4 Classes

4.1 Preamble

The exercises below are intended to provide a means to test your understanding of the programming-related material covered in the course. It is highly recommended that you work through these exercises as you cover the corresponding topics in the video lectures. By doing this, you will greatly strengthen your understanding of the material in these video lectures, which will greatly reduce the amount of pain and suffering required to complete the programming assignments in the course. In exercises that require the building (i.e., compiling and linking) of code, the CMake tool should be used for this purpose. For additional information about these exercises, refer to Section 1 of this document.

4.2 Topics Covered

These exercises cover classes, including such things as: data/function/type members, access specifiers, friends, const correctness, default/copy/move constructors, other constructors, destructors, copy/move assignment operators, initializer lists, operator overloading, and static/non-static members.

4.3 Exercises

1. (a) **[classes, data/function members, access specifiers]**
   Write the code for a class, called `Counter`, that can be used to represent a simple integer counter. The value of a counter can be set, queried, and incremented. More specifically, the class should meet the specifications given below.
   - The class has the following private members:
     - a data member called `count_` having type **`int`** that holds the current value of the counter.
   - The class has the following public members:
     - a function member called `getValue` that takes no parameters and returns the current value of the counter.
     - a function member called `setValue` that takes a single parameter and sets the current value of the counter to the value of this parameter; the function does not return any value.
     - a function member called `incrementValue` that increments (i.e., adds one to) the counter value and does not return any value.
   - The `getValue`, `setValue`, and `incrementValue` member functions should not be inline.

   Test your class with the `myTest` function given below. (To save typing, this function can be downloaded from the course web site in the file `myTest.cpp`.) Do all of the values printed by this test code make sense? What is the initial value of `counter1` printed on line 4 of the function `myTest`?

```
1  void myTest()
2  {
3      Counter counter1;
4      std::cout << "value for counter1: " << counter1.getValue() << "\n";
5
6      counter1.setValue(0);
7      Counter counter2(counter1);
8      Counter counter3;
9      counter3 = counter2;
10
11     std::cout << "value for counter1: " << counter1.getValue() << "\n";
12     std::cout << "value for counter2: " << counter2.getValue() << "\n";
13     std::cout << "value for counter3: " << counter3.getValue() << "\n";
14
15     counter1.increment();
16     counter2.increment();
17     counter2.increment();
18
19      std::cout << "value for counter1: " << counter1.getValue() << "\n";
20      std::cout << "value for counter2: " << counter2.getValue() << "\n";
21      std::cout << "value for counter3: " << counter3.getValue() << "\n";
22  }
```

(b) **[inline]**
Modify the code written in part (a) so that all of the member functions for the class `Counter` are inline.

(c) **[friends]**
Create a new function `getCounterValue` that does not belong to the `Counter` class. This function takes a single parameter of type `Counter`, and returns the value of the counter specified by this parameter. The `getCounterValue` function is not permitted to call `Counter::getValue` and must directly access the `count_` data member of the `Counter` object instead. Use friends so that `getCounterValue` can access the `count_` data member (which is private). Is this use of friends a good one? Explain.

(d) **[const correctness]**
Test the `Counter` class with the `myTest2` function given below. (To save typing, this function can be downloaded from the course web site in the file `myTest2.cpp`.) If the `Counter` class cannot properly handle this test code, fix this problem (by correcting the class, not by changing the test code). If the code does not compile, you have probably committed one of the cardinal sins of C++ programming.

```
void myTest2()
{
    Counter counter1;
    counter1.setValue(0);
    const Counter& counter2 = counter1;
    std::cout << "value of counter2: " << counter2.getValue() << "\n";
}
```

(e) **[default constructor, destructor, copy constructor, move constructor, copy assignment operator, move assignment operator]**
Add the following public function members to the class:
- a default constructor, which initializes the counter value to zero;
- a destructor;
- a copy constructor;
- a move constructor;
- a copy assignment operator; and
- a move assignment operator.

Again, test the code using the `myTest` function. Do all of the values printed by this test code make sense? What is the initial value of `counter1` printed on line 4 of the function `myTest`?

(f) **[other constructors]**
Add a constructor that takes the initial counter value as a parameter. Modify the `myTest` function so that a new `Counter` object called `counter4` is created using this new constructor.

(g) **[initializer lists]**
If the default constructor does not currently employ an initializer list (to set the counter value to zero), modify the constructor so that an initializer list is used. Also, modify the copy constructor to use an initializer list.

(h) **[operator overloading]**
Overload **`operator`**`<<` so that `Counter` objects can be directly written to an output stream. Writing a `Counter` object to a stream should simply output the counter value. Do not add any friends to the `Counter` class in your solution. The **`operator`**`<<` function should take a reference to a `std::ostream` object and a **`const`** reference to a `Counter` object as its first and second parameters, respectively; and a reference to the `std::ostream` object should be returned. Test your new stream inserter with the `myTest3` function below. (To save typing, this function can be downloaded from the course web site in the file `myTest3.cpp`.)

```
void myTest3()
{
    Counter counter;
```

```
        std::cout << counter << "\n";
        counter.increment();
        std::cout << counter << "\n";
    }
```

(i) **[static members]**
Add a function member, called `getNumberOfCounters`, to the `Counter` class that returns how many counters are currently in existence (i.e., have been created but not yet destroyed). The tracking of how many `Counter` objects are currently in existence must be maintained internally by the `Counter` class itself, not by other code. This includes any data needed to implement the preceding functionality. Do not use any global variables in your solution. Use the function `myTest4` shown below to test your code, after having replaced the comments in the function with the appropriate code. (To save typing, this function can be downloaded from the course web site in the file `myTest4.cpp`.)

```
void myTest4()
{
    {

        // Print the number of counters in existence here.
        // This value should be 0.

        Counter counter1;
        // Print the number of counters in existence here.
        // This value should be 1.

        Counter counter2 = counter1;
        // Print the number of counters in existence here.
        // This value should be 2.

    }

    // Print the number of counters in existence here.
    // This value should be 0.
}
```

(j) **[type members]**
Add a public type member called `Value` as a synonym for **`long long`**. Change the class so that the type `Value` is used to represent the counter value (instead of **`int`**). Modify the entire class as well as any code using the class so that `Value` is used as the counter value type instead of **`int`**. Ensure that all of your code still compiles and functions properly after the preceding changes are made.