Contents

1 Purpose

- Create an abstract data type (ADT)
- Implement the ADT, using the operator overloading facility of the C++ language
- Learn about function objects and how to define them

2 Background Information

A *data type* represents a set of data values sharing common properties, and thus the data type of a variable determines the set of values the variable can take.

An abstract data type (ADT) specifies a set of operations on a *data type*, independent of how the data type is actually represented and how the operations on the data type 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 Google search for "class Rational C++" will reveal many turnkey C++ classes, forcing homework assignments designed to provide practice with operator overloading to get a bit creative, choosing a *data type* that is not as ubiquitous as the rational or complex number ADTs, but one that lends itself to operator overloading just the same.

3 Data Type: Quad

In this assignment, we define the **Quad** data type as a set of objects, each comprising an ordered sequence of four real numbers. We denote a **Quad** object X by $\begin{bmatrix} x_1, x_2, x_3, x_4 \end{bmatrix}$ and $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$ interchangeably, where x_1, x_2, x_3, and x_4 are real numbers.

3.1 Special Quad Objects

Zero $Z = \begin{bmatrix} 0, 0, 0, 0 \end{bmatrix}$

Identity $I = \begin{bmatrix} 1, 0, 1, 0 \end{bmatrix}$

4 An ADT for Quads

- The *Quad* operations listed below are specified using the following notations:

α : a real number	X : $\begin{bmatrix} x_1, x_2, x_3, x_4 \end{bmatrix}$	$\lvert X \rvert$: absolute value of X
β : a real number	Y : $\begin{bmatrix} y_1, y_2, y_3, y_4 \end{bmatrix}$	$\lvert X_k \rvert$: absolute value of x_k, $k = 1, 2, 3, 4$

- **Scalar Addition and Subtraction**

  $\alpha \pm X = \left[\alpha \pm x_1, \alpha \pm x_2, \alpha \pm x_3, \alpha \pm x_4\right]$

  $X \pm \alpha = \pm(\alpha \pm X)$

- **Scalar Multiplication**

  $\alpha * X = \left[\alpha x_1, \alpha x_2, \alpha x_3, \alpha x_4\right]$

  $X * \alpha = \alpha * X$

- **Unary Addition and Subtraction**

  $+X = X$ and $-X = -1 * X$

- **Binary Addition and Subtraction**

$$X \pm Y = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \pm \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 \pm y_1 \\ x_2 \pm y_2 \\ x_3 \pm y_3 \\ x_4 \pm y_4 \end{bmatrix}$$

- **Multiplication**

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

- **Inversion**

  $X^{-1} = \beta^{-1} * \left[x_3, -x_2, x_1, -x_4\right]$ provided that $\beta = x_1 x_3 - x_2 x_4 \neq 0$

- **Division**

  $X/Y = X * Y^{-1}$

- **Scalar Division**

  $X/\alpha = X * \alpha^{-1}, \qquad \alpha \neq 0$

  $\alpha/X = \alpha * X^{-1}$

- **Absolute value**

  $|X_k| = |x_k|,\ k = 1, 2, 3, 4$

  $|X| = |x_1| + |x_2| + |x_3| + |x_4|$

- **Equality operators**

  equal to $X = Y$ if $|X - Y| \leq \epsilon$, where ϵ is a tolerance, a positive amount the value $|X - Y|$ can change and still be acceptable that $X = Y$.

  not equal to $X \neq Y \equiv \neg(X = Y)$ where the symbol $\neg$ denotes the negation operator.

- **Relational operators**

less than	$X < Y$ if $\neg(X = Y)$ and $\lvert X\rvert < \lvert Y\rvert$,
greater than	$X > Y \equiv Y < X$
greater than equal to	$X \geq Y \equiv \neg(X < Y)$
less than equal to	$X \leq Y \equiv X < Y$ or $X = Y$

5 Your Task

Implement the `Quad` ADT.

5.1 Representation

```
class Quad
{
private:
   std::array<double, 4> quad{};
public:
   static inline const double tolerance{ 1.0E-6 }; // C++17 and later
```

Note that it is quite common to make `static const`ants `public`.

5.2 Interface

5.2.1 Accessor and Mutator member functions

```
public:
   Quad get() const;
   void set(const Quad&);
```

5.2.2 Special Member Functions

- A normal constructor:

```
   Quad(double x1 = 0.0, double x2 = 0.0, double x3 = 0.0, double x4 = 0.0);
```

- The remaining "big five" special member functions are `default`ed, each of which in turn invokes the corresponding special member function of the class of each data member; in the case of `Quad`, the data members are of the built-in data type `double`, which is not even a class type, let alone having special member functions.

5.2.3 Overload the Compound Assignment Operators

Modifying their left-hand operand, these operators are commonly implemented as member functions.

`Quad op= Quad`	`X += Y` , `X -= Y` , `X *= Y` , `X /= Y`
`Quad op= double`	`X += a`, `X -= a`, `X *= a`, `X /= a`

5.2.4 Overload the Basic Arithmetic Binary Operators

Since they do not modify their operands and include symmetric versions, these operators are commonly implemented as free (non-member) functions.

`Quad op Quad`	`X + Y` , `X - Y` , `X * Y` , `X / Y`
`Quad op double`	`X + a`, `X - a`, `X * a`, `X / a`
`double op Quad`	`a + X` , `a - X` , `a * X` , `a / X`

Note That the last group of operations `double op Quad` cannot be implemented as member functions (why?)

5.2.5 Overload the Relational and Equality operators

Since these operators do not modify their operands, they are commonly implemented as free (non-member) functions.

`Quad op Quad`	`X == Y` , `X != Y`, `X < Y`, `X <= Y`, `X > Y`, `X >= Y`

5.2.6 Overload the Unary Increment and Decrement Operators

The pre-increment and pre-decrement operators do modify their single operand, but the post-increment and post-decrement operators do not. They are all commonly implemented as member functions.

`op Quad`	`+X`, `-X`, unary plus/minus
	`++X`, `--X`, pre-increment/decrement
	`X++`, `X--`, post-increment/decrement

5.2.7 Overload the Subscript Operator `[]`, both `const` and `non-const` Versions

Since the users expect to use the same notation for expressing the coordinates of a `Quad` object as defined above, use 1-based indexing regardless of the underlying representation of the coordinates. Throw an exception of the form `std::out_of_range("index out of bounds")` if the supplied subscript is invalid.

Usage: if `x` is a `const`/non-`const` `Quad`, then `x[1]`, `x[2]`, `x[3]`, and `x[4]` return a `const`/non-`const` reference to the coordinates `x.quad[0]`, `x.quad[1]`, `x.quad[2]`, and `x.quad[3]`, respectively.

5.2.8 Turn Objects of Quad Into Function Objects

An object of a class that overloads the function call operator `()` is called a **function object**.

You can overload the operator `()` as many time as you wish, of which each may return any of the types allowed for function return values, and unlike all the other operators, may have any number of parameters of the the types allowed for function parameters.

As a contrived example, overload the operator `()` five times:

`double operator()()`
Returns the largest coordinate values of the invoking `Quad` object.

`double operator()(size_t i)`
Returns the i'th coordinate value of the invoking `Quad` object.

`double operator()(size_t i, size_t j)`
Returns the larger of the i'th and j'th coordinate values of the invoking `Quad` object.

`double operator()(size_t i, size_t j, size_t k)`
Returns the largest of the i'th, j'th, and k'th coordinate values of the invoking `Quad` object.

`double operator()(size_t i, size_t j, size_t k, size_t l)`
Returns the largest of the i'th, j'th, k'th, and l'th coordinate values of the invoking `Quad` object.

Throw an exception of the form `std::out_of_range("index out of bounds")` if any of the supplied indices is invalid.

5.2.9 Overload the extraction (input) operator>> for reading a `Quad` object from an Input Stream

5.2.10 Overload the insertion (output) operator<< for a writing `Quad` object to an Output Stream

5.2.11 `absoluteValue()` member function

Implement an `absoluteValue()` member function that computes and returns the absolute value of the invoking object.

Since this member is not as common and well known as the arithmetic and relational operations, we choose to implement it as a named member function, using a meaningful name that reflects its functionality.

5.2.12 Introduce Private Facilitator

Feel free to introduce any number of private member functions to facilitate your task.

5.2.13 `inverse()` member function

Implement an `inverse()` member function that computes and returns the inverse of the invoking `Quad` object.

Again, since this member is not as common and well known as the arithmetic and relational operations, we choose to implement it as a named member function, using a meaningful name that reflects its functionality.

6 A Sample Test Driver

A sample test-driver program `test_Quad.cpp` has been posted posted on Moodle. For reference purposes, it is also reprinted here starting at page 7. Feel free to introduce any number of free functions to facilitate testing your `Quad` class operations.

7 Grading scheme

Functionality	• Correctness of execution of your program • Proper implementation of all specified requirements • Efficiency	60
OOP Style	• Encapsulating only the necessary data inside objects • Information hiding • Proper use of C++ constructs and facilities • No global variables • No use of the operator delete • No C-style memory functions such as malloc, alloc, free, etc.	20
Documentation	• Description of purpose of program • Javadoc comment style for all methods and fields • Comments for non-trivial code segments	10
Presentation	• Format, clarity, completeness of outpu • User friendly interface	5
Code Readability	• Meaningful identifiers, indentation, spacing	5

8 Testing Your Code

```
#include <cassert>
#include "Quad.h"
using std::cout;
using std::cin;
using std::endl;
/*
Tests class Quad. Specifically, tests:
overloaded constructors, overloaded compound assignment operator,
overloadedbasic arithmetic operator, overloaded unaryoperators,
overloaded pre/post-increment/decrement, overloaded subscripts,
overloaded function objects, overloaded input/output operators,
and overloaded relational operators.
@return 0 to indicate success.
*/
// function prototypes
void test_constructors_and_equality();
void test_multiplication_and_inverse();
void test_unary_operators();
void test_basic_arithmetic_operators();
void test_compound_assignment_operators();
void test_subscript_operator();
void test_relational_operators();
void test_function_objects();
void test_accessor_mutator();

int main()
{
   //test_insersion_extraction_operator();
   test_constructors_and_equality();
   test_multiplication_and_inverse();
   test_unary_operators();
   test_basic_arithmetic_operators();
   test_compound_assignment_operators();
   test_subscript_operator();
   test_relational_operators();
   test_function_objects();
   test_accessor_mutator();

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

```
void test_insersion_extraction_operator()
{
   Quad q;

   cout << "Please enter the numbers 4.5, 2.5, 7, 5, in that order\n\n";
   cin >> q;
   cout << "input = " << q << endl;
   assert(q == Quad(4.5, 2.5, 7, 5));
}
```

```
void test_constructors_and_equality()
{
   const Quad ZERO;
   // must not compile, because zero is const
   //ZERO[1] = 0;
   //ZERO[2] = 0;
   //ZERO[3] = 0;
   //ZERO[4] = 0;

   Quad q1a;                          // default ctor
   cout << "q1a = " << q1a << endl;   // cout << Quad
   assert(q1a == ZERO);               // Quad == Quad

   Quad q1b(2);                       // normal ctor with 1 arg
   cout << "q1b = " << q1b << endl;
   assert(q1b == Quad(2, 0, 0, 0));

   Quad q1c(2, 3);                    // normal ctor with 2 args
   cout << "q1c = " << q1c << endl;
   assert(q1c == Quad(2, 3, 0, 0));

   Quad q1d(2, 3, 8);                 // normal ctor with 3 args
   cout << "q1d = " << q1d << endl;
   assert(q1d == Quad(2, 3, 8, 0));

   Quad q1(2.5, 3.6, 8.7, 5.8);       // normal ctor with 4 args
}
```

```
void test_multiplication_and_inverse()
{
   const Quad IDENTITY(1, 0, 1, 0);
   Quad q1(2.5, 3.6, 8.7, 5.8);                  // normal ctor with 4 args
   Quad  q1_inverse = q1.inverse();              // inverse, copy ctor

   Quad q1_inverse_times_q1 = q1_inverse * q1;   // Quad * Quad
   assert(q1_inverse_times_q1 == IDENTITY);      // invariant, must hold

   Quad q1_times_q1_inverse = q1 * q1_inverse;
   assert(q1_times_q1_inverse == IDENTITY);      // invariant, must hold
}
```

```
void test_unary_operators()
{
   Quad q(2.5, 3.6, 8.7, 5.8);          // normal ctor with 4 args

   assert(+q == -(-q));                 // +Quad, -Quad
   Quad t = q;
   ++q;                                 // ++Quad
   assert(q == t + 1);
   --q;                                 // --Quad
   assert(q == t);

   Quad q_post_inc = q++;               // Quad++
   assert(q_post_inc == t);
   assert(q == t + 1);

   Quad q_post_dec = q--;               // Quad--
   assert(q_post_dec == t + 1);
   assert(q == t);

   Quad q2 = q++;                       //Quad++
   cout << "q = " << q << endl;
   cout << "q2 = " << q2 << endl;
   assert(q2 == q - Quad(1, 1, 1, 1));  // Quad - Quad

   Quad q3 = --q;                       // --Quad4D
   cout << "q = " << q << endl;
   cout << "q3 = " << q3 << endl;
   assert(q3 == q);
}
```

```
void test_basic_arithmetic_operators()
{
   Quad q1(2.5, 3.6, 8.7, 5.8);          // normal ctor with 4 args
   Quad q2(2, 3, 8);                     // normal ctor with 3 args

   cout << "\n";
   q2 += Quad(0, 0, 0, 5);               // Quad += Quad
   Quad q3 = q2 + 1.0;                   // Quad = Quad4D + int
   assert(q3 == Quad(3, 4, 9, 6));
   cout << "q3 = " << q3 << endl;

   q3 = 1 + q2;                          // Quad = double + Quad4D;
   assert(q3 == Quad(3, 4, 9, 6));

   Quad q4 = q3 - 1.0;                   // Quad = Quad4D - double
   assert(q4 == q2);
   cout << "q4 = " << q4 << endl;

   Quad q5 = 1.0 - q4;                   // Quad = double - Quad4D
   cout << "q5 = " << q5 << endl;
   assert(q5 == Quad(-1, -2, -7, -4));


   Quad q6 = q5 * 2.0;                   // Quad = Quad4D * double
   cout << "q6 = " << q6 << endl;
   assert(q6 == Quad(-2, -4, -14, -8));

   Quad q7 = -1 * q6;                    // Quad = double * Quad4D
   cout << "q7 = " << q7 << endl;
   assert(q7 == Quad(2, 4, 14, 8));
   assert(q7 / -1.0 == q6);              // Quad = Quad4D / double
   assert(1 / q7 == 1 * q7.inverse());   // double / Quad4D, inverse
   assert(-1.0 * q5 * 2.0 == q7);        // double * Quad4D * double

   Quad q8 = q1++;                       //Quad++
   Quad q9 = --q1;                       // --Quad4D
   q9--;                                 // Quad4D--
   cout << "q9 = " << q9 << endl;
   assert(q1 == 1 + q9);                 // double + Quad
   assert(q1 - 1 == q9);
   assert(-q1 + 1 == -q9);
   assert(2 * q1 == q9 + q1 + 1);
   assert(q1 * q1 == q1 * (1 + q9));
}
```

```
void test_compound_assignment_operators()
{
   Quad q1{ 3, 1, 7, 4 };
   q1 += q1;
   cout << "q1 = " << q1 << endl;
   assert(q1 == 2 * Quad(3, 1, 7, 4));

   Quad q2;
   q2 += (q1 / 2);
   cout << "q2 = " << q2 << endl;
   assert(q2 == Quad(3, 1, 7, 4));

   q2 *= 2;
   cout << "q2 = " << q2 << endl;
   assert(q2 == q1);

   q2 /= 2;
   cout << "q2 = " << q2 << endl;
   assert(q2 == q1 / 2);

   q2 += 10;
   cout << "q2 = " << q2 << endl;
   assert(q2 == (q1 + 20) / 2);

   q2 -= 10;
   cout << "q2 = " << q2 << endl;
   assert(q2 == 0.5 * q1);
}
```

```
void test_subscript_operator()
{
   Quad q(123, 6, 6, 4567.89);
   cout << "q = " << q << endl;

   // subscripts (non-const)
   q[1] = 3;
   q[2] = 1;
   q[3] = 7;
   q[4] = 4;
   cout << "q = " << q << endl;
   assert(q == Quad(3, 1, 7, 4));

   // subscripts (const)
   const Quad cq{ q };
   assert(cq == Quad(3, 1, 7, 4));
   assert(q == Quad(cq[1], cq[2], cq[3], cq[4]));
}
```

```
void test_relational_operators()
{
   Quad q{ 3, 1, 7, 4 };
   // relational operators
   double smallTol = Quad::tolerance / 10.0;
   Quad qNeighbor(3 - smallTol, 1 + smallTol, 7 - smallTol, 4 + smallTol);
   assert(q == qNeighbor);

   double tol = Quad::tolerance;
   assert(q != (q + tol));
   assert(q != (q + 0.25 * tol));
   assert(q == (q + 0.15 * tol));
   assert(q == q);

   assert(q < (q + 0.001));
   assert(q <= (q + 0.001));
   assert((q + 0.001) <= (q + 0.001));

   assert((q + 0.001) > q);
   assert((q + 0.001) >= q);
   assert((q + 0.001) >= (q + 0.001));
}
```

```
void test_function_objects()
{
   Quad q = Quad{ 4.5, 2.5, 7, 5 };

   // function objects
   assert(q() == 7.0);
   assert(q(1) == 4.5);
   assert(q(1, 4) == 5);
   assert(q(1, 2, 1) == 4.5);
   assert(q(2, 1, 3, 4) == 7);
}
```

```
void test_accessor_mutator()
{
   Quad q{ 4.5, 2.5, 7, 5 };
   Quad p{ q * 2 };
   assert(p.get() == q + q);
   p.set( q *= 2 );
   assert(-q / 4 + p ==  + q * 3 / 4);
}
```