**Operator overloading**

Q5.1 Recall the `Point2D` class described in lecture 7 and in exercise sheet 4 (questions 4.1–4.3). We can add two `Point2D` objects componentwise by adding their x and y values:

```cpp
Point2D a(3.0, 4.0), b(12.0, 5.0);
// create c and set it to "a + b"
Point2D c(a.GetX()+b.GetX(), a.GetY()+b.GetY());
```

This is cumbersome, however. It would be easier to write

```cpp
Point2D a(3.0, 4.0), b(12.0, 5.0);
// create c and set it to "a + b"
Point2D c = a.Add(b);
```

where `Add` is a public member function of `Point2D` that takes a `Point2D` as a parameter and returns a `Point2D`:

```cpp
Point2D Add(const Point2D &rhs)
{
    Point2D returnVal(x + rhs.x, y + rhs.y);
    return returnVal;
}
```

Implement this member function in the `Point2D` class and check that it works as expected. Note that we although we are adding two `Point2D` objects, we only pass one argument (the one on the right-hand side of the +) to the member function. The other operand (on the left-hand side) is the object on which we are calling the `Add` function. Note also that since `Add` is a member function of `Point2D`, it can access the private member variables `x` and `y` of `Point2D` – both of the current object and of the parameter `rhs`.

Q5.2 It would be even more straightforward if we could just write

```cpp
Point2D a(3.0, 4.0), b(12.0, 5.0);
// create c and set it to "a + b"
Point2D c = a + b;
```

We can do this with operator overloading. To overload the + operator, we simply need to define a member function of `Point2D` identical to the `Add` function in the previous question, but with the name `'operator+'`:

Implement this member function and check that it works as expected. Then implement an `operator-` to overload the - operator and allow code such as:

```cpp
Point2D a(3.0, 4.0), b(12.0, 5.0);
// create c and set it to "a - b"
Point2D c = a - b;
```
Most operators cannot be used with a class unless they are specifically overloaded for the class. However, C++ defines a default implementation of the = operator for any new class, which just copies the value of all member variables, so that

```cpp
Point2D a(3.0, 4.0), b(12.0, 5.0);
b = a; // set b to a
std::cout << "b = (" << b.GetX() << ", " << b.GetY() << ")" << std::endl;
```

displays (3, 4). The = operator can still be overloaded to change this default behaviour, if required.

Q5.3 Thinking of our Point2D objects as two-element vectors, the definitions + and - that we have just defined (adding one Point2D to another Point2D) correspond to vector addition. We can define the * operator to mean multiplication of a Point2D by a scalar (i.e. by a double), but this requires a slightly different approach, since the two operands of the * operator will be of different types (one scalar, one vector).

In this case there are two operators to define, one for Point2D * double and one for double * Point2D. The first of these (where Point2D is the left-hand operator) can be defined in a very similar way to operator + above, except this time using the operator * member function, taking a double parameter.

Recall that the operator overload goes in the class that is on the left-hand side of the operator. To implement double * Point2D we would therefore need to add an operator * member function to the double type. This is not possible (we cannot alter the basic types such as double or int) so a different approach is needed. Instead we can use a so-called friend non-member function, defined (like a member function) inside the Point2D class, as

```cpp
friend Point2D operator*(double lhs, const Point2D &rhs) {
    Point2D returnVal(rhs.x*lhs, rhs.y*lhs);
    return returnVal;
}
```

which takes both the parameters on the left-hand side and the right-hand side as arguments.

Implement both the Point2D * double and double * Point2D functions, and check that they are working by making sure that

```cpp
Point2D a(3.0, 4.0);
Point2D b = a*3.0, c = 4.5*a;
```

produces the expected values for b and c.

Q5.4 It would be convenient if we could display a Point2D object by writing

```cpp
Point2D a(3.0, 4.0);
std::cout << a << std::endl;
```

As our Point2D class stands, this code results in an error, since C++ does not know how it should display our class. We can fix this by implementing another friend non-member function in the class:
friend std::ostream& operator<<(std::ostream& os, const Point2D& rhs)
{
    os << // complete the function here
    return os;
}

Add this to your code, completing the commented statement so that it outputs the x and y member variables. Test it by running the code at the top of this question.

**Q5.5** Recall that when we pass parameters to a function, we can pass by value, by reference, and by const reference. When passing by const reference, C++ prevents us from modifying the value of that parameter inside the class. If the parameter being passed is simple numerical type (const int &p, say), then it is clear what is meant by 'modifying the value' of p: we are allowed to use (evaluate) the value of our parameter p, e.g.

```cpp
void MyFunction(const int &p)
{
    int x = 3*p;
}
```

but we are not allowed to change the value of p:

```cpp
void MyFunction(const int &p)
{
    p = 5; // error! can't change the value of a const reference
}
```

If the parameter passed by const reference is an object of class type, an additional question is posed: can we call member functions on that object? The answer is yes, but only for member functions that do not modify the object. We mark these member functions by adding a const after their name, e.g.

```cpp
double DistanceToOrigin() const
{
    return std::sqrt(x*x+y*y);
}
```

Go through your code for the Point2D object and add const to all member functions that do not alter the values of x and y. Check that you can now call these member functions (and only these ones) on parameters passed by const reference.

**Q5.6** We might find it useful to access the x and y components of our Point2D class through a square-bracket syntax, like a std::vector, where for a Point2D p, the x-component is accessed through p[0] and the y-component through p[1]. In this question we implement a way to read the values of x and y through this syntax, and in the following question we implement a way to modify them.

We do this by overloading operator[], a member function of the Point2D object which takes an int (0 or 1) and returns the double value of x or y.
double operator[](int index) const
{
    // ... implement operator here
}

Note that this operator overload is declared const as it returns the value of x or y but does not modify them. Write appropriate code in this function to access the x and y components of the Point2D, and test your code with

```cpp
int main()
{
    Point2D p(1.0, 2.0);
    std::cout << p[0] << "", " << p[1] << std::endl;
    return 0;
}
```

Q5.7 In the previous question, we were able to access the elements of the Point2D using p[0] and p[1]. But if we try to set the elements through these functions, as we can with a std::vector, an error occurs:

```cpp
int main()
{
    Point2D p(1.0, 2.0);
    p[0] = 3.0; // error!
    std::cout << p[0] << "", " << p[1] << std::endl;
    return 0;
}
```

To modify the components through the square bracket syntax, we need a second overload of the operator[], which takes the form,

```cpp
double &operator[](int index)
{
    // ...
}
```

Note that this function returns a double & – a reference to a double – which allows the returned value (the component x or y) to be changed. This is similar to the way in which passing a parameter to a function by reference allows the value passed to be changed. Note also that this overload of operator[] is not declared as const, because it does modify the class.

Write appropriate code in this function to set the x and y components of the Point2D, and test your code with

```cpp
int main()
{
    Point2D p(1.0, 2.0);
    p[0] = 3.0;
    p[1] = 4.0;
    std::cout << p[0] << "", " << p[1] << std::endl;
```
What behaviour do we want if the user tries \( p[3] = 4 \)? How can this be implemented?