1 Objectives

1. Experience creating an abstract data type (ADT)[1]
2. Implement an ADT in C++, using the operator overloading facility of the C++ language
3. Learn about function objects and how to define them

2 Assignment Background

A *data type* represents a set of data values sharing common properties. An abstract data type (ADT) specifies a set of operations on a *data type*, independent of how the data values are actually modeled or how the operations are implemented.

Classic ADTs such as rational number and complex number ADTs support many arithmetic, relational and other operations, making them ideal data types for operator overloading.

However, a search for "class rational c++" reveals many turnkey C++ classes, forcing assignments designed to provide practice with operator overloading to get a bit creative with their choice of *data types*; ideally, a *data type* that is not as ubiquitous as rational and complex number ADTs but lends itself to operator overloading just as good.

3 Introducing ADT Quad

Quad is an abstract data type, representing a set of values of the form

$$\frac{x_1}{x_2} + \frac{x_3}{x_4}\sqrt{2}, \text{ with } x_1, x_2, x_3, x_4 \text{ all integers and } x_2 \neq 0, x_4 \neq 0 \tag{1}$$

and providing a typical set of operations on those values. In this assignment, we will refer to such values as *quad* numbers. The integers x_1 and x_2 represent the *real part*, and the integers x_3 and x_4 represent the *quad part*, of a quad number.

3.1 Notation

To facilitate presentation of operations on quad numbers, we abstract quad numbers into a sequence of four *ordered* integers x_1, x_2, x_3, and x_4, corresponding to the same integers in (1). We use the notations $[x_1, x_2, x_3, x_4]$ and $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$ to represent the same quad number, whichever is more convenient for expressing an operation.

[1] ADT $\equiv$ Values + Operations − Implementation details of both

3.2 Special Quads

Quad x	$\left[x_1, x_2, x_3, x_4\right]$, $x_2 \neq 0$ and $x_4 \neq 0$
x Normalized	$\begin{cases} [-x_1, -x_2, x_3, x_4] & \text{if } x_2 < 0 \\ [x_1, x_2, -x_3, -x_4] & \text{if } x_4 < 0 \\ [-x_1, -x_2, -x_3, -x_4] & \text{if } x_2 < 0 \text{ and } x_4 < 0 \end{cases}$
x Reduced	$\left[\frac{x_1}{\gcd(x_1,x_2)}, \frac{x_2}{\gcd(x_1,x_2)}, \frac{x_3}{\gcd(x_3,x_4)}, \frac{x_4}{\gcd(x_3,x_4)}\right]$
x Standardized	Both Reduced and Normalized
x Conjugated	$\left[x_1, x_2, -x_3, x_4\right]$, $x_2 \neq 0$ and $x_4 \neq 0$
Zero	$\left[0, z_2, 0, z_4\right]$, $z_2 \neq 0$ and $z_4 \neq 0$
Standard zero	$\left[0, 1, 0, 1\right]$
Identity	$\left[1, 1, 0, i_4\right]$, $i_4 \neq 0$
Standard identity	$\left[1, 1, 0, 1\right]$
An integer k as a Quad	the quad number $\left[k, 1, 0, 1\right]$

where $\gcd(a,b)$, called the greatest common divisor of the nonzero integers a and b, represents a positive integer d such that (1) d is a divisor of both a and b, and (2) any divisor of both x and y is also a divisor of d. For example, $\gcd(63,14) = 7$, and $\gcd(30,45) = 15$; thus, the reduced form of the quad number $\left[63, 14, 30, 45\right]$ is $\left[9, 2, 2, 3\right]$.

3.3 Operations on Quads

The basic operations on quad values are listed below, using as operands the quad numbers $x = \left[x_1, x_2, x_3, x_4\right]$ and $y = \left[y_1, y_2, y_3, y_4\right]$.

Operation	Definition
Negation	$-\left[x_1, x_2, x_3, x_4\right] = \left[-x_1, x_2, -x_3, x_4\right]$
Addition **Subtraction**	$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \pm \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 y_2 \pm x_2 y_1 \\ x_2 y_2 \\ x_3 y_4 \pm x_4 y_3 \\ x_4 y_4 \end{bmatrix}$

Multiplication $$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} * \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 y_1 x_4 y_4 + 2 x_3 y_3 x_2 y_2 \\ x_2 y_2 x_4 y_4 \\ x_1 y_3 y_2 x_4 + y_1 x_3 x_2 y_4 \\ x_2 y_2 x_4 y_4 \end{bmatrix}$$

Scalar Multiplication $$k * x = \begin{bmatrix} kx_1 \\ x_2 \\ kx_3 \\ x_4 \end{bmatrix} = x * k$$

Scalar Addition
Scalar Subtraction $$x \pm k = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \pm \begin{bmatrix} k \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \pm kx_2 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$ and $k \pm x = \pm(x \pm k)$

Inversion $$x^{-1} = \begin{pmatrix} x_1 x_3 x_4^2 \\ \alpha \\ -x_2 x_3^2 x_4 \\ \alpha \end{pmatrix}$$ provided that $\alpha = x_1^2 x_4^2 - 2 x_2^2 x_3^2 \neq 0$

Division $x/y = x * y^{-1}$

Scalar Division $$x/k = x * \left(\frac{1}{k}\right) = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} * \begin{bmatrix} 1 \\ k \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ kx_2 \\ x_3 \\ kx_4 \end{bmatrix} \qquad k \neq 0, x_2 \neq 0, x_4 \neq 0$$

Scalar Division $k/x = k * x^{-1}$

Equal? $x = y$ if $x_1 = y_1$, $x_2 = y_2$, $x_3 = y_3$, and $x_4 = y_4$

Similar? x is similar to y if $x_1 y_2 = y_1 x_2$ and $x_3 y_4 = y_3 x_4$

Norm of x $\left(\frac{x_1}{x_2}\right)^2 + \left(\frac{x_3}{x_4}\right)^2$, a floating-point number measuring x

Absolute value of x $\sqrt{\text{Norm of } x}$, a floating-point number

Less than? $x < y$ is true if Norm of $x <$ Norm of y; false, otherwise.

4 Your Task

Implement the **Quad** ADT described above. Your **Quad** class should have the following members:

1. A container of type **std::array<long long int, 4>** to store a quad number
2. A constructor taking four parameters of type **long long int**, all four with default values. The zero quad number $[0, 1, 0, 1]$ provides the default values.
3. Defaulted copy constructor, defaulted assignment operator, defaulted virtual destructor
4. Compound assignment operators. Typically, all are implemented as member functions.

quad **op** quad	**x += y**, **x −= y**, **x *= y**, **x /= y**,
quad **op** integer	**x += k**, **x −= k**, **x *= k**, **x /= k**

5. Basic arithmetic operators. Not all can be implemented as members. None modifies it operands. For consistency, all are typically implemented as free functions.

quad **op** quad	**x + y** , **x − y**, **x * y**, **x / y**,
quad **op** integer	**x + k** , **x − k**, **x * k**, **x / k**,
integer **op** quad	**k + x**, **k − x**, **k * x**, **k / x**

6. Relational operators. Not all can be implemented as members. None modifies it operands. For consistency, all are typically implemented as free functions.

quad **op** quad	**x < y**, **x <= y**, **x > y**, **x >= y**, **x == y**, **x != y**
quad **op** integer	**x < k**, **x <= k**, **x > k**, **x >= k**, **x == k**, **x != k**
integer **op** quad	**k < x**, **k <= x**, **k > x**, **k >= x**, **k == x**, **k != x**

7. An overloaded XOR **operator^** such that **x^k** returns the quad resulting from raising **x** to the power **k** (an integer). Does not modify **x**.
8. Unary operators **++x**, **x++**, **−−x**, **x−−**, **+x**, and **–x**. All implemented as members.
9. Subscript operators **[ ]**, both const and non-const versions. If subscript is invalid, must throw: **invalid_argument("index out of bounds")**
10. Function call operator **( )** overload that takes no arguments and returns the absolute value, a **double**, of the quad number. This will effectively turn **Quad** objects like **x** into functions–hence the name "function objects."
11. Overloaded input operator for reading **Quad** values
12. Overloaded output operator for writing **Quad** values
13. **isSimilar()**, **inverse()**, **absoluteValue()**, **norm()**, **normalize()**, **reduce()**, **conjugate()**, **standardize()**. Since these members are not as common and well known as arithmetic and relational operations, we choose to implement them as named member functions, using meaningful names that reflect their functionality, as opposed to implement them using an unintuitive and weird looking name that begins with the word "**operator**" and ends with a C++ operator symbol.

(a) **x.isSimilar(y)** returns true if **x** is similar to **y**; false, otherwise

(b) **x.isSimilar(k)** returns true if **x** is similar to **Quad(k)**; false, otherwise

(c) **x.inverse()** returns the inverse of **x**

(d) **x.absoluteValue()** returns the absolute value of **x**, a **double** value

(e) **x.norm()** returns the norm of **x**, a **double** value

(f) **x.normalize()** normalizes **x**; returns void

(g) **x.reduce()** reduces **x**; returns void

(h) **x.standardize()** both reduces and normalizes **x**; returns void

(i) **x.conjugate()** returns the conjugate of **x**

5 Requirements

1. Quad numbers such as $x = \left[x_1, x_2, x_3, x_4\right]$ must be normalized so that x_2 and x_4 are positive.
2. Quad operations can produce a quad number whose integer components are huge in magnitude, resulting in integer overflow. Therefore, to reduce possibility of integer overflow, operations that create or modify quad numbers *must reduce* the resulting quad numbers.

6 Basic guidelines

Use the following guidelines[2] to choose to either implement operators as a member function or a non-member function:

Operator	Recommended Implementation
$=$, ( ), [ ], $->$	must be member
All unary operators	member
Compound assignment operators	member
All other binary operators	non-member

7 Deliverables

1. Header files: **Quad.h**
2. Implementation files: **Quad.cpp**, **quad_test_driver.cpp**
3. A **README.txt** text file (as described in the course outline).

[2]Rob Murray, C++ Strategies & Tactics, Addison-Wesley, 1993, page 47.

8 Sample Test Driver

```
#include <iostream>
#include <iomanip>
#include <string>
#include <limits>
#include <cassert>
#include "Quad.h"
using std::cout;
using std::endl;

/*
Tests class Quad. Specifically, tests constructors, compound assignment
operator overloads, basic arithmetic operator overloads, unary +, unary -,
pre/post-increment/decrement, subscripts, function objects,
input/output operators, isSimilar, absValue, and equality relational operators.

@return 0 to indicate success.
*/
void print(const std::string item, const Quad& q)
{
   cout << std::left << std::setw(20) << item << q << endl;
}
int main()
{
   //cout << "sizeof(long long int) = " <<  sizeof(long long int) << '\n';
   //cout << "Minimum value for long long int: "
         //<< std::numeric_limits<long long int>::min() << '\n';
   //cout << "Maximum value for long long int: "
        // << std::numeric_limits<long long int>::max() << '\n';
   //Minimum value for 8-byte long long int: -9223372036854775808
   //Maximum value for 8-byte long long int : 9223372036854775807

   Quad zero; // defaluts to the zero quad [0,1,0,1]
   print("Quad zero", zero);
   assert(zero == Quad(0,1,0,1));

   Quad one(1);  //  [1,1,0,1]
   print("Quad identity(1)", one);
   assert(one == Quad(1, 1, 0, 1));

   Quad half(1,2);  //  [1,2,0,1]
   print("Quad half(1,2)", half);
   assert(half == Quad(1, 2, 0, 1));

   Quad q0;
   print("Quad q0", q0);
   assert(q0 == zero);
```

```

    Quad q1(2); // defaluts to the zero quad [2,1,0,1]
    print("Quad q1(2)", q1);
    assert(q1 == Quad(2, 1, 0, 1));
    Quad q2(2,3); // defaluts to [2,3,0,1]
    print("Quad q2(2,3)", q2);
    assert(q2 == Quad(2, 3, 0, 1));

    Quad q3(2, 3, 4); // defaluts to [2,3,4,1]
    print("Quad q3(2,3,4)", q3);
    assert(q3 == Quad(2, 3, 4, 1));

    Quad q4(2, 3, 4, 5); //  [2,3,4,5]
    print("Quad q4(2, 3, 4, 5)", q4);
    assert(q4 == Quad(2, 3, 4, 5));

    assert(q0 + one == one);
    assert(q0 * one == zero);
    assert(one * one == one);
    assert(one + one == Quad(2));
    assert(Quad(2) - one == one);
    assert(Quad(1, 2).isSimilar(Quad(10, 20)));
    assert(Quad(1, 2) == Quad(10, 20)); // Quad's ctor always standardizes the constru
    assert(Quad(1, 2) == Quad(1, 2));

    Quad q11(100, -100, -1000, -1000);
    print("q11", q11);
    assert(q11 == Quad(-1, 1, 1, 1));
    q11[1] = 1;
    assert(q11 == Quad(1, 1, 1, 1));
    Quad q23(10, 5, 6, 2);
    assert(q23 == Quad(2,1,3,1));
    Quad q22{ q23 - Quad(0,10,10,10) };
    assert(q22 == Quad(2, 1, 2, 1));
    assert(q22 - q11 == q23 - q22 + one);

    // additions and subtractions
    Quad q01234 = q0 + q1 + q2 + q3 + q4;
    print("q01234", q01234);
    Quad qs01234 = -q0 - q1 - q2 - q3 - q4;
    print("qs01234", qs01234);
    assert(q01234 == -qs01234);
    Quad m01234 = 5LL * q0 + 4LL*q1 + 3LL*q2 + 2LL * q3 + 1LL * q4;
    print("m01234", m01234);
    Quad n01234 = q0 * 5LL + q1 * 4LL + q2 * 3LL +  q3 * 2LL + q4 * 1LL;
    print("n01234", n01234);
    assert(m01234 == n01234);

    // mutiplications, division
    Quad qm1234 = q1 * q2 * q3 * q4;
    print("qm1234", qm1234);
    Quad qmr1234 = q4 * q3 * q2 * q1;
    print("qmr1234", qmr1234);
    assert(qm1234 / q4 / q1 == (q3/3LL + q3 / 3LL + q3 / 3LL) * (q2/ 2LL + q2 / 2LL));
```

```
    cout << setw(20) << "q4.norm() = " << q4.norm() << endl;
    double  size_of_q4 = q4; // quad to double ( not double to quad! )
    cout << setw(20) << "size of q4 = " <<  size_of_q4 << endl;
    cout << setw(20) << "conjugate of q4 = " <<  q4.conjugate() << endl;
    Quad q5{ qmr1234 }; //
    print("Quad q5{ qmr1234 }", q5);

    // inverse
    Quad q5_inverse{ q5.inverse() };
    print("Quad q5_inverse", q5_inverse);
    assert(q5_inverse * q5 == one);
    assert(q5 == one / q5_inverse);
    assert(q5_inverse == one / q5);

    // operator []
    Quad q6{};
    q6[1] = 10;
    q6[2] = -20;
    q6[3] = 0;
    q6[4] = 40;
    print("q6", q6);
    q6.normalize();
    print("q6 normalized", q6);
    q6.reduce();
    print("q6 reduced", q6);

    //operator ++, --, both versions
    ++q6;
    print("++q6", q6);
    q6 += half;
    print("q6 += half", q6);
    assert(q6 == one);
    q6++;
    print("q6++", q6);
    assert(q6 == one + one);
    q6--;
    print("q6--", q6);
    assert(q6 == one);
    --q6;
    print("--q6", q6);
    assert(q6 == zero);

    // operator ^ to raise a quad to a positive integer power
    Quad q7{ half };
    print("q7", q7);
    q7 = half ^ 1;
    print("q7 = half ^ 1", q7);
    q7 = half ^ 2;
    print("q7 = half ^ 2", q7);
    q7 = half ^ 3;
    print("q7 = half ^ 3", q7);
```

```
    q7 = half ^ 4;
    print("q7 = half ^ 4", q7);
    q7 = half ^ 5;
    print("q7 = half ^ 5", q7);

    // operator >>
    Quad input_quad{};
    cin >> input_quad;
    print("input_quad", input_quad);
    // operator ^ to raise a quad to a positive integer power
    Quad q8{};
    print("q8", q8);
    q8 = input_quad ^ 1;
    print("q8 = input_quad ^ 1", q8);
    q8 = input_quad ^ 2;
    print("q8 = input_quad ^ 2", q8);
    q8 = input_quad ^ 3;
    print("q8 = input_quad ^ 3", q8);
    q8 = input_quad ^ 4;
    print("q8 = input_quad ^ 4", q8);
    q8 = input_quad ^ 5;
    print("q8 = input_quad ^ 5", q8);

    // operator ^ to raise a quad to a negative integer power
    Quad q9{};
    print("q9", q9);
    q9 = input_quad ^ (-1);
    print("q9 = input_quad ^ (-1)", q9);
    q9 = input_quad ^ (-2);
    print("q9 = input_quad ^ (-2)", q9);
    q9 = input_quad ^ (-3);
    print("q9 = input_quad ^ (-3)", q9);
    q9 = input_quad ^ (-4);
    print("q9 = input_quad ^ (-4)", q9);
    q9 = input_quad ^ (-5);
    print("q9 = input_quad ^ (-5)", q9);

    assert(q8 * q9 == one);

    cout << "Test completed successfully!" << endl;
    return 0;
}
```

Output

```
Quad zero           [0, 1, 0, 1]
Quad identity(1)    [1, 1, 0, 1]
Quad half(1,2)      [1, 2, 0, 1]
Quad q0             [0, 1, 0, 1]
Quad q1(2)          [2, 1, 0, 1]
Quad q2(2,3)        [2, 3, 0, 1]
Quad q3(2,3,4)      [2, 3, 4, 1]
Quad q4(2, 3, 4, 5) [2, 3, 4, 5]
q11                 [-1, 1, 1, 1]
q01234              [4, 1, 24, 5]
qs01234             [-4, 1, -24, 5]
m01234              [12, 1, 44, 5]
n01234              [12, 1, 44, 5]
qm1234              [1232, 135, 64, 15]
qmr1234             [1232, 135, 64, 15]
q4.norm() =         1.08444
size of q4 =        1.04137
conjugate of q4 =   [2, 3, -4, 5]
Quad q5{ qmr1234 }  [1232, 135, 64, 15]
Quad q5_inverse     [10395, 53392, -1215, 13348]
q6                  [10, -20, 0, 40]
q6 normalized       [-10, 20, 0, 40]
q6 reduced          [-1, 2, 0, 1]
++q6                [1, 2, 0, 1]
q6 += half          [1, 1, 0, 1]
q6++                [2, 1, 0, 1]
q6--                [1, 1, 0, 1]
--q6                [0, 1, 0, 1]
q7                  [1, 2, 0, 1]
q7 = half ^ 1       [1, 2, 0, 1]
q7 = half ^ 2       [1, 4, 0, 1]
q7 = half ^ 3       [1, 8, 0, 1]
q7 = half ^ 4       [1, 16, 0, 1]
q7 = half ^ 5       [1, 32, 0, 1]
About to create the quad [a, b, c, d]
Enter four numbers a, b, c, d, in that order:
1 2 3 4
input_quad          [1, 2, 3, 4]
q8                  [0, 1, 0, 1]
q8 = input_quad ^ 1 [1, 2, 3, 4]
q8 = input_quad ^ 2 [11, 8, 3, 4]
q8 = input_quad ^ 3 [29, 16, 45, 32]
q8 = input_quad ^ 4 [193, 64, 33, 16]
q8 = input_quad ^ 5 [589, 128, 843, 256]
q9                  [0, 1, 0, 1]
q9 = input_quad ^ (-1)[-4, 7, 6, 7]
q9 = input_quad ^ (-2)[88, 49, -48, 49]
q9 = input_quad ^ (-3)[-928, 343, 720, 343]
q9 = input_quad ^ (-4)[12352, 2401, -8448, 2401]
q9 = input_quad ^ (-5)[-150784, 16807, 107904, 16807]
Test completed successfully!
```

9 Evaluation Criteria

Evaluation Criteria		
Functionality	Testing correctness of execution of your program, Proper implementation of all specified requirements, Efficiency	60%
OOP style	Encapsulating only the necessary data inside your objects, Information hiding, Proper use of C++ constructs and facilities. No use of operator **new** and operator **delete**. No C-style coding and memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc.	20%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial pieces of code in submitted programs	10%
Presentation	Format, clarity, completeness of output, user friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	5%