Übung 04 – Klasse rational_t erweitern

Aufwand: 16h

Inhalt

Aufgabe 1 – Klasse rational_t erweitern:

In dieser Übung ging es um die Erweiterung der Klasse „rational_t“ aus der vergangenen Übung um Templates und Traits.

Lösungsidee:

Grundsätzlich bin ich vom Stand der Übung 3 ausgegangen, wo das Rechnen mit rational_t's mit den Typen „int“ möglich war.

Angefangen habe ich damit, dass ich die Klasse als Template gekennzeichnet habe und überall die statischen Datentypen (in diesem Fall int) durch den Template Datentyp ausgetauscht habe. Hierfür wurden speziell auch typedefs für den Datentyp und die nelms_traits_class angelegt.

Anschließend dazu habe ich alle statischen „Null“, „Eins“ usw. Einträge, welche durch die Traits abgedeckt werden sollen durch die entsprechenden „nelms_traits_class“-Funktionen ausgetauscht.

Nach dem Testen der Funktionalität mit int und doubles bin ich zur Implementierung der Matrix vorangeschritten. Die Matrix besteht grundsätzlich aus einem 2-dimensionalen C-Array, einem Konstruktor und den Operatoren. Beim Konstruktor ist der sogenannte Conversion Konstruktor sehr wichtig, um direkt aus doubles bzw. Integern eine Matrix zu machen. Implementiert habe ich sämtliche Rechen- und Vergleichsoperatoren zwischen zwei Matrizen, da andere Werte ja durch den Conversion-Konstruktor implizit umgewandelt werden sollten.

Anschließend bin ich zu der Implementierung der Operatoren übergegangen:

- Abs liefert den Betrag einer Zahl a
- Divides liefert zurück, ob eine Zahl a ohne Rest von Zahl b geteilt wird.
- Equals liefert zurück, ob Zahl a gleich Zahl b ist.
- GCD liefert den „greatest common divider“ (größter gemeinsamer Teiler) von Zahl a und Zahl b zurück.
- Is_negative liefert zurück, ob eine Zahl a negativ (unter null) ist.
- Is_Zero liefert zurück, ob eine Zahl null ist.
- Negate liefert die Negative Variante der Zahl a zurück.
- Remainder liefert den Rest einer Division zwischen Zahl a und Zahl b zurück.

Meine Anforderungen für diese Operationen sind lediglich, dass mit den Zahlen bzw. Datentypen gerechnet werden kann und diese verglichen werden können. Deshalb weist auch meine Matrix eben diese Operatoren auf.

Quelltext

Main.cpp

```
#include "rational_type.h"
#include "divide_by_zero_exception.h"
#include<iostream>
#include<fstream>
#include "matrix.h"

void test_inverse()
{
	std::cout << "Testcase 1.1: Inverse Test:\n";
	std::cout << "Tests if inverse() function is working.\n";
	rational_t<int> int_rat(5, 1);
	rational_t<double> dbl_rat(1.2, 2.4);
	// Matrix is being skipped due to it not working

	int_rat.inverse();
	dbl_rat.inverse();
	std::cout << "\nExpected: Int = <1/5>, Double = <2/1>\n";
	std::cout << "Actual: ";
	std::cout <<  "Int = " << int_rat << ", " <<
				"Double = " << dbl_rat << std::endl;
	std::cout << std::endl;
}

void test_int_ops()
{
	std::cout << "Testcase 2.1: Calculating with rational_t<int>:\n";
	std::cout << "Testing if calculations still work after applying the template.\n\n";
	rational_t<int> r(-1, 2);

	std::cout	<< r * -10 << std::endl
			<< r * rational_t<int>(20, -2)	<< std::endl;

	r = 7;

	std::cout	<< r + rational_t<int>(2, 3)				<< std::endl
			<< 10 / r / 2 + rational_t<int>(6, 5)	<< std::endl;

	std::cout << std::endl;
}
void test_double_ops()
{
	std::cout << "Testfall 2.2: Calculating with rational_t<double>:\n";
	std::cout << "Testing if calculations still work after applying the template.\n\n";
	rational_t<double> r(-1, 2);

	std::cout << r * -10.0f << std::endl
		<< r * rational_t<double>(20, -2) << std::endl;

	r = -3.1f;

	std::cout << r + rational_t<double>(2, 3) << std::endl
		<< 10 / r / 2 + rational_t<double>(6, 5) << std::endl;

	std::cout << std::endl;
}
```

```
void test_matrix_ops()
{
	std::cout << "Testcase 2.3: Calculating with rational_t<Matrix<int>>:\n";
	std::cout << "Testing if calculations still work after applying the template.\n\n";
	/*rational_t<Matrix<int>> r(2, 2);*/
	std::cout << "Doesn't work. More on that matter in the document.\n";
	// Arithmethic operators would function analogous to the ones above, but since the
matrix doesn't work im sparing the writing work.
	std::cout << std::endl;
}

void test_matrix()
{
	std::cout << "Testcase 4.1: Testing if creating a rational with matrices works.\n";
	std::cout << "Expected: <2,2>\n";
	// rational_t<Matrix<int>> r(2, 2);
	std::cout << "As already mentioned above I had issues creating the class.\n";
	std::cout << std::endl;
}


void main()
{
	test_inverse();

	test_int_ops();
	test_double_ops();
	test_matrix_ops();

	test_matrix();
}
```

Rational_type.h

```cpp
#ifndef RATIONAL_TYPE_H
#define RATIONAL_TYPE_H
#include<iostream>
#include "operations.h"
#include "divide_by_zero_exception.h"
#include<string>
#include "matrix.h"

template<typename T = int, typename S = ops::nelms_traits_t<T>>
class rational_t {

	typedef T value_t;
	typedef S traits_t;

	// Friend declarations - barton nackman trick (inline)
	friend std::ostream& operator<<(std::ostream& lhs, rational_t<value_t> const& rhs)
	{
		// Delegate to print
		rhs.print(lhs);
		return lhs;
	}
	friend std::istream& operator>>(std::istream& lhs, rational_t<value_t>& rhs)
	{
		// Delegate to scan
		rhs.scan(lhs);
		return lhs;
	}
	friend rational_t<value_t> operator+(T const lhs, rational_t<value_t>& rhs)
	{
		rational_t<value_t> tmp(lhs);
		return tmp + rhs;
	}
	friend rational_t<value_t> operator-(T const lhs, rational_t<value_t>& rhs)
	{
		rational_t<value_t> tmp(lhs);
		return tmp - rhs;
	}
	friend rational_t<value_t> operator*(T const lhs, rational_t<value_t>& rhs)
	{
		rational_t<value_t> tmp(lhs);
		return tmp * rhs;
	}
	friend rational_t<value_t> operator/(T const lhs, rational_t<value_t>& rhs)
	{
		rational_t<value_t> tmp(lhs);
		return tmp / rhs;
	}
	friend rational_t<value_t> operator+(rational_t<value_t>& lhs, T const rhs)
	{
		rational_t<value_t> tmp(rhs);
		return tmp + lhs;
	}
	friend rational_t<value_t> operator-(rational_t<value_t>& lhs, T const rhs)
	{
		rational_t<value_t> tmp(rhs);
		return lhs - tmp;
	}
```

```
	friend rational_t<value_t> operator*(rational_t<value_t>& lhs, T const rhs)
	{
		rational_t<value_t> tmp(rhs);
		return tmp * lhs;
	}
	friend rational_t<value_t> operator/(rational_t<value_t>& lhs, T const rhs)
	{
		rational_t<value_t> tmp(rhs);
		return lhs / tmp;
	}

private:
	value_t numerator;
	value_t denominator;


	// Transforms the current rational into it's canonical representation
	void normalize()
	{
		// Calculate common divider
		value_t gcd_value(ops::gcd(numerator, denominator));

		// If there is a common divider between numerator and denominator, divide by
it.
		if (gcd_value != traits_t::zero())
		{
			numerator /= gcd_value;
			denominator /= gcd_value;
		}

		// Also, if the denominator is negative, flip the signs.
		if (denominator < 0)
		{
			numerator *= ops::negate(traits_t::one());
			denominator *= ops::negate(traits_t::one());
		}
	}
	// Returns whether this rational is in a consistent state
	bool is_consistent() const
	{
		return !ops::is_zero(get_denominator());
	}


	// Compares two rationals.
	// Returns 0 if equal, -1 if main rational is less and +1 if main rational is
greater.
	int cmp(rational_t<value_t> const& other) const
	{
		// Calculate lowest common multiple and calculate "normalized" numerators
(if they had the same denominators)
		value_t lcm_value(ops::lcm(this->denominator, other.denominator));
		value_t num1(lcm_value / this->denominator * this->numerator);
		value_t num2(lcm_value / other.denominator * other.numerator);
```

```
// Then compare and return the result
		if (num1 > num2)
		{
			return 1;
		}
		else if (num1 < num2)
		{
			return -1;
		}
		else
		{
			return 0;
		}
	}

	// Adds the value of another rational
	void add(rational_t<value_t> const& other)
	{
		// Calc lowest common multiple between the two values
		value_t new_denominator(ops::lcm(get_denominator(),
other.get_denominator()));

		// Then convert the value to have the final denominator
		this->numerator *= new_denominator / get_denominator();
		this->denominator = new_denominator;

		// Then add the multiplied numerator of the other half.
		numerator += other.get_numerator() * (new_denominator /
other.get_denominator());

		// At the end, normalize it to get canonical form.
		normalize();
	}
	// Subtracts the value of another rational
	void sub(rational_t<value_t> const& other)
	{
		// Just like in add(), first convert both numbers to have the same
denominator, then substract them.
		value_t new_denominator(lcm(get_denominator(), other.get_denominator()));
		this->numerator *= new_denominator / this->denominator;
		this->denominator = new_denominator;

		numerator -= other.numerator * (new_denominator / other.denominator);
		normalize();
	}
	// Multiplies by the value of another rational
	void mul(rational_t<value_t> const& other)
	{
		// When multiplying to rationals, no previous conversion is needed.
		numerator *= other.get_numerator();
		denominator *= other.get_denominator();

		normalize();
	}
```

```
// Divides by the value of another rational
	void div(rational_t<value_t> const& other)
	{
		// If denominator is 0, abort and throw an exception.
		if (!other.is_consistent())
		{
			throw divide_by_zero_exception();
		}

		rational_t tmp(other.denominator, other.numerator);
		mul(tmp);

		normalize();
	}
	// Prints the current rational to the designated outputstream
	std::ostream& print(std::ostream& out = std::cout) const
	{
		out << as_string();
		return out;
	}
	// Reads in a rational from the given inputstream
	std::istream& scan(std::istream& in)
	{
		// read in two values from the designated stream
		// If reading in fails, set it to 1 or 0 respectively
		if (!(in >> this->numerator))
			this->numerator = traits_t::zero;
		if (!(in >> this->denominator))
			this->denominator = traits_t::one;

		return in;
	}

public:
	// Creates a rational using default values for "one".
	rational_t()
	{
		numerator = traits_t::one();
		denominator = traits_t::one();
	}
	// Creates a rational <numerator/one>
	rational_t(T const& _numerator)
	{
		numerator = _numerator;
		denominator = traits_t::one();
	}
	// Creates a rational <numerator, denominator>
	rational_t(T const& _numerator, T const& _denominator)
	{
		numerator = _numerator;
		denominator = _denominator;

		// If not consistent (0 in denominator), display and throw an
error/Exception.
		if (!is_consistent())
		{
			throw divide_by_zero_exception();
		}
```

```
			normalize();
		}
		// Creates a rational using another rational
		rational_t(rational_t<value_t> const& src)
		{
			numerator = src.numerator;
			denominator = src.denominator;
		}


		// Assignment operator override.
		rational_t<value_t> operator=(rational_t<value_t> const& src)
		{
			// On assignment, check if the object is the same.
			if (*this == src)
			{
				return *this;
			}
			else
			{
				// If it is a different object assign the values directly instead of
claiming new memory.
				this->numerator = src.numerator;
				this->denominator = src.denominator;
				return *this;
			}
		}

		// Add operator override.
		rational_t<value_t> operator+(rational_t<value_t> const& other)
		{
			rational_t tmp(*this);
			tmp.add(other);
			return tmp;
		}
		// Subtract operator override.
		rational_t<value_t> operator-(rational_t<value_t> const& other)
		{
			rational_t tmp(*this);
			tmp.sub(other);
			return tmp;
		}
		// Multiply operator override.
		rational_t<value_t> operator*(rational_t<value_t> const& other)
		{
			rational_t tmp(*this);
			tmp.mul(other);
			return tmp;
		}
		// Divide operator override.
		rational_t<value_t> operator/(rational_t<value_t> const& other)
		{
			rational_t tmp(*this);
			tmp.div(other);
			return tmp;
		}
```

```cpp
// Add assignment operator override.
	rational_t<value_t> operator+=(rational_t<value_t> const& other)
	{
		add(other);
		return *this;
	}
	// Subtract assignment operator override.
	rational_t<value_t> operator-=(rational_t<value_t> const& other)
	{
		sub(other);
		return *this;
	}
	// Multiply assignment operator override.
	rational_t<value_t> operator*=(rational_t<value_t> const& other)
	{
		mul(other);
		return *this;
	}
	// Divide assignment operator override.
	rational_t<value_t> operator/=(rational_t<value_t> const& other)
	{
		div(other);
		return *this;
	}

	// Equal operator override.
	bool operator==(rational_t<value_t> const& other) const
	{
		return cmp(other) == 0;
	}
	// Unequal operator override
	bool operator!=(rational_t<value_t> const& other) const
	{
		return cmp(other) != 0;
	}
	// Greater than operator override
	bool operator>(rational_t<value_t> const& other) const
	{
		return cmp(other) == 1;
	}
	// Less than operator override
	bool operator<(rational_t<value_t> const& other) const
	{
		return cmp(other) == -1;
	}
	// Greater or equal than operator override
	bool operator>=(rational_t<value_t> const& other) const
	{
		return cmp(other) != -1;
	}
	// Less or equal than operator override
	bool operator<=(rational_t<value_t> const& other) const
	{
		return cmp(other) != 1;
	}
```

```
// Returns the numerator of the current rational
	value_t get_numerator() const
	{
		return numerator;
	}

	// Returns the denominator of the current rational
	value_t get_denominator() const
	{
		return denominator;
	}

	// Returns the current rational as string as <numerator/denominator>
	std::string as_string() const
	{
		return "<" + ops::to_string(get_numerator()) + "/" +
ops::to_string(get_denominator()) + ">";
	}



	// Returns whether the rational is positive or not
	bool is_positive()
	{
		return !(ops::is_negative(get_numerator()) ||
ops::is_zero(get_denominator()));
	}
	// Returns whether the rational is negative or not
	bool is_negative()
	{
		return ops::is_negative(get_numerator());
	}
	// Returns whether the rational is zero or not
	bool is_zero()
	{
		return ops::is_zero(get_numerator());
	}

	// Swaps numerator and denominator.
	void inverse()
	{
		value_t tmp = numerator;
		numerator = denominator;
		denominator = tmp;
	}
};

#endif
```

Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H
#include<vector>

template<typename T>
class Matrix {
	typedef T value_t;

private:
	value_t** elements;

	// Adds the value of another rational
	void add(Matrix<value_t> const& other)
	{
		elements[0][0] += other.elements[0][0];
	}



// Subtracts the value of another rational
	void sub(Matrix<value_t> const& other)
	{
		elements[0][0] -= other.elements[0][0];
	}
	// Multiplies by the value of another rational
	void mul(Matrix<value_t> const& other)
	{
		elements[0][0] *= other.elements[0][0];
	}
	// Divides by the value of another rational
	void div(Matrix<value_t> const& other)
	{
		elements[0][0] /= other.elements[0][0];
	}

public:

	Matrix()
	{
		elements = new value_t*[1];
		elements[0] = new value_t[1];
		elements[0][0] = 1;
	}
	Matrix(T const& val)
	{
		elements = new value_t*[1];
		elements[0] = new value_t[1];
		elements[0][0] = val;
	}
	~Matrix()
	{
		delete[] elements[0];
		delete[] elements;
	}
```

```
T get_element() const
{
	return elements[0][0];
}

Matrix<value_t> operator=(Matrix<value_t> const& other)
{
	elements[0][0] = other.elements[0][0];
	return *this;
}
Matrix<value_t> operator+(Matrix<value_t> const& other)
{
	Matrix<value_t> tmp(*this);
	tmp.add(other);
	return tmp;
}
// Subtract operator override.
Matrix<value_t> operator-(Matrix<value_t> const& other)
{
	Matrix<value_t> tmp(*this);
	tmp.sub(other);
	return tmp;
}



// Multiply operator override.
Matrix<value_t> operator*(Matrix<value_t> const& other)
{
	Matrix<value_t> tmp(*this);
	tmp.mul(other);
	return tmp;
}
// Divide operator override.
Matrix<value_t> operator/(Matrix<value_t> const& other)
{
	Matrix<value_t> tmp(*this);
	tmp.div(other);
	return tmp;
}
// Add assignment operator override.
Matrix<value_t> operator+=(Matrix<value_t> const& other)
{
	add(other);
	return *this;
}
// Subtract assignment operator override.
Matrix<value_t> operator-=(Matrix<value_t> const& other)
{
	sub(other);
	return *this;
}
// Multiply assignment operator override.
Matrix<value_t> operator*=(Matrix<value_t> const& other)
{
	mul(other);
	return *this;
}
```

```
	// Divide assignment operator override.
	Matrix<value_t> operator/=(Matrix<value_t> const& other)
	{
		div(other);
		return *this;
	}

	// Equal operator override.
	bool operator==(Matrix<value_t> const& other) const
	{
		return elements[0][0] == other.elements[0][0];
	}

	// Unequal operator override
	bool operator!=(Matrix<value_t> const& other) const
	{
		return elements[0][0] != other.elements[0][0];
	}
	// Greater than operator override
	bool operator>(Matrix<value_t> const& other) const
	{
		return elements[0][0] > other.elements[0][0];
	}
	// Less than operator override
	bool operator<(Matrix<value_t> const& other) const
	{
		return elements[0][0] < other.elements[0][0];
	}


	// Greater or equal than operator override
	bool operator>=(Matrix<value_t> const& other) const
	{
		return elements[0][0] >= other.elements[0][0];
	}
	// Less or equal than operator override
	bool operator<=(Matrix<value_t> const& other) const
	{
		return elements[0][0] <= other.elements[0][0];
	}
};

#endif
```

Divide_by_zero_exception.h

```
#ifndef DIVIDE_BY_ZERO_EXCEPTION_H
#define DIVIDE_BY_ZERO_EXCEPTION_H
#include<exception>

class divide_by_zero_exception : public std::exception
{
public:
	virtual const char* what() const throw()
	{
		return "Divide by 0 exception";
	}
};

#endif
```

Testfälle:

- Inverser Operator
- Rechnen mit Rational_t
- Funktionalität der Matrix

Inverse Funktion:

```
Testcase 1.1: Inverse Test:
Tests if inverse() function is working.

Expected: Int = <1/5>, Double = <2/1>
Actual: Int = <1/5>, Double = <2/1>
```

Der Testfall hat funktioniert!

Rechnen mit Rational_t:

```
Testcase 2.1: Calculating with rational_t<int>:
Testing if calculations still work after applying the template.

<5/1>
<5/1>
<23/3>
<67/35>

Testfall 2.2: Calculating with rational_t<double>:
Testing if calculations still work after applying the template.

<5/1>
<5/1>
<-15309209/6291456>
<-13421774/32505855>

Testcase 2.3: Calculating with rational_t<Matrix<int>>:
Testing if calculations still work after applying the template.

Doesn't work. More on that matter in the document.
```

Bis auf die Matrix hat der Testfall wieder funktioniert. Ich bin die Erstellung der Matrix Schrittweise mit dem Debugger durchlaufen: Die Matrix wird problemlos erstellt, jedoch wird nach dem Zuweisen von Zähler und Nenner der Destruktor aufgerufen und ich verstehe leider nicht warum. Ich habe probiert mit Referenzen etc. zu Arbeiten bzw. nur Werte zu kopieren, aber leider ergebnislos.

Funktionalität der Matrix:

```
Testcase 4.1: Testing if creating a rational with matrices works.
Expected: <2,2>
As already mentioned above I had issues creating the class.
```

Beim Durchlaufen des Vorgangs kann gesehen werden, dass die Matrix richtig angelegt wird, es aber dann wie oben erwähnt zum Fehler kommt, weil nicht auf Konsistenz überprüft werden kann, weil Zähler/Nenner direkt nach Zuweisung gelöscht werden.