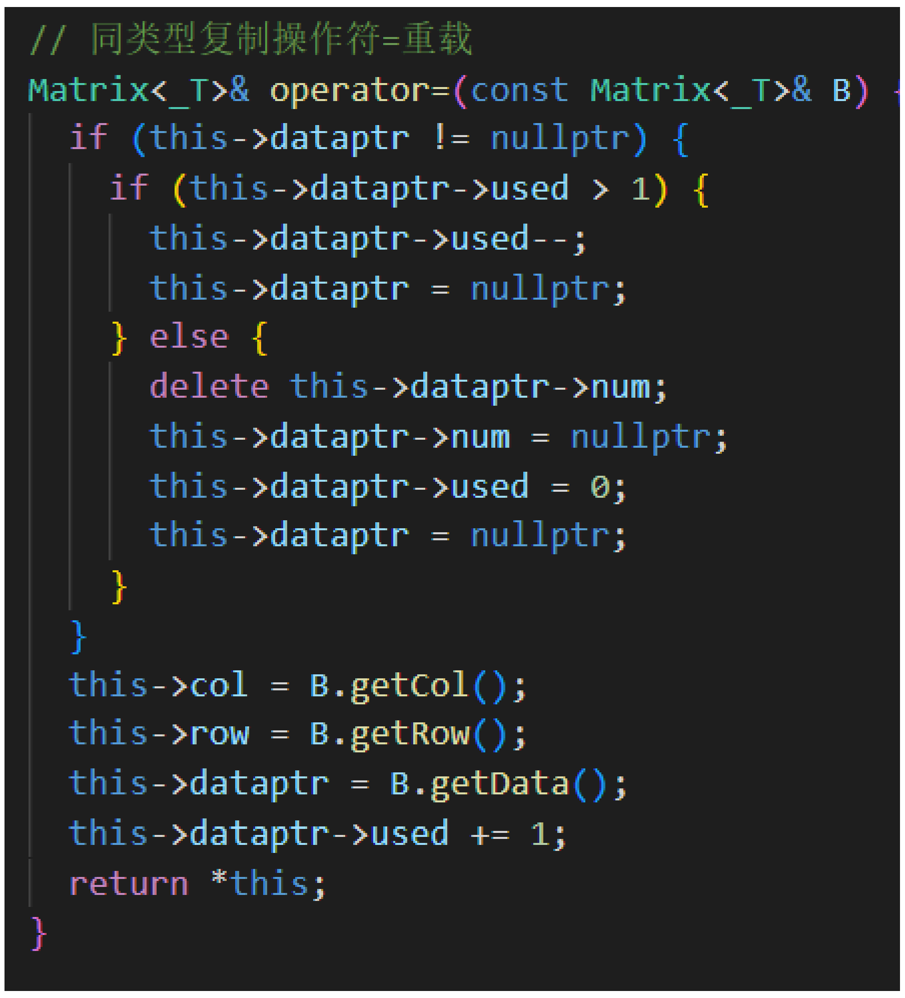
```cpp
// 重载运算符*的函数模板(参数为常量)
template <typename T>
Matrix<_T> operator*(const T scalar) {
  if (this->row != 0 && this->col != 0) {
    for (int i = 0; i < row * col; i++) {
      this->dataptr->num[i] *= (_T)scalar;
    }
    return *this;
  } else {
    std::cout << "mul by empty matrix" << std::endl;
    return *this;
  }
}

// 重载运算符*的函数模板(参数为矩阵), 暂不支持不同类型矩阵相乘
template <typename T>
Matrix<_T> operator*(Matrix<T> B) {
  if (!this->row || !this->col || !B.getRow() || !B.getCol()) {
    std::cout << "mul by null matrix" << std::endl;
    return *this;
  } else if (this->col != B.getRow()) {
    std::cout << "mul by miss-matching matrix" << std::endl;
    return *this;
  } else {
    Matrix<_T> Temp(this->row, B.getCol());
    _T trans;
    int m = this->row;
    int n = B.getCol();
    int p = this->col;
    int lda = m;
    int ldb = n;
    int ldc = m;
    matmul_strassen(m, n, p, this->dataptr->num, lda, B.getData()->num, ldb,
                    Temp.dataptr->num, ldc);
    return Temp;
```

The overloading of comparison operators is divided into two types: matrices of the same type compare elements one by one, matrices of different types directly return false. The latter one is implemented through function templates to support multiple types.

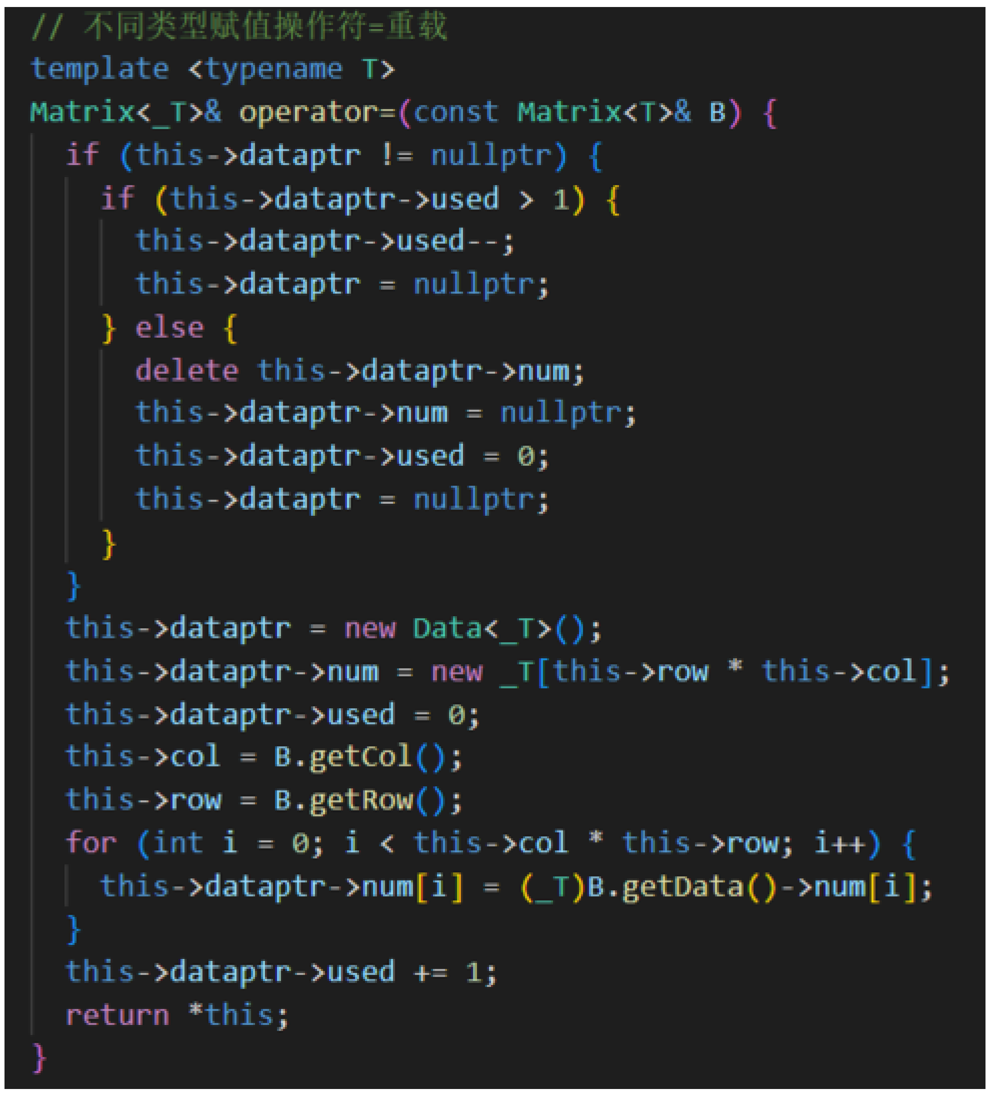


```
// 重载运算符==函数
bool operator==(const Matrix<_T>& B) {
    if (this->col == B.getCol() && this->row == B.getRow()) {
        if (this->dataptr == B.getData()) {
            return true;
        }
        for (int i = 0; i < this->col * this->row; i++) {
            if (this->dataptr->num[i] != B.getData()->num[i]) {
                return false;
            }
        }
        return true;
    }
    return false;
}

// 重载运算符==函数模板，不同类型矩阵直接返回false
template <typename T>
bool operator==(const Matrix<T>& B) {
    return false;
}
```

The same is true for assignment operator overloading, which can be implemented in two ways.
When assigning different types of matrices, take elements one by one and perform mandatory type conversion, because the memory sizes used by different data types are different.

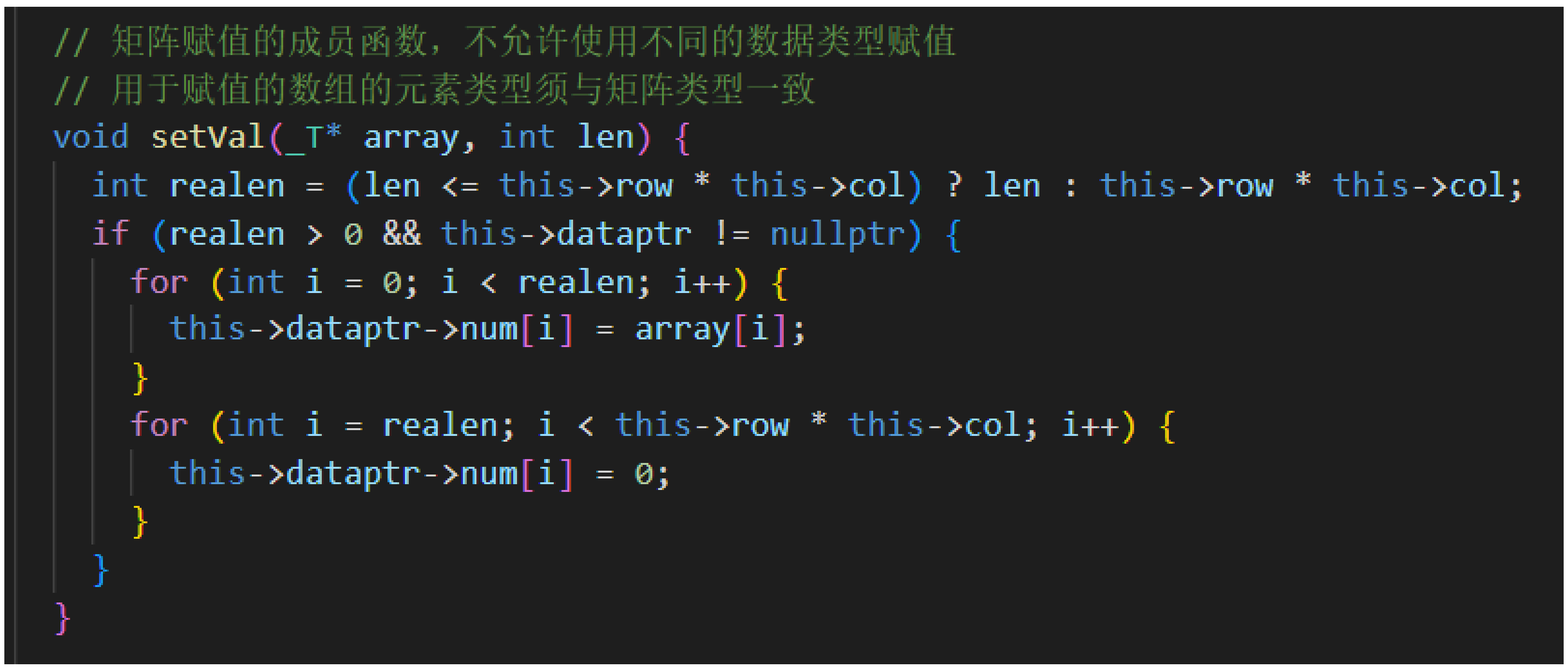


```
// 同类型复制操作符=重载
Matrix<_T>& operator=(const Matrix<_T>& B)
    if (this->dataptr != nullptr) {
        if (this->dataptr->used > 1) {
            this->dataptr->used--;
            this->dataptr = nullptr;
        } else {
            delete this->dataptr->num;
            this->dataptr->num = nullptr;
            this->dataptr->used = 0;
            this->dataptr = nullptr;
        }
    }
    this->col = B.getCol();
    this->row = B.getRow();
    this->dataptr = B.getData();
    this->dataptr->used += 1;
    return *this;
}
```



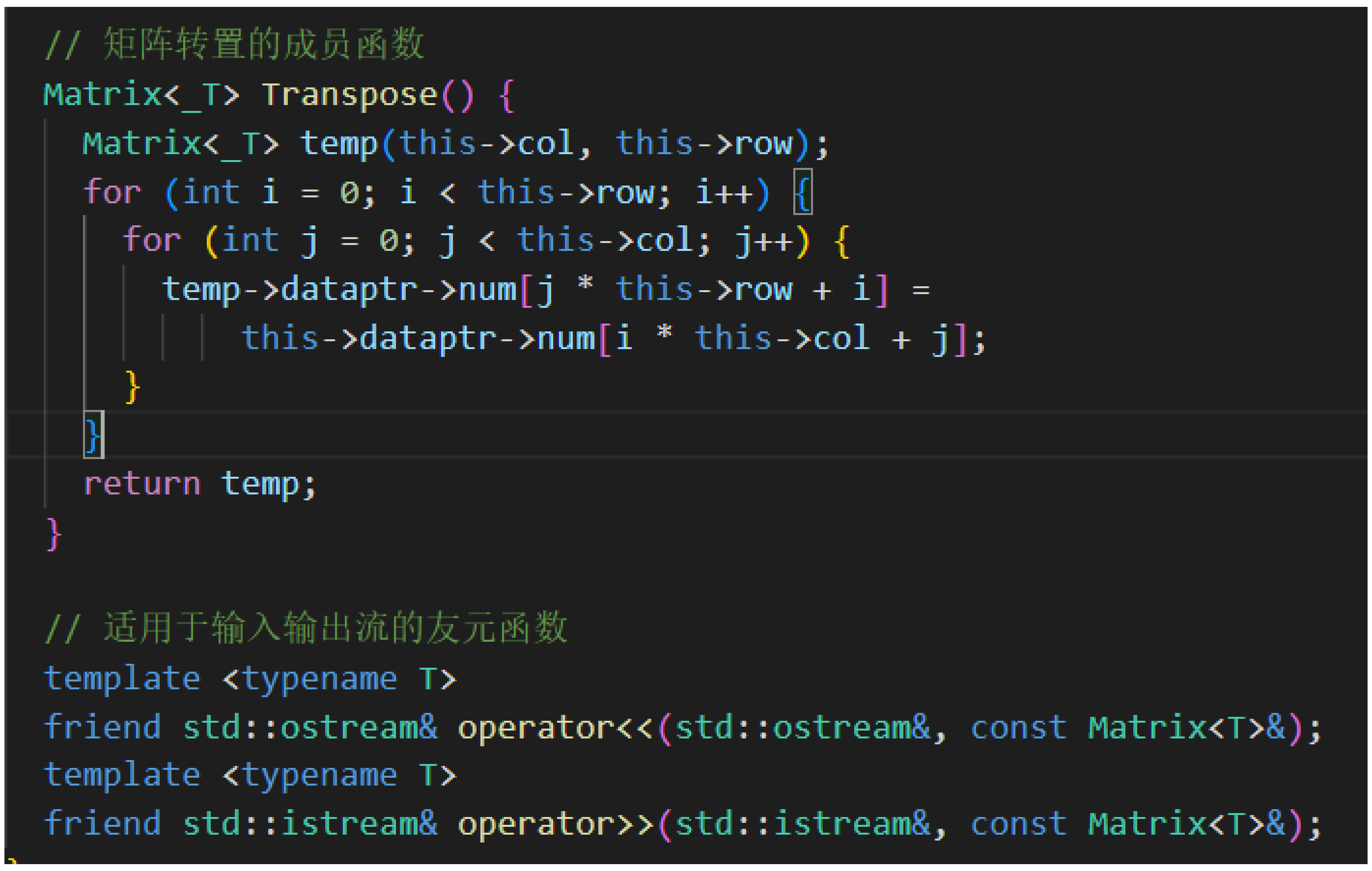
```
// 不同类型赋值操作符=重载
template <typename T>
Matrix<_T>& operator=(const Matrix<T>& B) {
  if (this->dataptr != nullptr) {
    if (this->dataptr->used > 1) {
      this->dataptr->used--;
      this->dataptr = nullptr;
    } else {
      delete this->dataptr->num;
      this->dataptr->num = nullptr;
      this->dataptr->used = 0;
      this->dataptr = nullptr;
    }
  }
  this->dataptr = new Data<_T>();
  this->dataptr->num = new _T[this->row * this->col];
  this->dataptr->used = 0;
  this->col = B.getCol();
  this->row = B.getRow();
  for (int i = 0; i < this->col * this->row; i++) {
    this->dataptr->num[i] = (_T)B.getData()->num[i];
  }
  this->dataptr->used += 1;
  return *this;
}
```

For the same reason, matrix assignment functions do not allow assignments to matrices using arrays of different types (directly disallowed by compilation).



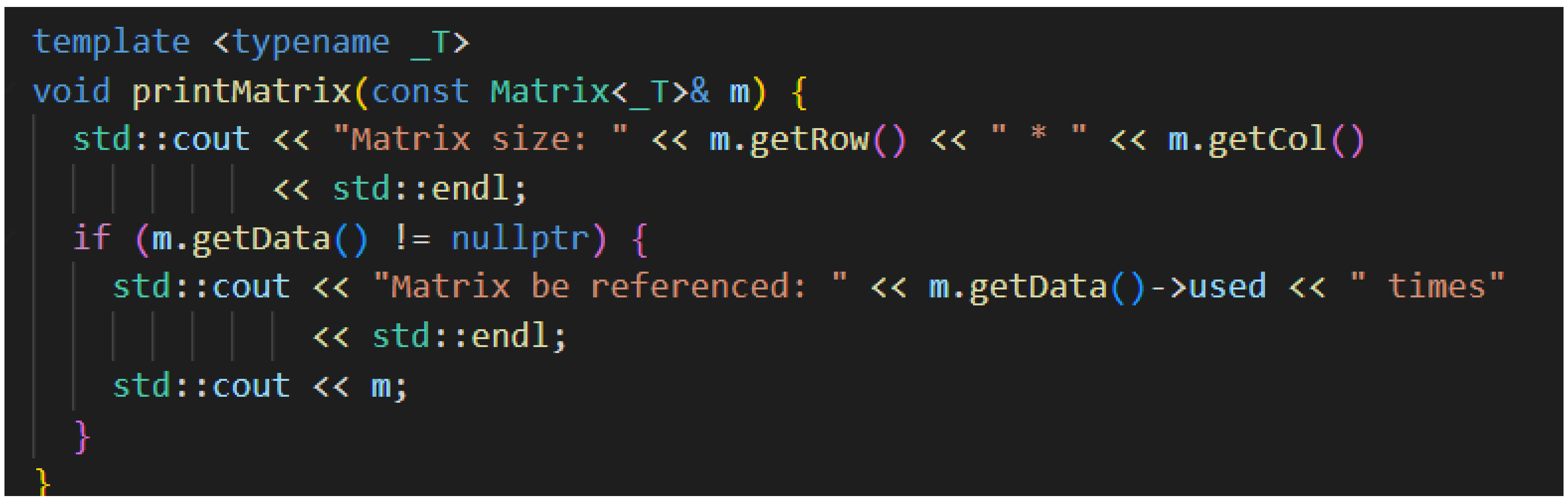
```
// 矩阵赋值的成员函数，不允许使用不同的数据类型赋值
// 用于赋值的数组的元素类型须与矩阵类型一致
void setVal(_T* array, int len) {
  int realen = (len <= this->row * this->col) ? len : this->row * this->col;
  if (realen > 0 && this->dataptr != nullptr) {
    for (int i = 0; i < realen; i++) {
      this->dataptr->num[i] = array[i];
    }
    for (int i = realen; i < this->row * this->col; i++) {
      this->dataptr->num[i] = 0;
    }
  }
}
```

The rest are the matrix transpose function and the input and output friend functions.



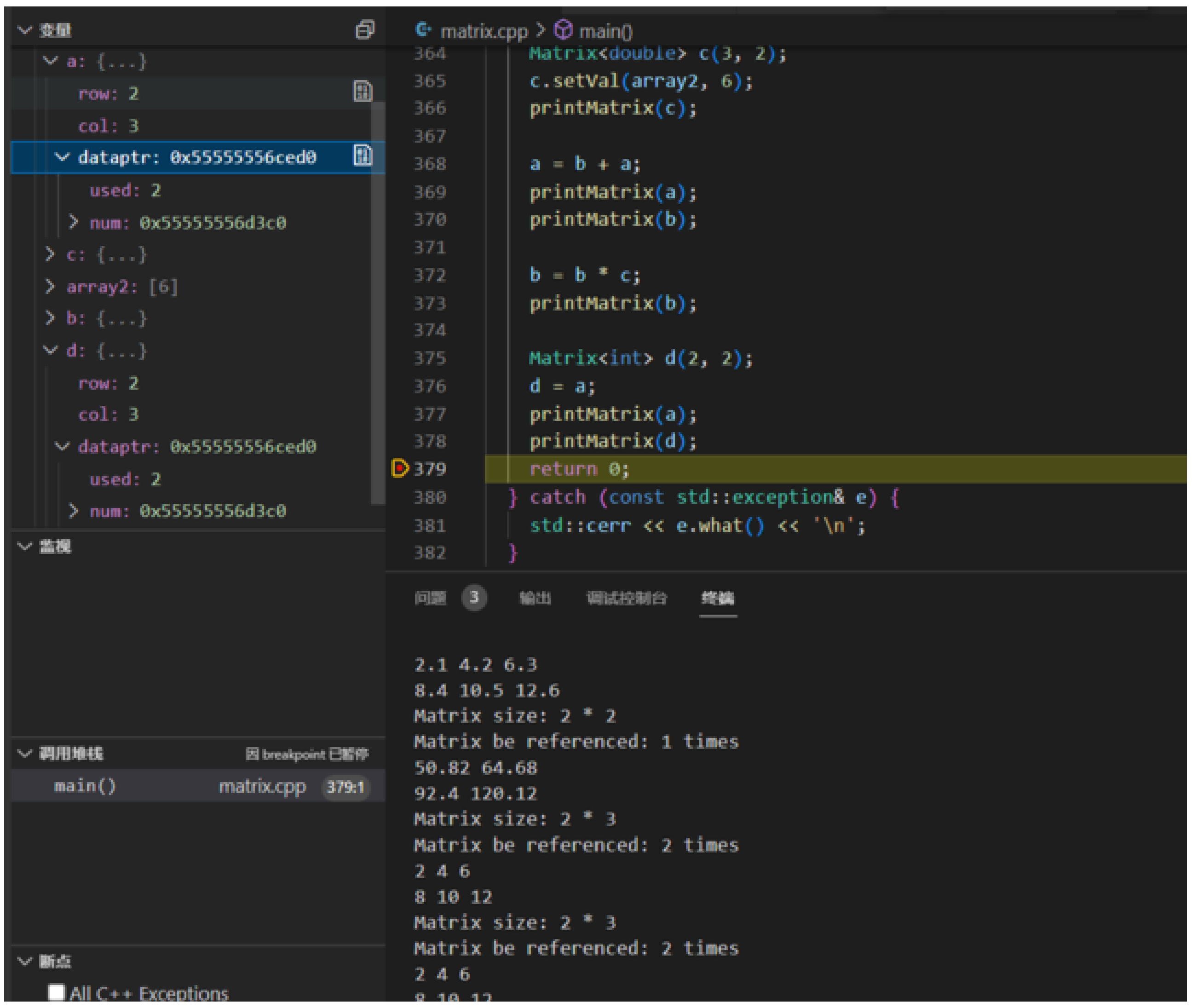
```
// 矩阵转置的成员函数
Matrix<_T> Transpose() {
  Matrix<_T> temp(this->col, this->row);
  for (int i = 0; i < this->row; i++) {
    for (int j = 0; j < this->col; j++) {
      temp->dataptr->num[j * this->row + i] =
          this->dataptr->num[i * this->col + j];
    }
  }
  return temp;
}

// 适用于输入输出流的友元函数
template <typename T>
friend std::ostream& operator<<(std::ostream&, const Matrix<T>&);
template <typename T>
friend std::istream& operator>>(std::istream&, const Matrix<T>&);
```

A function template for printing matrices is defined outside the Matrix class template for testing. This function can also be a member function.



```cpp
template <typename _T>
void printMatrix(const Matrix<_T>& m) {
    std::cout << "Matrix size: " << m.getRow() << " * " << m.getCol()
              << std::endl;
    if (m.getData() != nullptr) {
        std::cout << "Matrix be referenced: " << m.getData()->used << " times"
                  << std::endl;
        std::cout << m;
    }
}
```

The main function is written at the end of the file for testing.



```
变量
  a: {...}
    row: 2
    col: 3
    dataptr: 0x55555556ced0
      used: 2
      num: 0x55555556d3c0
  c: {...}
  array2: [6]
  b: {...}
  d: {...}
    row: 2
    col: 3
    dataptr: 0x55555556ced0
      used: 2
      num: 0x55555556d3c0
监视
调用堆栈                breakpoint 已暂停
  main()        matrix.cpp   379:1
断点
  All C++ Exceptions
```

```cpp
        Matrix<double> c(3, 2);
        c.setVal(array2, 6);
        printMatrix(c);

        a = b + a;
        printMatrix(a);
        printMatrix(b);

        b = b * c;
        printMatrix(b);

        Matrix<int> d(2, 2);
        d = a;
        printMatrix(a);
        printMatrix(d);
        return 0;
    } catch (const std::exception& e) {
        std::cerr << e.what() << '\n';
    }
```

```
2.1 4.2 6.3
8.4 10.5 12.6
Matrix size: 2 * 2
Matrix be referenced: 1 times
50.82 64.68
92.4 120.12
Matrix size: 2 * 3
Matrix be referenced: 2 times
2 4 6
8 10 12
Matrix size: 2 * 3
Matrix be referenced: 2 times
2 4 6
8 10 12
```

The performance of the program is as expected, and the debugger also shows that memory reuse (soft copy) has been successfully implemented.

About Region of Interest (ROI):
I don't think sloppy implementation of the `ROI` function without more wrappers is a good idea. Unless the output of the `ROI` function is set to `void`, the user will gain the ability to directly operate of memory where data stored as a one-dimensional array. This may cause the storage structure broken when the user modifies this piece of memory, causing unexpected problems when the destructor function is called, and may even cause a ***memory leak***.

Difficulties & Solutions

A member function of a class template cannot be defined as a function template outside the template. The compiler will complain `too many template parameters in template redeclaration template`.
My guess is that the compiler cannot know the template parameter information of the template class to which the externally defined function belongs.
At the same time, the compiler does not allow me to directly use template parameters to assign or compare pointers through pre-judgment. The compiler will complain about`comparison of distinct pointer types`or`Distinct pointer types lacks a cast`, even though my tests have determined that no assignment or comparison will be performed on arguments of different types.
Therefore, I have to define some member functions of class templates into two categories: member functions that only handle parameters of the same type and function templates that adapt to other types, as I mentioned above.

Still, I have a lot of unanswered questions, which are listed in the code comments.