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
- Use the C++ standard library container type `std::array` rather than raw dumb arrays.

2 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 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 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 Point4D

3.1 Point4D Data Type

The `Point4D` type represents points with four coordinates x_1, x_2, x_3, and x_4, all real numbers.

We denote a `Point4D` point X as $\left[x_1, x_2, x_3, x_4\right]$ and $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$, interchangeably.

3.1.1 Special Point4D Points

Zero $Z = \left[0, 0, 0, 0\right]$

Identity $I = \left[1, 0, 1, 0\right]$

3.2 Point4D Operations

Notation: $X = \left[x_1, x_2, x_3, x_4\right]$, $Y = \left[y_1, y_2, y_3, y_4\right]$, α and β denote real numbers

Operation	Definition
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	$X/\alpha = X * \alpha^{-1}$, $\alpha \neq 0$ $\alpha/X = \alpha * X^{-1}$
$\lvert X \rvert$**, Absolute value of** X	$\lvert x_1 \rvert + \lvert x_2 \rvert + \lvert x_3 \rvert + \lvert x_4 \rvert$
Relational operators	• $X = Y$ if $\lvert X - Y \rvert \leq \epsilon$, where ϵ is a tolerance: a positive amount the value $\lvert X - Y \rvert$ can change and still be acceptable that $X = Y$. • $X < Y$ if $\neg(X = Y)$ and $\lvert X \rvert < \lvert Y \rvert$ where $\neg$ denotes the negation operator. Recall that the definitions of the $<$, $=$, and $\neg$ operators are sufficient for deriving the definitions of the following common relational operators: • $X > Y \equiv Y < X$ • $X \neq Y \equiv \neg(X = Y)$ • $X \geq Y \equiv \neg(X < Y)$ • $X \leq Y \equiv X < Y$ or $X = Y$

4 Your Task

Implement the Point4D ADT described above.

4.1 Representation of Coordinates

There are several options for representing the four coordinates of Point4D objects, including, for example:

- four doubles x1, x2, x3, x4,
- an array double x[4],
- an array of four pointers to doubles; specifically: double *parray[4]; (hopefully never!)
- A standard library sequential container such as array, vector, list, forward_list, deque.
- etc.

In this assignment, we choose to use the C++ standard array class, a templated container class that models fixed-size arrays, providing an efficient and convenient alternative to raw dumb arrays.

The std::array container class is a template with two parameters: the type of the elements in the container and the fixed size of the container.

```
std::array<double, 4> point;
```

Common ways to both read and write the elements of point include

- using std::array's subscript operator[] overload For example, the following statements set point to the identity Point4D:

  ```
  point[0] = point[2] = 1.0;
  point[1] = point[3] = 0.0;
  ```

  Note that point is an object of class std::array, not a raw array. The statements above are effectively equivalent to

  ```
  point.operator[](0) = 1.0;
  point.operator[](2) = 1.0;
  point.operator[](1) = 0.0;
  point.operator[](3) = 0.0;
  ```

  For this to compile, the function calls in the assignment statements must, of course, each return a reference.

  This way involves no bounds-checking on the supplied subscripts.

- using std::array's at() member function. For example, the following statements set point to the identity Point4D:

  ```
  point.at(0) = point.at(2) = 1.0;
  point.at(1) = point.at(3) = 0.0;
  ```

  For this to compile, the function calls in the assignment statements must, of course, each return a reference. The statements above are effectively equivalent to

  ```
  point[0] = point[2] = 1.0;
  point[1] = point[3] = 0.0;
  ```

  The only difference is that std::array's at() member function will throw an std::out_of_range exception if the supplied subscript is outside the range of the array.

4.2 Representation of Classwide Tolerance

- Declare the following private members to represent a classwide tolerance:

```
private:
   static double tolerance;
   static void setTolerance(double tol);
   static double getTolerance();
```

4.3 Implementation

4.3.1 Static Member Defitions

1. Define the `static` members:

```
double Point4D::tolerance = 1.0E-6;
void Point4D::setTolerance(double tol) { tolerance = std::abs(tol); }
double Point4D::getTolerance() { return tolerance; }
```

4.3.2 Constructors and Destructor

2. A constructor taking four parameters of type `double`, specifying a default value of zero for each argument passed to the constructor.

   The constructor is required to be declared `explicit` to avoid conversion (through the one-argument constructor) from `double` to `Point4D` , which is mathematically undefined.

3. Defaulted copy constructor

```
Point4D(const Point4D&) = default;
```

   Justification
   The compiler synthesized copy constructor member-wise copies the members of its argument into the object being created. This is exactly the desired behavior when one `Point4D` object is copied to another, as `Point4D` doesn't handle any dynamic resources.

4. Defaulted assignment operator

```
Point4D& operator=(const Point4D&) = default;
```

   Justification
   The compiler synthesized assignment operator copy assigns the members of the right-hand side operand to the corresponding members of the left-hand side operand. This is exactly the desired behavior when one `Point4D` object is assigned to another.

5. Defaulted destructor

```
virtual ~Point4D() = default;
```

Justification

The compiler synthesized destructor doesn't do anything, which is exactly the desired behavior when a `Point4D` object goes out of scope.

4.3.3 Operator Overloads

6. Compound assignment operators. All are commonly implemented as member functions. All modify their left-hand side operands.

   `Point4D op= Point4D` `X += Y , X -= Y , X *= Y , X /= Y`

   `Point4D op= double` `X += a, X -= a, X *= a, X /= a`

7. Basic arithmetic operators. Not all can be implemented as members. None modifies its operands. For consistency, all are commonly implemented as free (non-member) functions.

   `Point4D op Point4D` `X + Y , X - Y , X * Y , X / Y`

   `Point4D op double` `X + a, X - a, X * a, X / a`

   `double op Point4D` `a + X , a - X , a * X , a / X`

   The last group of operations `double op Point4D` cannot be provided by member functions (why?)

8. Relational operators. All can be implemented as members. None modifies its operands. For consistency, all are implemented as free functions.

   `Point4D op Point4D` `X == Y , X != Y, X < Y, X <= Y, X > Y, X >= Y`

9. Unary operators. All are commonly implemented as members.

   `op Point4D` `+X`, `-X`, unary plus/minus

   `++X`, `--X`, pre-increment/decrement

   `X++`, `X--`, post-increment/decrement

10. Subscript `operator[]` (both const and non-const). Use 1-based indexing to preserve the mathematical notation above, regardless of the underlying representation. Must throw `std::out_of_range("index out of bounds")` if the supplied subscript is invalid.

    Usage: if `x` is a `Point4D`, then `x[1]`, `x[2]`, `x[3]`, `x[4]` correspond to the four coordinates of `x`, respectively.

11. Function call `operator()` overload that takes no arguments and returns a `double` approximating the absolute value of the invoking object.

    Usage: if `p` is a `Point4D`, then `p()` should return the absolute-value of `p`.

    The function call operator `()` enables `Point4D` objects such as `p` to behave like functions (hence the name "function objects"). You can overload it as many time as you wish, having each return a type of your choice.

12. Overloaded extraction (input) operator `>>` for reading `Point4D` objects

13. Overloaded insertion (output) operator `<<` for writing `Point4D` objects

14. An `absoluteValue()` member function to return the absolute value of the invoking object.

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

5 Operator Overloading Guidelines[1]

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

6 C++ Operator Overloading Rules

- Operator overloads can either be implemented as member functions or as free functions and cannot have default arguments.
- Implemented as free functions, a unary operator takes one argument, and a binary operator takes two arguments.
- Implemented as member functions, a unary operator takes no arguments, and a binary operator takes one argument.
- At least one argument must be a class object, for example, `Point4D`, in the case at hand.
- Specified by their use with built-in types, the precedence, grouping, and number of arguments of the C++ operators cannot be changed.

6.1 C++ Operator Precedence, Grouping, and number of arguments

The C++ operators and their precedence are listed on the next page. Operators at the top of the list evaluate before those at the bottom. Operators with the same precedence level are grouped together between horizontal lines. Operators that cannot be overloaded are listed in red. Operators that must be overloaded as class member functions are listed in blue. The remaining operators can be overloaded either as class member functions or as free (global, top level) functions.

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

C++ Operator	Meaning	Associativity	Usage
::	global scope	$R \to L$	::name
::	class, namespace scope	$L \to R$	name::member
.	direct member	$L \to R$	object.member
->	indirect member	$L \to R$	pointer->member
[]	subscript	$L \to R$	pointer[expr]
()	function call	$L \to R$	expr(arg)
()	type construction	$L \to R$	type(expr)
++ --	postfix increment/decrement	$L \to R$	lvalue++ lvalue--
++ --	prefix increment/decrement	$R \to L$	++lvalue --lvalue
sizeof	size of object	$R \to L$	sizeof expr
sizeof	size of type	$R \to L$	sizeof(type)
typeid	type identification	$R \to L$	typeid(expr)
const_cast	specialized cast	$R \to L$	const_cast<expr>
dynamic_cast	specialized cast	$R \to L$	dynamic_cast<expr>
reinterpret_cast	specialized cast	$R \to L$	reinterpret_cast<expr>
static_cast	specialized cast	$R \to L$	static_cast<expr>
()	traditional cast	$R \to L$	(type)expr
~	one's complement	$R \to L$	~expr
!	logical NOT	$R \to L$	!expr
$-, +$	unary minus, unary plus	$R \to L$	-expr, +expr
&	address of	$R \to L$	&lvalue
*	dereference	$R \to L$	*expr
new	create object	$R \to L$	new type
new[]	create array	$R \to L$	new type[]
delete	destroy object	$R \to L$	delete ptr
delete[]	destroy array	$R \to L$	delete [] ptr
.*	member dereference	$L \to R$	object.*ptr_to_member
->*	indirect member dereference	$L \to R$	ptr->*ptr_to_member
*, /, %	multiply, divide, modulus	$L \to R$	expr * expr, expr / expr, expr % expr
+, -	add, subtract	$L \to R$	expr + expr, expr - expr
<<,>>	left shift, right shift	$L \to R$	expr << expr,expr >> expr
<	less than	$L \to R$	expr < expr
<=	less than or equal to	$L \to R$	expr <= expr
>	greater than	$L \to R$	expr > expr
>=	greater than or equal to	$L \to R$	expr >= expr
==, !=	equal, not equal	$L \to R$	expr == expr, expr != expr
&	bitwise AND	$L \to R$	expr & expr
^	bitwise XOR	$L \to R$	expr ^ expr
|	bitwise OR	$L \to R$	expr | expr
&&	logical AND	$L \to R$	expr & expr
||	logical OR	$L \to R$	expr || expr
?:	conditional expression	$L \to R$	expr ? expr : expr
=	assignment	$R \to L$	lvalue = expr
*=	multiply update	$R \to L$	lvalue *= expr
/=	divide update	$R \to L$	lvalue /= expr
%=	modulus update	$R \to L$	lvalue %= expr
+=	add update	$R \to L$	lvalue += expr
-=	subtract update	$R \to L$	lvalue -= expr
<<=	left shift update	$R \to L$	lvalue <<= expr
>>=	right shift update	$R \to L$	lvalue >>= expr
&=	bitwise AND update	$R \to L$	lvalue &= expr
|=	bitwise OR update	$R \to L$	lvalue |= expr
^	bitwise XOR update	$R \to L$	lvalue ^= expr
throw	throw exception	$R \to L$	throw expr
,	comma	$L \to R$	expr, expr

7 Deliverables

1. Header files: `Point4D.h`
2. Implementation files: `Point4D.cpp`, `test_Point4D.cpp`
3. A `README.txt` text file (as described in the course outline).

7.1 A sample makefile

A sample makefile, in case you want to run your program outside an IDE under Linux

```
CXX = g++      # compiler command name
CXXFLAGS = -g -Wall -std=c++14    # compilation flags

EXEC = run        # "run" is the name of the final executable

# List of all object files required to build the executable "run"
OBJS = Point4D.o test_Point4D.o

${EXEC}: ${OBJS}   # the ultimate target EXEC depends on OBJS
	${CXX} ${CXXFLAGS} -o ${EXEC} ${OBJS}    # command to build EXEC

# target Point4D.o depends on Point4D.cpp Point4D.h
Point4D.o: Point4D.cpp Point4D.h
	${CXX} ${CXXFLAGS} -c Point4D.cpp   # command to build Point4D.o

# target test_Point4D.o depends on test_Point4D.cpp Point4D.h
test_Point4D.o: test_Point4D.cpp Point4D.h
	${CXX} ${CXXFLAGS} -c test_Point4D.cpp  # command to build test_Point4D.o

clean:
	rm -f ${EXEC} ${OBJS}  # remove the executable and all object files
```

- The symbol denotes a tab character
- Command lines must start with (unintuitive but important rule)
- Enter and save the boxed text above in a file named `Makefile` or `makefile`
- To remove the executable and all object files enter the command `make clean`
- To build the executable enter the command `make`
- To run your program enter `./run`

8 Sample Test Driver

A sample test-driver program `test_Point4D.cpp` has been posted posted on Moodle. For reference purposes, it is also reprinted here starting at page 10.

9 Marking scheme

60%	Program correctness
20%	Proper use of pointers, dynamic memory management, and `C++` concepts. No C-style memory functions such as `malloc`, `alloc`, `realloc`, `free`, etc. No C-style coding.
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program

10 test_Point4D.cpp

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cassert>
#include "Point4D.h"
using std::cout;
using std::cin;
using std::endl;
/*
Tests class Point4D. Specifically, tests constructors, compound assignment
operator overloads, basic arithmetic operator overloads, unary +, unary -,
pre/post-increment/decrement, subscripts, function objects,
input/output operators, and relational operators.
@return 0 to indicate success.
*/

int main()
{
   const Point4D ZERO;
   // must not compile, because zero is const
   //ZERO[1] = 0;
   //ZERO[2] = 0;
   //ZERO[3] = 0;
   //ZERO[4] = 0;
   const Point4D IDENTITY(1, 0, 1, 0);

   Point4D m1a;                          // default ctor
   cout << "m1a = " << m1a << endl;      // cout << Point4D
   assert( m1a == ZERO);                 // Point4D == Point4D

   Point4D m1b(2);                       // normal ctor with 1 arg
   cout << "m1b = " << m1b << endl;
   assert(m1b == Point4D(2, 0, 0, 0));

   Point4D m1c(2, 3);                    // normal ctor with 2 args
   cout << "m1c = " << m1c << endl;
   assert(m1c == Point4D(2, 3, 0, 0));

   Point4D m1d(2, 3, 8);                 // normal ctor with 3 args
   cout << "m1d = " << m1d << endl;
   assert(m1d == Point4D(2, 3, 8, 0));

   Point4D m1(2.5, 3.6, 8.7, 5.8);                 // normal ctor with 4 args
   Point4D  m1_inverse = m1.inverse();             // inverse, copy ctor
```

```
Point4D m1_inverse_times_m1 = m1_inverse * m1;   // Point4D * Point4D
assert(m1_inverse_times_m1 == IDENTITY);         // invariant, must hold

Point4D m1_times_m1_inverse = m1 * m1_inverse;
assert(m1_times_m1_inverse == IDENTITY);         // invariant, must hold

assert(+m1 == -(-m1));                           // +Point4D, -Point4D
Point4D t1 = m1;
++m1;                                            // ++Point4D
assert(m1 == t1 + 1);
--m1;                                            // --Point4D
assert(m1 == t1);

Point4D m1_post_inc = m1++;                      // Point4D++
assert(m1_post_inc == t1);
assert(m1 == t1 + 1);

Point4D m1_post_dec = m1--;                      // Point4D--
assert(m1_post_dec == t1 + 1);
assert(m1 == t1);

cout << "\n";
m1d += Point4D(0, 0, 0, 5);                  // Point4D += Point4D
Point4D m2 = m1d  + 1.0;                       // Point4D = Quad4D + int
assert(m2 == Point4D(3, 4, 9, 6));
cout << "m2 = " << m2 << endl;

m2 = 1 + m1d;                                // Point4D = double + Quad4D;
assert(m2 == Point4D(3, 4, 9, 6));

Point4D m3 = m2 - 1.0;                       // Point4D = Quad4D - double
assert(m3 == m1d);
cout << "m3 = " << m3 << endl;

Point4D m4 = 1.0 - m3;                       // Point4D = double - Quad4D
cout << "m4 = " << m4 << endl;
assert(m4 == Point4D(-1, -2, -7, -4));

Point4D m5 =  m4 * 2.0 ;                     // Point4D = Quad4D * double
cout << "m5 = " << m5 << endl;
assert(m5 == Point4D(-2, -4, -14, -8));
```

```
Point4D m6 = -1 *  m5;                          // Point4D = double * Quad4D
cout << "m6 = " << m6 << endl;
assert(m6 == Point4D(2, 4, 14, 8));
assert(m6 / -1.0 == m5);                        // Point4D = Quad4D / double
assert(1/m6  == 1*m6.inverse());                // double / Quad4D, inverse
assert(-1.0 * m4  * 2.0 == m6);                 // double * Quad4D * double

Point4D m7 = m1++;                              //Point4D++
cout << "m1 = " << m1 << endl;
cout << "m7 = " << m7 << endl;
assert(m7 == m1 - Point4D(1, 1, 1, 1));         // Point4D - Point4D

Point4D m8 = --m1;                             // --Quad4D
cout << "m1 = " << m1 << endl;
cout << "m8 = " << m8 << endl;
assert(m8 == m1 );

m8--;                                          // Quad4D--
cout << "m8 = " << m8 << endl;
assert(m1 == 1 + m8);                          // double + Point4D
assert(m1 - 1 == m8);
assert(-m1 + 1 == -m8);
assert(2 * m1 == m8 + m1 + 1);
assert(m1 * m1 == m1 *(1 + m8));

Point4D m9(123, 6, 6, 4567.89);
cout << "m9 = " << m9 << endl;

// subscripts (non-const)
m9[1] = 3;
m9[2] = 1;
m9[3] = 7;
m9[4] = 4;
cout << "m9 = " << m9 << endl;
assert(m9 == Point4D(3, 1, 7, 4));

// relational operators
double smallTol = Point4D::getTolerance() / 10.0;
Point4D m9Neighbor(3 - smallTol, 1 + smallTol, 7 - smallTol, 4 + smallTol);
assert(m9 == m9Neighbor);

double tol = Point4D::getTolerance();
assert(m9 != (m9 + tol));
assert(m9 != (m9 + 0.25 * tol));
```

```
    assert(m9 == (m9 + 0.15 * tol));
    assert(m9 == m9);

    assert(m9 < (m9 + 0.001));
    assert(m9 <= (m9 + 0.001));
    assert((m9 + 0.001) <= (m9 + 0.001));

    assert((m9 + 0.001) > m9);
    assert((m9 + 0.001) >= m9);
    assert((m9 + 0.001) >= (m9 + 0.001));

    // compound operators

    m9 += m9;
    cout << "m9 = " << m9 << endl;
    assert(m9 == 2 * Point4D(3, 1, 7, 4));

    Point4D m10;
    m10 += (m9 / 2);
    cout << "m10 = " << m10 << endl;
    assert(m10 == Point4D(3, 1, 7, 4));

    m10 *= 2;
    cout << "m10 = " << m10 << endl;
    assert(m10 == m9);

    m10 /= 2;
    cout << "m10 = " << m10 << endl;
    assert(m10 == m9/2);

    m10 += 10;
    cout << "m10 = " << m10 << endl;
    assert(m10 == (m9 +20) / 2);

    m10 -= 10;
    cout << "m10 = " << m10 << endl;
    assert(m10 == 0.5 * m9);
```

```

    //testing operator>>
    Point4D input;

    cout << "Please enter the numbers 1.5, 2.5, 3, 4, in that order\n\n";
    cin >> input;
    cout << "input = " << input << endl;

    Point4D diff = input - Point4D(1.5, 2.5, 3, 4);
    assert(diff.absValue() <= tol);     // absolute value
    assert(diff() <= tol);              // function object

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