2.4 - Basic Operator Overloading

Exercise 1: Adding Operators to the Point class

By supporting operators, you can make your classes easier and more natural to use. However you must not "overuse" operators. Only use operators if the functionality of the operator is clear without reading documentation. Thus adding mathematical operators to a complex number class is good but using a + operator with a double as an argument on a point to increase the x-coordinate is questionable. So use operators with care.

In this exercise we add a few operators to the *Point* class. Most operators do not change the original objects but return the result as a new object. Normally only the = operator and += and variants change the original object. Add the following operators:

```
Point operator - () const;               // Negate the coordinates.
Point operator * (double factor) const;  // Scale the coordinates.
Point operator + (const Point& p) const; // Add coordinates.
bool operator == (const Point& p) const; // Equally compare operator.
Point& operator = (const Point& source); // Assignment operator.
Point& operator *= (double factor);      // Scale the coordinates & assign.
```

Most operators above are self-explanatory. The = and *= operator changes the current object so they can't be const functions. To make it possible to chain the = and *= operators (E.g.: `a=b+=c`), it must return the "this" object.

Change the main program to test the operators.

Since the assignment operator is part of the canonical header file, add this operator also to the *Line* and, if applicable, the*Circle* class. Always be sure to preclude self-assignment inside your assignment operator functions.

Exercise 2: Ostream << Operator

It would be nice if you could send a point or a line directly to the *cout* object without calling the *ToString()* method, just as with the primitive types. This is possible by adding a << operator function that has on the left an *std::ostream* and on the right the point or line object. Since you can't add a member function to the *std::ostream* class, you have to create it as a global function (outside the class definition, but inside the class header file):

```
ostream& operator << (ostream& os, const Point& p); // Send to ostream.
```

The implementation uses the << operator to send data to the *os* input argument. Since it is a global function, you can't access the private members of *Point*. To simplify things, you can use the *ToString()* method of *Point* to get the string to send to the *os* argument.

Also create a similar << operator for printing lines (and circles if you made a circle class). Adapt the test program to test the << operator for points and lines.

Exercise 3: Constructors as conversion operator

In this exercise we are going to do a little experiment. First add to the *Point* class a constructor that accepts one *double* as an argument. The implementation should set both the x- and y-coordinate to this value.

Next try the following code in the test program:

```
Point p(1.0, 1.0);
if (p==1.0) cout<<"Equal!"<<endl;
else cout<<"Not equal"<<endl;
```

Will this code compile and can you explain why?

It turns out that the *Point* constructor with the single *double* argument is implicitly used to convert the number in the *if* statement to a *Point* object. Thus constructors are used as implicit conversion operators.

To prevent the usage of constructors are implicit conversion operators, declare the constructor as *explicit*:

```
explicit Point(double value);
```

Now try to compile the program again and you will see that now the *if* statement gives a compiler error. You can still use the constructor as conversion operator but then explicitly:

```
if (p==(Point)1.0) cout<<"Equal!"<<endl;
```

Exercise 4: Friends

Normally, only member functions of a class can access the private members of that class. Global functions and other classes can't access the private members of that class. You can make an exception on that rule by declaring certain functions or classes as friend.

For example the << operator for sending the point or line to the *std::ostream* class had to be a global function and thus can't access the private members. Move the << operator of *Point* and *Line* inside the class definition and declare it as *friend*. The function remains a global function but it can now access the data members directly without the need for calling the getters or *ToString()* function.

Normally, friends are to be avoided because it violates the data hiding principle. But in case of global operator functions it makes sense because you would actually want to make those global operator functions as member function but this was not possible.

2.5 - The Free Store

Exercise 1: Dynamically Creating Objects

Until now, we created objects on the stack. The stack is limited in size thus when creating many objects, the stack will overflow. Also arrays created on the stack can only have a fixed size determined at compile time. To overcome these problems we have to create objects on the heap using *new*.

In the main program, create *Point* objects on the heap with *new* using the default constructor, constructor with coordinates and the copy constructor and assign it to pointer (Point*) variables. Note that you can't directly pass a pointer variable as argument to the copy constructor. The pointer must first be dereferenced when passing it to the copy constructor. (*Point* p2=new Point(*p1)*).

Next call the *Distance()* function on the pointers and try to send the *Point* pointers to *cout*. You need to dereference the pointer when passing it as argument. Don't forget to delete the object created with *new*. Test the main program.

Next, we will create an array of points. First ask the user for the size of the array and read it using *cin*. Then try to create an array on the stack using the entered size. You will see a compiler error. Arrays on the stack must have the size set at compile time. Now create the array on the heap using *new*. Can you use other constructors than the default constructor for the objects created in the array? Don't forget to delete the array after use. Don't forget the square brackets when deleting arrays!

Some C++ compilers (e.g. GCC) support variable length arrays (VLA) where the size can be determined at run-time. However this is a C99 feature that is not in the C++ standard. Because VLA is not in the C++ standard you should avoid its usage since it will lead to less portable code.

Exercise 2: Creating Array of Pointers

In this exercise we make it a little more complex. With an array of *Points*, all points are created with the default constructor. When creating an array of pointers, each element in the array must be created separately but can be created with other constructors than the default constructor. Thus creating an array of pointers is a two step process:

- Create an array of *Point* pointers with 3 elements on the heap.
- Create for each element in the array a point on the heap.
- Iterate the array and print each point in the array.
- Iterate the array again and delete each point in the array.
- Delete the array itself.

Also make a drawing of the memory layout.

Exercise 3: Creating Array Class

It is easy to forget to delete memory created with *new*. A good practice is to let a class manage memory. Then the client of that class does not have to manage memory and can't forget to delete memory. So instead of creating a C array with new, you can use an array class that handle memory for you.

In this exercise we are going to create an array class for *Point* objects (see Figure 5):

- Add a source- and header file for the *Array* class to your current project.
- Add a data member for a dynamic C array of *Point* objects (`Point* m_data`).
- Add a data member for the size of the array.
- Add a default constructor that allocates for example 10 elements.
- Add a constructor with a *size* argument. The implementation should allocate the number of elements specified by the*size* input argument.
- Add a copy constructor. Keep in mind that just copying the C array pointer will create an *Array* object that shares the data with the original *Array* object. Thus you need to allocate a new C array with the same size and copy each element separately.
- Add a destructor. It should delete the internal C array. Don't forget the square brackets.
- Add an assignment operator. Keep in mind you can't copy only the C array pointers just as in the case of the copy constructor.
- Also don't forget to delete the old C array and allocate new memory before copying the elements. This is because C arrays can't grow.

Further check if the source object is not the same as the *this* object. If you don't check it, then a statement like `arr1=arr1` will go wrong. The internal C array will then be deleted before it is copied.

- Add a *Size()* function that returns the size of the array.
- Add a *SetElement()* function that sets an element. When the index is out of bounds, ignore the "set". We will add better error handling later.
- Add a *GetElement()* function. You can return the element by reference since the returned element has a longer lifetime than the *GetElement()* function. When the index is out of bounds, return the first element. We will add better error handling later.
- You can also add a square bracket operator. Return a reference so the [] operator can be used for both reading and writing elements. When the index is out of bounds, return the first element. We will add better error handling later.
  ```
  Point& operator [] (int index);
  ```
- In the main program, test the *Array* class.

Figure 5: Array class containing Point objects

To make the array class complete, you should also add the following *const* version of the square bracket operator. Can you explain why you would need this?:

```
const Point& operator [] (int index) const;
```

2.6 - Namespaces

Exercise 1: CAD and Container Namespaces

To avoid name conflicts, programmers can place their classes in a namespace. For example the standard library is placed in a namespace called *std*.

You should put your classes in your own namespace. Thus place the CAD classes (*Shape*,*Point*, *Line*, etc) in the namespace: `YourName::CAD`

Place the container classes (*Array*) in the namespace: `YourName::Containers`

Now access the classes in your own namespace using:

- Full class name including namespace for the *Point* used in the *Array* class. Note that you can use only the *CAD* part of the namespace without the *YourName* part because the *Point* is also in the *YourName* namespace.
- In the main program, the full namespace for *Point* class.
- In the main program, using declaration for using a single class (*Line*).
- In the main program, using declaration for a complete namespace (*Containers*).
- In the main program, using the *Circle* class by creating a shorter alias for the *YourName::CAD* namespace.