基于 MyVector 类的向量操作工具课程项目

邱姜铭 卫祎懿 刘逸儒 陆宇涵

(上海大学 计算机工程与科学学院)

摘　要　我们编写了基于自定义向量模板类‘MyVector’的项目。通过运算符重载，该系统提供了丰富的向量操作功能，以便轻松处理向量数据。测试结果证明，‘MyVector’类为向量操作提供了一个强大的工具，有望提高向量处理的效率和灵活性

关键词　自定义向量类；向量操作；模板类；运算符重载

A Course Task Management System Based on Vector Class

Qiu Jiangming, Wei Yiyi, Liu Yiru, Lu Yuhan

(School of Computer Engineering and Science, Shanghai University)

Abstract　According to the project objectives, This report introduces a project based on a custom string class, MyString, which provides extensive string manipulation capabilities and operator overloading for easier handling and manipulation of string data. The core functionalities of the project include string concatenation, substring searching, string replacement, case conversion, and operator overloading. Additionally, the class includes several other features such as string clearing, string copying, and retrieving string length. Through efficient memory management and friend functions, the MyString class offers a powerful tool for string manipulation, promising to enhance the efficiency and flexibility of string processing.

Key words Custom vector class, vector manipulation, template class, operator reload.

目录

1 项目整体介绍

1.1 文件结构说明

项目主要文件有 CMakeLists.txt, my_assert.h, MyVector.h, MyVectorTest.cpp。CMakeLists.txt 是配置文件; MyVector.h 里定义了模板类的私有数据成员和成员函数, 是对 MyVector 模板类的封装; my_assert.h 定义了异常处理的方法; MyVectorTest.cpp 是主程序的代码部分。

1.2 程序清单

1.2.1 数据成员设计

```
value_type *data;     //向量存储的数据类型

size_type dim;        //向量维度数据的数据类型，为 int
                      //而为了易读性，使用 typedef int size_type
```

1.2.2 成员函数（友元函数）原型设计

序号	函数名	作用
构造函数、析构函数		
1	`MyVector(size_type dim, value_type *&&data);`	转换构造函数
2	`MyVector(size_type dim, const value_type *data);`	指定长度的构造转换函数
3	`MyVector(MyVector &&v) noexcept;`	移动构造函数
4	`MyVector(const MyVector &v);`	（深）拷贝构造函数
5	`MyVector();`	默认构造函数
6	`~MyVector();`	析构函数
运算符重载		
7	`friend std::ostream &operator<<(std::ostream &os, const MyVector<T> &v)`	输出流重载
8	`std::istream &operator>>(std::istream &is);`	输入流重载
9	`friend std::istream &operator>>(std::istream &is, MyVector<T> &v)`	针对 MyVector 类的输入流重载
10	`bool operator==(const MyVector &v) const;`	重载比较运算符
11	`bool operator!=(const MyVector &v) const;`	重载比较运算符
12	`MyVector &operator=(const MyVector &v);`	重载赋值运算符
13	`MyVector &operator=(MyVector &&v) noexcept;`	重载赋值运算符，阻止异常传播

14	`MyVector operator+(const MyVector &v) const;`	重载加减运算
15	`MyVector &operator+=(const MyVector &v);`	
16	`MyVector operator-(const MyVector &v) const;`	
17	`MyVector operator-() const;`	
18	`MyVector &operator-=(const MyVector &v);`	
19	`value_type operator*(const MyVector &v) const;`	重载点乘
20	`MyVector operator*(value_type k) const;`	重载数乘
21	`MyVector &operator*=(value_type k);`	
22	`value_type &operator[](size_type i);`	重载下标访问
23	`const value_type &operator[](size_type i) const;`	
功能类函数		
22	`void checkDim(const MyVector &v) const noexcept(false);`	检查维数
23	`size_type getDim() const;`	获取维数
结果判断		
24	`void my_assert_impl(T actual, T expected)`	普通判断
25	`void my_assert_impl(T actual, std::string_view expected)`	针对输入类型为 string 的情况
26	`my_assert(actual, expected)`	宏定义，方便在语句的执行过程中判断
异常判断		
27	`my_assert_throws(x, exception)`	语句执行过程中捕获异常

2 函数实现的设计

2.1类成员函数

2. 1. 1 构造函数

在本项目中，有一个默认构造函数、一个移动构造函数、一个拷贝构造函数 ，它们适用于用不同的方法创建向量。

```
template<class T>//默认构造函数创建空向量
MyVector<T>::MyVector() {
    this->dim = 0;
    this->data = new value_type[0]{};
}

template<class T>//指定维数，和每个维数上的大小（变量传入）
MyVector<T>::MyVector(size_type dim, value_type *&&data) {
```

```
    if (dim < 0) {
        throw std::invalid_argument("Dimension must be non-negative");
    }
    this->dim = dim;
    this->data = data;
}

template<class T>//指定维数，和每个维数上的大小（常量传入）
MyVector<T>::MyVector(size_type dim, const value_type *data) {
    if (dim < 0) {
        throw std::invalid_argument("Dimension must be non-negative");
    }
    this->dim = dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = data[i];
    }
}

template<class T>//拷贝构造函数，字面意思
MyVector<T>::MyVector(const MyVector &v) {
    this->dim = v.dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = v.data[i];
    }
}

template<class T>//移动构造函数
MyVector<T>::MyVector(MyVector &&v) noexcept {
    this->dim = v.dim;
    this->data = v.data;
    v.data = nullptr;
    v.dim = 0;
}
```

2.1.2 运算符重载

1. 输入输出流符号>>，<<

```
template<class T>
std::istream &MyVector<T>::operator>>(std::istream &is) {
    for (size_type i = 0; i < dim; ++i) {
        is >> data[i];
    }
    return is;
}
template<class T>
std::ostream &MyVector<T>::operator<<(std::ostream &os) const {
    os << "(";
    for (size_type i = 0; i < dim; ++i) {
```

```
        os << data[i];
        if (i != dim - 1) {
            os << ", ";
        }
    }
    os << ")";
    return os;
}
```

2. 比较运算符==、!=

比较相等时，维数、数值都要判断是否相等。不等于运算则直接返回==运算的反结果。

```
template<class T>
bool MyVector<T>::operator==(const MyVector &v) const {
    if (dim != v.dim) {
        return false;
    }
    for (size_type i = 0; i < dim; ++i) {
        if (data[i] != v.data[i]) {
            return false;
        }
    }
    return true;
}

template<class T>
bool MyVector<T>::operator!=(const MyVector &v) const {
    return !operator==(v);
}
```

3. 赋值运算=

```
template<class T>
MyVector<T> &MyVector<T>::operator=(const MyVector &v) {
    if (this == &v) {
        return *this;
    }
    delete[] data;
    this->dim = v.dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = v.data[i];
    }
    return *this;
}

template<class T>
MyVector<T> &MyVector<T>::operator=(MyVector<T> &&v) noexcept {
```

```
    if (this == &v) {
        return *this;
    }
    this->dim = v.dim;
    this->data = v.data;
    v.dim = 0;
    v.data = nullptr;
    return *this;
}
```

4. 加减运算

```
template<class T>
MyVector<T> MyVector<T>::operator+(const MyVector &v) const {
    checkDim(v);
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] += v.data[i];
    }
    return result;
}

template<class T>
MyVector<T> &MyVector<T>::operator+=(const MyVector &v) {
    checkDim(v);
    for (size_type i = 0; i < dim; ++i) {
        data[i] += v.data[i];
    }
    return *this;
}

template<class T>
MyVector<T> MyVector<T>::operator-(const MyVector &v) const {
    checkDim(v);
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] -= v.data[i];
    }
    return result;
}

template<class T>
MyVector<T> MyVector<T>::operator-() const {
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] = -result.data[i];
    }
    return result;
}
```

```cpp
template<class T>
MyVector<T> &MyVector<T>::operator-=(const MyVector &v) {
    checkDim(v);
    for (size_type i = 0; i < dim; ++i) {
        data[i] -= v.data[i];
    }
    return *this;
}
```

5. 数乘、点乘、叉乘

先判断维数是否匹配，再进入运算

```cpp
//数乘
template<class T>
MyVector<T> MyVector<T>::operator*(value_type k) const {
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] *= k;
    }
    return result;
}

template<class T>
MyVector<T> &MyVector<T>::operator*=(value_type k) {
    for (size_type i = 0; i < dim; ++i) {
        data[i] *= k;
    }
    return *this;
}

//点乘
template<class T>
typename MyVector<T>::value_type MyVector<T>::operator*(const MyVector &v) const {
    checkDim(v);
    value_type result = 0;
    for (size_type i = 0; i < dim; ++i) {
        result += data[i] * v.data[i];
    }
    return result;
}

//叉乘
template<class T>
MyVector<T> MyVector<T>::cross(const MyVector &v) const {
    if (dim != 3 || v.dim != 3) {
        throw std::invalid_argument("Dimension mismatch");
    }
```

```
    MyVector result(3, new value_type[3]{data[1] * v.data[2] - data[2] *
                                          v.data[1],
                                         data[2] * v.data[0] - data[0] *
                                          v.data[2],
                                         data[0] * v.data[1] - data[1] *
                                          v.data[0]});
    return result;
}
```

6. 下标访问[]

两个函数，分别返回引用和常量引用。在执行时遇到异常即抛出。

```
template<class T>
typename MyVector<T>::value_type &MyVector<T>::operator[](size_type i) {
    if (i < 0 || i >= dim) {
        throw std::out_of_range("Index out of range");
    }
    return data[i];
}

template<class T>
const typename MyVector<T>::value_type &MyVector<T>::operator[](size_type i) const
{
    if (i < 0 || i >= dim) {
        throw std::out_of_range("Index out of range");
    }
    return data[i];
}
```

2.1.3 其他操作

1. 获取向量维数

```
   template<class T>
typename MyVector<T>::size_type MyVector<T>::getDim() const {
    return dim;
}
```

2. 检查维数

```
template<class T>
void MyVector<T>::checkDim(const MyVector &v) const noexcept(false) {
    if (dim != v.dim) {
        throw std::invalid_argument("Dimension mismatch");
    }
}
```

2.2异常处理测试/结果判断

2.2.1 异常处理测试：

```
#define my_assert_throws(x, exception)
std::cout << std::endl;
try {
    std::cout << print_green("Expecting Exception: ") << #exception <<
std::endl;
    run(x);
    std::cout << print_red("No Exception throws Failed") << std::endl;
} catch (exception &e) {
    std::cout << e.what() <<endl;
    std::cout << print_green("Exception Matched") << std::endl;
```

这里并没有使用函数，而是使用宏定义。因为如果将要执行的语句放进函数参数列表，得到的是语句的返回值，而不是语句的执行状态。故要么使用宏，要么使用 lambda 表达式。我们这里选择了宏定义。

2.2.2 结果判断：

```
template<typename T>
void my_assert_impl(T actual, T expected) {
    if (actual == expected) {
        std::cout << print_green(" Passed") << std::endl;
    } else {
        std::cout << print_red(" Failed") << std::endl;
    }
}

template<typename T>
void my_assert_impl(T actual, std::string_view expected) {
    std::ostringstream oss;
    oss << actual;
    if (oss.str() == expected) {
        std::cout << print_green(" Passed") << std::endl;
    } else {
        std::cout << print_red(" Failed") << std::endl;
    }
}
```

```
#define my_assert(actual, expected)
std::cout << print_green("Expected:") <<" (" << #expected <<") "
    << print_red("Actual:") <<" (" << print_yellow(#actual) << ")";
my_assert_impl<decltype(actual)>(actual, expected);
```

对 my_assert_impl（）进行重载，以应对不同的测试数据传入。此外，仍然是为了获取到语句执行的状态，这个函数还有一个宏定义版本。

3 测试结果展示

```
空向量测试
Running MyVector<int> empty_vector(0, new int[0]{})
Expected: ("()") Actual: (empty_vector) Passed
Running MyVector<double> double_vector(3, new double[3]{1.1, 2.2, 3.3})
输出测试
Expected: ("(1, 2, 3)") Actual: (v1) Passed
Expected: ("(4, 5, 6)") Actual: (v2) Passed
加减法测试
Expected: ("(5, 7, 9)") Actual: (v1 + v2) Passed
Expected: ("(-3, -3, -3)") Actual: (v1 - v2) Passed
点乘测试
Expected: (32) Actual: (v1 * v2) Passed
数乘测试
Expected: ("(2, 4, 6)") Actual: (v1 * 2) Passed
下标访问[]测试
Expected: (2) Actual: (static_cast<int>(v1[1])) Passed
下标赋值[]测试
Running v1[1] = 10
Expected: ("(1, 10, 3)") Actual: (v1) Passed
叉乘测试
Expected: ("(1, 10, 3)") Actual: (v1) Passed
Expected: ("(4, 5, 6)") Actual: (v2) Passed
Expected: ("(45, 6, -35)") Actual: (v1.cross(v2)) Passed
Expected: (-v1.cross(v2)) Actual: (v2.cross(v1)) Passed
向量拷贝测试
Running MyVector<int> v3(v1)
Expected: ("(1, 10, 3)") Actual: (v3) Passed
向量移动测试
Running MyVector<int> v4(std::move(v3))
Expected: ("(1, 10, 3)") Actual: (v4) Passed
向量赋值测试
Running v3 = v4
Expected: ("(1, 10, 3)") Actual: (v3) Passed
向量自赋值测试
Running v3 = v3
Expected: ("(1, 10, 3)") Actual: (v3) Passed
向量加减赋值测试
Running v3 += v4
Expected: ("(2, 20, 6)") Actual: (v3) Passed
Running v3 -= v4
Expected: ("(1, 10, 3)") Actual: (v3) Passed
向量数乘赋值测试
Running v3 *= 2
Expected: ("(2, 20, 6)") Actual: (v3) Passed
向量比较测试
Expected: ("(1, 10, 3)") Actual: (v1) Passed
Expected: ("(4, 5, 6)") Actual: (v2) Passed
Expected: (false) Actual: (v1 == v2) Passed
Expected: (true) Actual: (v1 != v2) Passed
```

```
向量缩放测试
Expected: (3) Actual: (v1.getDim()) Passed
Running v1.resize(5)
Expected: ("(1, 10, 3, 0, 0)") Actual: (v1) Passed
Expected: (5) Actual: (v1.getDim()) Passed
Running v1.resize(2)
Expected: ("(1, 10)") Actual: (v1) Passed
Expected: (2) Actual: (v1.getDim()) Passed
Running v1.resize(3)
Expected: ("(1, 10, 0)") Actual: (v1) Passed
Expected: (3) Actual: (v1.getDim()) Passed
Running v1.resize(0)
Expected: ("()") Actual: (v1) Passed
Expected: (0) Actual: (v1.getDim()) Passed
```

```
输入测试
Running v1.resize(2)
Running istringstream is("2 3")
Running is >> v1
Expected: ("(2, 3)") Actual: (v1) Passed
异常测试
Expected: ("(2, 3)") Actual: (v1) Passed
Expected: ("(4, 5, 6)") Actual: (v2) Passed

Expecting Exception: invalid_argument
Running v1.resize(-1)
Dimension must be non-negative
Exception Matched

Expecting Exception: invalid_argument
Running v1 + v2
Dimension mismatch
Exception Matched

Expecting Exception: invalid_argument
Running v1 - v2
Dimension mismatch
Exception Matched

Expecting Exception: invalid_argument
Running v1 * v2
Dimension mismatch
Exception Matched

Expecting Exception: invalid_argument
Running v1.cross(v2)
Dimension mismatch
Exception Matched

Expecting Exception: out_of_range
Running v1[3]
Index out of range
Exception Matched
```

4 亮点总结

这个项目基于自定义模板类 MyVector，提供了一组字向量操作功能和运算符重载，用于更轻松地处理向量数据。

1.模板类的使用：为了同时适配整型与浮点型，使用了泛型。

2. 宏定义的使用：让程序对异常作出反应，一般使用 lambda 表达式。我们则选择了另一种同样的可行的方法，即宏定义。

3. 结果打印颜色设置：通过宏定义，使得命令行显示结果时能呈现出不同颜色，使测试结果更明了易读。比如“passed”显示为绿色，”failed“显示为红色。

4.重载：我们对包括构造函数的许多函数在内进行了多个重载，以应对不同的输入情况，对程序的泛用性大有裨益。

致谢

在完成本论文的过程中，我们要衷心感谢指导老师对我们的悉心指导和支持。此外，我们还要感谢小组中每一位成员的通力合作和辛勤付出。大家共同努力，相互配合，在项目设计、项目实现、程序整合与测试以及论文撰写的各个环节上都做出了重要贡献。没有大家的共同努力和合作精神，这项研究工作将无法顺利完成。

衷心感谢所有对我们研究工作做出贡献的人们！

。

附录 完整代码

MyVector.h

```
#ifndef MYVECTOR_MYVECTOR_H
#define MYVECTOR_MYVECTOR_H

#include <iostream>

template<class T>
class MyVector {
public:

    typedef T value_type;

    typedef int size_type;

    MyVector();

    MyVector(size_type dim, value_type *&&data);

    MyVector(size_type dim, const value_type *data);

    MyVector(const MyVector &v);

    MyVector(MyVector &&v) noexcept;

    ~MyVector();

    void resize(size_type new_dim);

    std::ostream &operator<<(std::ostream &os) const;

    friend std::ostream &operator<<(std::ostream &os, const MyVector<T> &v) {
        return v.operator<<(os);
    }

    std::istream &operator>>(std::istream &is);

    friend std::istream &operator>>(std::istream &is, MyVector<T> &v) {
        return v.operator>>(is);
```

```
    }

    bool operator==(const MyVector &v) const;

    bool operator!=(const MyVector &v) const;

    MyVector &operator=(const MyVector &v);

    MyVector &operator=(MyVector &&v) noexcept;

    MyVector operator+(const MyVector &v) const;

    MyVector &operator+=(const MyVector &v);

    MyVector operator-(const MyVector &v) const;

    MyVector operator-() const;

    MyVector &operator-=(const MyVector &v);

    value_type operator*(const MyVector &v) const;

    MyVector operator*(value_type k) const;

    MyVector &operator*=(value_type k);

    MyVector cross(const MyVector &v) const;

    value_type &operator[](size_type i);

    const value_type &operator[](size_type i) const;

    size_type getDim() const;

private:
    void checkDim(const MyVector &v) const noexcept(false);

    value_type *data;
    size_type dim;
};

template<class T>
```

```cpp
MyVector<T>::MyVector() {
    this->dim = 0;
    this->data = new value_type[0]{};
}

template<class T>
MyVector<T>::MyVector(size_type dim, value_type *&&data) {
    if (dim < 0) {
        throw std::invalid_argument("Dimension must be non-negative");
    }
    this->dim = dim;
    this->data = data;
}

template<class T>
MyVector<T>::MyVector(size_type dim, const value_type *data) {
    if (dim < 0) {
        throw std::invalid_argument("Dimension must be non-negative");
    }
    this->dim = dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = data[i];
    }
}

template<class T>
MyVector<T>::MyVector(const MyVector &v) {
    this->dim = v.dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = v.data[i];
    }
}

template<class T>
MyVector<T>::MyVector(MyVector &&v) noexcept {
    this->dim = v.dim;
    this->data = v.data;
    v.data = nullptr;
    v.dim = 0;
}
```

```cpp
template<class T>
MyVector<T>::~MyVector() {
    delete[] data;
}

template<class T>
void MyVector<T>::resize(size_type new_dim) {
    if (new_dim < 0) {
        throw std::invalid_argument("Dimension must be non-negative");
    }
    value_type *new_data = new value_type[new_dim];
    for (size_type i = 0; i < new_dim; ++i) {
        new_data[i] = i < dim ? data[i] : 0;
    }
    delete[] data;
    this->data = new_data;
    this->dim = new_dim;
}

template<class T>
std::ostream &MyVector<T>::operator<<(std::ostream &os) const {
    os << "(";
    for (size_type i = 0; i < dim; ++i) {
        os << data[i];
        if (i != dim - 1) {
            os << ", ";
        }
    }
    os << ")";
    return os;
}

template<class T>
std::istream &MyVector<T>::operator>>(std::istream &is) {
    for (size_type i = 0; i < dim; ++i) {
        is >> data[i];
    }
    return is;
}

template<class T>
```

```cpp
bool MyVector<T>::operator==(const MyVector &v) const {
    if (dim != v.dim) {
        return false;
    }
    for (size_type i = 0; i < dim; ++i) {
        if (data[i] != v.data[i]) {
            return false;
        }
    }
    return true;
}

template<class T>
bool MyVector<T>::operator!=(const MyVector &v) const {
    return !operator==(v);
}

template<class T>
MyVector<T> &MyVector<T>::operator=(const MyVector &v) {
    if (this == &v) {
        return *this;
    }
    delete[] data;
    this->dim = v.dim;
    this->data = new value_type[dim];
    for (size_type i = 0; i < dim; ++i) {
        this->data[i] = v.data[i];
    }
    return *this;
}

template<class T>
MyVector<T> &MyVector<T>::operator=(MyVector<T> &&v) noexcept {
    if (this == &v) {
        return *this;
    }
    this->dim = v.dim;
    this->data = v.data;
    v.dim = 0;
    v.data = nullptr;
    return *this;
}
```

```
template<class T>
MyVector<T> MyVector<T>::operator+(const MyVector &v) const {
    checkDim(v);
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] += v.data[i];
    }
    return result;
}

template<class T>
MyVector<T> &MyVector<T>::operator+=(const MyVector &v) {
    checkDim(v);
    for (size_type i = 0; i < dim; ++i) {
        data[i] += v.data[i];
    }
    return *this;
}

template<class T>
MyVector<T> MyVector<T>::operator-(const MyVector &v) const {
    checkDim(v);
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] -= v.data[i];
    }
    return result;
}

template<class T>
MyVector<T> MyVector<T>::operator-() const {
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] = -result.data[i];
    }
    return result;
}

template<class T>
MyVector<T> &MyVector<T>::operator-=(const MyVector &v) {
    checkDim(v);
```

```
    for (size_type i = 0; i < dim; ++i) {
        data[i] -= v.data[i];
    }
    return *this;
}

template<class T>
typename MyVector<T>::value_type MyVector<T>::operator*(const MyVector &v) const {
    checkDim(v);
    value_type result = 0;
    for (size_type i = 0; i < dim; ++i) {
        result += data[i] * v.data[i];
    }
    return result;
}

template<class T>
MyVector<T> MyVector<T>::operator*(value_type k) const {
    MyVector result(*this);
    for (size_type i = 0; i < dim; ++i) {
        result.data[i] *= k;
    }
    return result;
}

template<class T>
MyVector<T> &MyVector<T>::operator*=(value_type k) {
    for (size_type i = 0; i < dim; ++i) {
        data[i] *= k;
    }
    return *this;
}

template<class T>
MyVector<T> MyVector<T>::cross(const MyVector &v) const {
    if (dim != 3 || v.dim != 3) {
        throw std::invalid_argument("Dimension mismatch");
    }
    MyVector result(3, new value_type[3]{data[1] * v.data[2] - data[2] * v.data[1],
                                         data[2] * v.data[0] - data[0] * v.data[2],
                                         data[0] * v.data[1] - data[1] *
v.data[0]});
```

```
    return result;
}

template<class T>
typename MyVector<T>::value_type &MyVector<T>::operator[](size_type i) {
    if (i < 0 || i >= dim) {
        throw std::out_of_range("Index out of range");
    }
    return data[i];
}

template<class T>
const typename MyVector<T>::value_type &MyVector<T>::operator[](size_type i) const {
    if (i < 0 || i >= dim) {
        throw std::out_of_range("Index out of range");
    }
    return data[i];
}

template<class T>
typename MyVector<T>::size_type MyVector<T>::getDim() const {
    return dim;
}

template<class T>
void MyVector<T>::checkDim(const MyVector &v) const noexcept(false) {
    if (dim != v.dim) {
        throw std::invalid_argument("Dimension mismatch");
    }

}


#endif //MYVECTOR_MYVECTOR_H
```

my_assert.h

```
#ifndef MYSTRING_MY_ASSERT_H
#define MYSTRING_MY_ASSERT_H

#include <iostream>
```

```
#ifdef _WIN32
#include "windows.h"
#define print_red(str) "";SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_RED); std::cout << str;
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED |
FOREGROUND_GREEN | FOREGROUND_BLUE);std::cout<<
#define print_green(str) "";SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_GREEN); std::cout << str;
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED |
FOREGROUND_GREEN | FOREGROUND_BLUE);std::cout<<
#define print_yellow(str)
"";SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED |
FOREGROUND_GREEN); std::cout << str;
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED |
FOREGROUND_GREEN | FOREGROUND_BLUE);std::cout<<
#else
#define print_red(str) "\033[31m" << str << "\033[0m"
#define print_green(str) "\033[32m" << str << "\033[0m"
#define print_yellow(str) "\033[33m" << str << "\033[0m"
#endif
#define run(x) std::cout << "Running " << print_yellow(#x) << std::endl; x;

#define my_assert_throws(x, exception)
std::cout << std::endl;
try {
    std::cout << print_green("Expecting Exception: ") << #exception <<
std::endl;    \
    run(x);
    std::cout << print_red("No Exception throws Failed") << std::endl;
} catch (exception &e) {
    std::cout << e.what() <<endl;
    std::cout << print_green("Exception Matched") << std::endl;
}

#define my_assert(actual, expected)
std::cout << print_green("Expected:") <<" (" << #expected <<") "
     << print_red("Actual:") <<" (" << print_yellow(#actual) << ")";
my_assert_impl<decltype(actual)>(actual, expected);


template<typename T>
void my_assert_impl(T actual, T expected) {
```

```
    if (actual == expected) {
        std::cout << print_green(" Passed") << std::endl;
    } else {
        std::cout << print_red(" Failed") << std::endl;
    }
}

template<typename T>
void my_assert_impl(T actual, std::string_view expected) {
    std::ostringstream oss;
    oss << actual;
    if (oss.str() == expected) {
        std::cout << print_green(" Passed") << std::endl;
    } else {
        std::cout << print_red(" Failed") << std::endl;
    }
}


#endif //MYSTRING_MY_ASSERT_H
```

MyVectorTest.cpp

```
#include <sstream>
#include "MyVector.h"
#include "my_assert.h"

using namespace std;


int main() {
    cout << "空向量测试" << endl;
    run(MyVector<int> empty_vector(0, new int[0]{}))
    my_assert(empty_vector, "()")

    run(MyVector<double> double_vector(3, new double[3]{1.1, 2.2, 3.3}))

    MyVector<int> v1(3, new int[3]{1, 2, 3});
    MyVector<int> v2(3, new int[3]{4, 5, 6});
    cout << "输出测试" << endl;
    my_assert(v1, "(1, 2, 3)")
    my_assert(v2, "(4, 5, 6)")
```

```
    cout << "加减法测试" << endl;
    my_assert(v1 + v2, "(5, 7, 9)")
    my_assert(v1 - v2, "(-3, -3, -3)")
    cout << "点乘测试" << endl;
    my_assert(v1 * v2, 32)
    cout << "数乘测试" << endl;
    my_assert(v1 * 2, "(2, 4, 6)")
    cout << "下标访问[]测试" << endl;
    my_assert(static_cast<int>(v1[1]), 2)
    cout << "下标赋值[]测试" << endl;
    run(v1[1] = 10)
    my_assert(v1, "(1, 10, 3)")
    cout << "叉乘测试" << endl;
    my_assert(v1, "(1, 10, 3)")
    my_assert(v2, "(4, 5, 6)")
    my_assert(v1.cross(v2), "(45, 6, -35)")
    my_assert(v2.cross(v1), -v1.cross(v2))
    cout << "向量拷贝测试" << endl;
    run(MyVector<int> v3(v1))
    my_assert(v3, "(1, 10, 3)")
    cout << "向量移动测试" << endl;
    run(MyVector<int> v4(std::move(v3)))
    my_assert(v4, "(1, 10, 3)")
    cout << "向量赋值测试" << endl;
    run(v3 = v4)
    my_assert(v3, "(1, 10, 3)")
    cout << "向量自赋值测试" << endl;
    run(v3 = v3)
    my_assert(v3, "(1, 10, 3)")
    cout << "向量加减赋值测试" << endl;
    run(v3 += v4)
    my_assert(v3, "(2, 20, 6)")
    run(v3 -= v4)
    my_assert(v3, "(1, 10, 3)")
    cout << "向量数乘赋值测试" << endl;
    run(v3 *= 2)
    my_assert(v3, "(2, 20, 6)")
    cout << "向量比较测试" << endl;
    my_assert(v1, "(1, 10, 3)")
    my_assert(v2, "(4, 5, 6)")
    my_assert(v1 == v2, false)
    my_assert(v1 != v2, true)
```

```
    cout << "向量缩放测试" << endl;
    my_assert(v1.getDim(), 3)
    run(v1.resize(5))
    my_assert(v1, "(1, 10, 3, 0, 0)")
    my_assert(v1.getDim(), 5)
    run(v1.resize(2))
    my_assert(v1, "(1, 10)")
    my_assert(v1.getDim(), 2)
    run(v1.resize(3))
    my_assert(v1, "(1, 10, 0)")
    my_assert(v1.getDim(), 3)
    run(v1.resize(0))
    my_assert(v1, "()")
    my_assert(v1.getDim(), 0)


    cout << "=====================================" << endl;
    cout << "输入测试" << endl;
    run(v1.resize(2))
    run(istringstream is("2 3"))
    run(is >> v1)
    my_assert(v1, "(2, 3)")
    cout << "异常测试" << endl;
    my_assert(v1, "(2, 3)")
    my_assert(v2, "(4, 5, 6)")
    my_assert_throws(v1.resize(-1), invalid_argument)
    my_assert_throws(v1 + v2, invalid_argument)
    my_assert_throws(v1 - v2, invalid_argument)
    my_assert_throws(v1 * v2, invalid_argument)
    my_assert_throws(v1.cross(v2), invalid_argument)
    my_assert_throws(v1[3], out_of_range)
}
```