COEN 79L - Object-Oriented Programming and Advanced Data Structures

Lab 2

- Sub-Project 1: **Statistician**

Specify, design, and implement a class called `statistician`.
After a `statistician` is initialized, it can be given a sequence of double numbers. Each number in the sequence is given to the statistician by activating a member function called `next`.
For example, we can declare a statistician called `s`, and then give it the sequence of numbers 1.1, –2.4, 0.8 as shown here:

```
statistician s;
s.next(1.1);
s.next(-2.4);
s.next(0.8);
```

After a sequence has been given to a statistician, there are various member functions to obtain information about the sequence.

- Name the class "`statistician`" with all lowercase letters.
- The function "`next`" should add a number to the sequence.
- The function "`length`" should return the number of values in the sequence.
- The function "`sum`" should return the sum of the numbers in the sequence.
- The function "`mean`" should return the mean of the numbers in the sequence.
- The function "`minimum`" should return the smallest number in the sequence.
- The function "`maximum`" should return the largest number in the sequence.
- The function "`reset`" should erase the sequence.

Notice that the length and sum functions can be called at any time, even if there are no numbers in the sequence. In this case of an "empty" sequence, both length and sum will be zero. But the other member functions all have a precondition requiring that the sequence is non-empty.

You should also provide a member function that erases the sequence (so that the statistician can start afresh with a new sequence).

Notes: Do not try to store the entire sequence (because you don't know how long this sequence will be). Instead, just store the necessary information about the sequence: What is the sequence length? What is the sum of the numbers in the sequence? What are the last, smallest, and largest numbers? Each of these pieces of information can be stored in a private member variable that is updated whenever `next` is activated.

In addition to the above functions, add the following to your documentation and implement them:

```
// NON-MEMBER functions for the statistician class:
//   statistician operator +(const statistician& s1, const statistician& s2)
//     Postcondition: The statistician that is returned contains all the
//     numbers of the sequences of s1 and s2.
//   statistician operator *(double scale, const statistician& s)
//     Postcondition: The statistician that is returned contains the same
//     numbers that s does, but each number has been multiplied by the
//     scale number.
//   bool operator ==(const statistician& s1, const statistician& s2)
//     Postcondition: The return value is true if s1 and s2 have the zero
//     length. Also, if the length is greater than zero, then s1 and s2 must
//     have the same length, the same  mean, the same minimum,
//     the same maximum, and the same sum.
```

- Sub-Project 2: **Pseudorandom number generator**

In this project you will design and implement a class that can generate a sequence of **pseudorandom** integers, which is a sequence that appears random in many ways.

The approach uses the **linear congruence method**, explained here. The linear congruence method starts with a number called the **seed**. In addition to the seed, three other numbers are used in the linear congruence method, called the **multiplier**, the **increment**, and the **modulus**. The formula for generating a sequence of pseudorandom numbers is quite simple. The first number is:

```
(multiplier * seed + increment) % modulus
```

This formula uses the `%` operator, which computes the remainder from an integer division.

Each time a new random number is computed, the value of the seed is changed to that new number. For example, we could implement a pseudorandom number generator with `multiplier` = 40, `increment` = 725, and `modulus` = 729. If we choose the `seed` to be 1, then the sequence of numbers will proceed as shown here:

First number

```
= (multiplier * seed + increment) % modulus
= (40 * 1 + 725) % 729 = 36
```

and 36 becomes the new seed.

Next number

```
= (multiplier * seed + increment) % modulus
= (40 * 36 + 725) % 729
= 707
```

and 707 becomes the new seed.

Next number

```
= (multiplier * seed + increment) % modulus
= (40 * 707 + 725) % 729
= 574
```

and 574 becomes the new seed, and so on.

These particular values for `multiplier`, `increment`, and `modulus` happen to be good choices. The pattern generated will not repeat until 729 different numbers have been produced. Other choices for the constants might not be so good.

For this project, **design and implement a class** that can generate a pseudorandom sequence in the manner described. The initial `seed`, `multiplier`, `increment`, and `modulus` should all be parameters of the constructor. There should also be a member function to permit the `seed` to be changed, and a member function to generate and return the next number in the pseudorandom sequence.

- Name the class `rand_gen`
- The order of the parameters for the constructor of `rand_gen` should be: `seed`, `multiplier`, `increment`, and then `modulus`
- Name the function to generate the next number "`next`".
- Name the function to change the value of the seed "`set_seed`".

COEN 79L - Object-Oriented Programming and Advanced Data Structures

Lab 2

Submission guideline

The code for each of these classes should be divided across two files:

1. A `.h` file for the interface/class definition.
2. And a `.cpp` file for the implementation of the class' functions.

You should be submitting these files:

Sub-project 1: `statistician.h, statistician.cpp`

Sub-project 2: `random.h, and random.cpp`

Further, you should upload these as **separate files** rather than zipping or tarring them together.

You **should not** include a `main` function in any of these files, but you should write one in a separate `.cpp` file in order to test the functionality of your classes before submitting your code. You should not include the files with your `main` function(s) in your submission.

Regarding using namespace, for the statistician project, your code should look like this:

statistician.h	statistician.cpp

`#ifndef STATS_H`
`#define STATS_H`
`#include <iostream>`
 	
`namespace coen79_lab1`
`{`
`  //class definition…`
`}`
 	
`#endif`
	`#include <cassert>`
	`#include <iostream>`
	`#include "statistician.h"`
	
	`using namespace std;`
	`using namespace coen79_lab1;`
	
	`namespace coen79_lab1`
	`{`
	`  // implementations`
	`}`

Follow a similar approach for the random number generator.

▪ Appendix: **Member function or non-member function**

If you define your operator overloaded function **as member function**, then the compiler translates expressions like `s1 + s2` into `s1.operator+(s2)`. **That means, the operator overloaded member function gets invoked on the first operand.** That is how member functions work!

But what if the first operand is not a class? **There's a major problem if we want to overload an operator where the first operand is not a class type, rather say `double`.** You cannot write like this `10.0 + s2`. However, you can write operator overloaded member function for expressions like `s1 + 10.0`.

To solve this ordering problem, we define operator overloaded function as friend IF it needs to access private members. **Make it friend ONLY when it needs to access private members.** Otherwise simply make it **non-friend non-member** function to improve encapsulation!

```
class Sample
{
 public:
    Sample operator + (const Sample& op2); //works with s1 + s2
    Sample operator + (double op2); //works with s1 + 10.0

    //Make it `friend` only when it needs to access private members.
    //Otherwise simply make it **non-friend non-member** function.

    friend Sample operator + (double op1, const Sample& op2); //works with 10.0 + s2
}
```

When declaring it as a member function of a class, the left operand always has to be an object of that class, because it is being invoked as `s1.operator+(s2).`
For overloading the operators as a member function, you could pass in whatever data type you wanted as `s2`, it doesn't necessarily have to be another of whatever object we're working with.
As per the example above, it has 2 overloaded member operator+. One of them takes in an object, and one takes in a double, allowing for more flexibility. With both of these implemented, you can do `s1 + s2`; as well as `s1 + 10.0`;

However, if you did `10.0 + s1`; this would give you an error, because it does not make sense to say `(10.0).operator+(s1),` which is what this is trying to do.

If you want to allow something like `10.0 + s1` (or in more generic terms, non-object + object), then you need to declare a **Non-Member** overloaded operator.
This would be declared **outside** the class definition as something like:

```
Sample operator+(double op1, const Sample& s2);
```

This would allow you to do `10.0 + s1;`

However, as I declared it here, this function **does not** have direct access to any private variables of the Sample class. It can only access the public variables and functions. If you do not need direct access to the private data, then it is preferred to do it this way.

If you do need direct access to private variables, that is when we would declare it as a **friend** function.