Part C – Encapsulation

Operator Overload

Workshop 5

In this workshop, you will write codes to practice operator overloading to implement Ships and Engine classes.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities to

- code a compositional relationship across multiple classes
- manage dynamic memory within a class
- overload an operator as a member of a class
- overload an operator as a helper function of a class
- reflect upon what you have learned

SUBMISSION POLICY

The workshop is divided into 2 sections;

lab - 50% of the total mark

To be completed before the end of the lab period and submitted from the lab.

DIY - 50% of the total mark

To be completed within **5 days** after the day of your lab (meaning it will be **due 2 days prior to following lab period for the next Workshop**).

The *lab* section is to be completed after the workshop is published, and before the end of the lab session. The *lab* is to be submitted during the workshop period from the lab.

If you attend the lab period and cannot complete the *lab* portion of the workshop during that period, ask your instructor for permission to complete the *lab* portion

after the period. You must be present at the lab in order to get credit for the *lab* portion.

If you do not attend the workshop, you can submit the *lab* section along with your *DIY* section (see Submission Penalties below). The *DIY* portion of the lab is due on the day that is **5 days** after your scheduled *lab* workshop (by 23:59 or 11:59PM) (even if that day is a holiday).

The *DIY* (Do It Yourself) section of the workshop is a task that utilizes the concepts you have done in the **lab** section. This section is open ended with no detailed instructions other than the required outcome.

All your work (all the files you create or modify) must contain your **name, Seneca email and student number**.

You are responsible to back up your work regularly.

Ask your professor if there are any additional requirements for your specific section.

CITATION AND SOURCES

When submitting Workshops, Project and assignment deliverables, a file called **sources.txt** must be present. This file will be submitted with your work automatically.

You are to write either of the following statements in the file "**sources.txt**":

I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.

Then add your name and your student number as signature

OR:

Write exactly which part of the code of the workshops or the assignment are given to you as help and who gave it to you or which source you received it from.

You need to mention the workshop name or assignment name and also mention the file name and the parts in which you received the code for help.

Finally add your name and student number as signature.

By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrong doing.

SUBMISSION PENALTIES:

The following are penalties associated with the workshop if the submission policies above regarding due dates and other requirements are not met. It is advised to pay close attention to these penalties in order to avoid them.

- If the **lab** is submitted past the due date but before the rejection date, it will be **worth half of its total weight** for the Workshop. If a lab is worth 50% then it will be worth 25%.
- If the **diy** is submitted past the due date but before the rejection date, it will be **worth half of its total weight** for the Workshop. If the diy is worth 50% then it will be worth 25%.
- If a submitted **reflection** (reflect.txt) is deemed to be insufficient in depth or does not demonstrate a strong understanding of the core concepts used in the Workshop, a **potential maximum reduction of 30%** may be applied the overall mark of the Workshop.
- If any of lab, diy or reflection is missing the total mark of the workshop will be **zero**.

WORKSHOP DUE DATES

You can see the exact due dates of all assignments by adding -due after the submission command:

Run the following script from your account (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 244/NXX/WS05/lab -due<ENTER>
~profname.proflastname/submit 244/NXX/WS05/DIY -due<ENTER>
```

COMPILING AND TESTING YOUR PROGRAM

All your code should be compiled using this command on matrix:

g++ -Wall -std=c++11 -o ws (followed by your .cpp files)
After compiling and testing your code, run your program as follows to check for possible memory leaks: (assuming your executable name is "ws") **valgrind ws**

LAB - 50%

Engine Class

Design and code a class named `Engine` that holds information about a type of simulated vehicle's engine. Place your class definition in a header file named **`Engine.h`** and your function definitions in an implementation file named **`Engine.cpp`**. Include in your solution all of the statements necessary for your code to compile under a standard C++ compiler and within the **`sdds`** namespace.

Upon instantiation, an `Engine` object may receive no parameters and goes to a safe empty state or the following two parameters:

- A double value, representing the engine size.
- A null-terminated C-style string holding the type of engine. You may assume that the string is no larger than 30 characters, excluding the null terminator.

If the size provided is positive, and the type is not empty, the values are accepted. Otherwise the object adopts ***a safe empty state***.

Your design includes the following member functions:

- `double get() const`- a query that returns the size of the engine.
- `void display() const` - a query that displays the size as well as the type of the engine.

Ship Class

Design and code a class named `Ship` that holds information about a ship used in a simulator. Place your class definition in a header file named **`Ship.h`** and your function definitions in an implementation file named **`Ship.cpp`**. Include in your solution all of the statements necessary for your code to compile under a standard C++ compiler and within the **`sdds`** namespace. The `Ship` class contains an array of `Engine` objects of fixed size 10, a character array for the ship *type* of fixed size 7, an integer storing the *number* of engines (<=10) currently defined for the ship, and a floating point number representing the *distance* a ship can travel.

Upon instantiation, a `Ship` object may receive no parameters **or** the following three parameters.

- A null-terminated C-style string holding the type of ship. This string is always of length six, not including the null terminator.
- An array of engines, representing the ship engines used by the given ship (note large ships typically have many engines).
- The length of the `Engine` array passed in.

If the object receives a non-null string for the ship type of 6 characters or less, and an array of minimum length one for the engines, the object accepts the string as the ship's type, and sets up the array. Otherwise, the object assumes a ***safe empty state***.

Your design also includes the following member functions and helper operators:

- `bool empty() const` - a query that returns true if the object is in a safe empty state; false otherwise.
- `float calculatePower() const` - a query that returns the ship output power as a function of the number of engines and capacity of each engine. The output power for each engine is (engine size X 5). The ship output power is the sum of output power of each engine on the ship.
- `void display() const` - a query that displays the ship type as well as the total ship power to two (2) decimal places in a field of six (6), as shown in the example below, as well as displaying the individual engine types and capacities used in the evaluation. If the current object is empty, this function displays "No available data".
- `Ship& operator+=(Engine e)-` a member operator that adds an engine to the array of engines. Please make sure the number of engines is less than the maximum number of engines allowed. Also, check if the ship has a valid type, if not, display the following message: "The ship doesn't have a type! Engine cannot be added!"
- `bool operator==(const Ship&, const Ship&)-` a friend helper operator that returns true if two ships have the exact same power, false otherwise.
- `bool operator<(const Ship&, double)`-a non-friend helper operator that compares the total output power of the left hand operand against the right hand operand. This function returns true if the total power of the ship is below the value provided, false otherwise.

LAB MAIN MODULE

```
// OOP244 Workshop 5: Operator Overloading
// File: mainTester.cpp
// Version: 1.0
// Date: 2/2/2020
// Author: Elnaz Delpisheh
// Description:
// This file tests lab section of your workshp
//////////////////////////////////////////////

#include <iostream>
#include "Ship.h"
#include "Engine.h"

using namespace std;
using namespace sdds;
int const MIN=90;


int main()
{

    Engine engines[3]=
                    {Engine("V8",6.0),
                    Engine("V8",8.0),
                    Engine("Inline",4.2)};
    Engine engines2[4]=
                    {Engine("D3",7.0),
                    Engine("D0",2.0),
                    Engine("D1",3.2)};

    Ship titanic("liner",engines,3);

    Ship rainbow;

    cout << "Testing Ship objects" << endl ;
    titanic.display();
    cout << endl;
    rainbow.display();
    cout << endl;
    Ship aurora("apollo",engines2,3);
```

```
    aurora.display();

    cout << endl<< endl;

    //Adding an engine to the ship
    cout << "Adding an engine to Titanic" << endl;

    Engine engine = Engine("V10",3.0);
    titanic += engine;
    titanic.display();

    cout << endl<< endl;

    cout << "Adding an engine to Rainbow" << endl;

    Engine engine2 = Engine("V20",40.0);
    rainbow += engine2;
    rainbow.display();

    cout << endl<< endl;

    cout << "Adding an engine to Aurora" << endl;

    aurora += engine2;
    aurora.display();

    cout << endl<< endl;

    //Comparing with the standards:
    cout<<"Comparing Titanic with the standards:"<<endl;
    if(titanic<MIN)
     cout<<"Below average!"<<endl;
    else
     cout<<"Above average!"<<endl;
    cout << endl<< endl;

    cout<<"Comparing the power of Titanic and Aurora:"<<endl;

    if(titanic == aurora)
        cout<<"The ships have the same power!"<<endl;
    else
        cout<<"The ships have different power!"<<endl;

    return 0;

}
```

```
/*
 Testing Ship objects
 liner- 91.00
 6.00 liters - V8
 8.00 liters - V8
 4.20 liters - Inline

 No available data

 apollo- 61.00
 7.00 liters - D3
 2.00 liters - D0
 3.20 liters - D1


 Adding an engine to Titanic
 liner-106.00
 6.00 liters - V8
 8.00 liters - V8
 4.20 liters - Inline
 3.00 liters - V10


 Adding an engine to Rainbow
 The ship doesn't have a type! Engine cannot be added!
 No available data


 Adding an engine to Aurora
 apollo-261.00
 7.00 liters - D3
 2.00 liters - D0
 3.20 liters - D1
 40.00 liters - V20


 Comparing Titanic with the standards:
 Above average!


 Comparing the power of Titanic and Aurora:
 The ships have different power!

 */
```

To complete your coding

a) Include in each file an appropriate header comments uniquely identify the file (include your name, section, id and date of file was last modified)

b) Make sure your function prototypes have meaningful argument names to help understand what the function does.

LAB SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `Ship module` `Engine module` and the `ShipMain.cpp` program to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

~profname.proflastname/submit 244/NXX/WS05/lab<ENTER>

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

DIY (DO IT YOURSELF) – 50%

Part 1: In class Ship, assume there is no limit for the array of engines, representing the engines in the ship.

If the object receives a non-null string for the ship type of 6 characters or less and a positive value for the size of the array, the object accepts the string as the engine type, and sets up the array. Otherwise, the object assumes a ***safe empty state***.

Remember for the operator +=, you will need to

- *Dynamically create a new array of desired size (Size should be one more than the old array). (This step will require a second pointer for temporary use).*
- *Copy the old array's contents into the new one.*
- *Copy the object of type engine to the array.*
- *Deallocate the memory of the old array (avoid memory leak).*
- *Adjust pointers so that the new array has the desired name.*

Part 2: In class Ship, assume there is no limit for the array representing the type of the ship (memory should be dynamically allocated).

DIY MAIN MODULE

```
// OOP244 Workshop 5: Operator Overloading
// File: mainTester.cpp
// Version: 1.0
// Date: 2/2/2020
// Author: Elnaz Delpisheh
// Description:
// This file tests lab section of your workshp
/////////////////////////////////////////////



#include <iostream>
#include "Ship.h"
#include "Engine.h"

using namespace std;
using namespace sdds;
int const MIN=90;


int main()
{

    Engine engines[3]=
    {Engine("V8",6.0),
        Engine("V8",8.0),
        Engine("Inline",4.2)};
    Engine engines2[4]=
    {Engine("D3",7.0),
```

```
    Engine("D0",2.0),
    Engine("D1",3.2)};

Ship titanic("liner",engines,3);

Ship rainbow;

cout << "Testing Ship objects" << endl ;
titanic.display();
cout << endl;
rainbow.display();
cout << endl;
Ship aurora("apollo",engines2,3);

aurora.display();

cout << endl<< endl;

//Adding an engine to the ship
cout << "Adding an engine to Titanic" << endl;

Engine engine = Engine("V10",3.0);
titanic += engine;
titanic.display();

cout << endl<< endl;

cout << "Adding an engine to Rainbow" << endl;

Engine engine2 = Engine("V20",40.0);
rainbow += engine2;
rainbow.display();

cout << endl<< endl;

cout << "Adding an engine to Aurora" << endl;

aurora += engine2;
aurora.display();

cout << endl<< endl;

//Comparing with the standards:
cout<<"Comparing Titanic with the standards:"<<endl;
if(titanic<MIN)
    cout<<"Below average!"<<endl;
```

```
    else
        cout<<"Above average!"<<endl;
    cout << endl<< endl;

    cout<<"Comparing the power of Titanic and Aurora:"<<endl;

    if(titanic == aurora)
        cout<<"The ships have the same power!"<<endl;
    else
        cout<<"The ships have different power!"<<endl;

    return 0;

}
/*
Testing Ship objects
liner- 91.00
6.00 liters - V8
8.00 liters - V8
4.20 liters - Inline

No available data

apollo- 61.00
7.00 liters - D3
2.00 liters - D0
3.20 liters - D1


Adding an engine to Titanic
liner-106.00
6.00 liters - V8
8.00 liters - V8
4.20 liters - Inline
3.00 liters - V10


Adding an engine to Rainbow
The ship doesn't have a type! Engine cannot be added!
No available data


Adding an engine to Aurora
apollo-261.00
7.00 liters - D3
2.00 liters - D0
```

```
3.20 liters - D1
40.00 liters - V20


Comparing Titanic with the standards:
Above average!


Comparing the power of Titanic and Aurora:
The ships have different power!
*/
```

REFLECTION

Study your final solutions for each deliverable of the Workshop, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time and the result is suggested to be at least 150 words in length.**

Create a file named `reflect.txt` that contains your **detailed description of the topics that you have learned** in completing this workshop and mention any issues that caused you difficulty and how you solved them. Add any other comments you wish to make. Submit this file along with your "DIY" submission.

DIY SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload the updated `Ship` module to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

~profname.proflastname/submit 244/NXX/WS05/diy<ENTER>

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.