Part C – Encapsulation

Member Operator Overloads

Workshop 5 (1.1 – Added Pseudo Code for at_home)

In this workshop, you will work with a class that incorporates operator overloads to describe behavior in terms of arithmetic operations. In addition, you will also work with helper functions that assist the class. The classes used in this workshop will be the Parts that make up a boxing Robot.

LEARNING OUTCOMES

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

- To overload an operator as a member function of a class type
- To access the current object from within a member function
- To identify the lifetime of an object, including a temporary object
- To describe to your instructor what you have learned in completing this workshop
- DIY – Able to resize dynamics arrays

SUBMISSION POLICY

The workshop is divided into 3 sections;
in-lab - 30% of the total mark
To be completed before the end of the lab period and submitted from the lab.
at-home - 35% of the total mark
To be completed within 2 days after the day of your lab.
DIY (Do It yourself) – 35% of the total mark
To be completed within 3 days after the at-home due date.
The *in-lab* section is to be completed after the workshop is published, and before the end of the lab session. The *in-lab* is to be submitted during the workshop period form the lab.

If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. You must be present at the lab in order to get credit for the *in-lab* portion.
If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below). The *at-home* portion of the lab is due on the day that is 2 days after your scheduled *in-lab* workshop (23:59) (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 in-lab + at-home section. This section is completely open ended with no detailed instructions other than the required outcome. You must complete the DIY section up to 3 days after the at-home section.
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 the DIY part of the workshop, 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 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.

Late Submission Penalties:

-*In-lab* portion submitted late, with *at-home* portion:
    0 for *in-lab*. Maximum of **DIY+at-home**/10 for the workshop.
-at-home or DIY submitted late:
    1 to 2 days, -20%, 3 to 7 days -50% after that submission rejected.
-If any of *the at-home* or in-lab portions is missing, the mark for the whole workshop will be **0/10**
-If DIY portion is missing you will lose the mark for the DIY portion of the workshop.

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/in_lab -due<ENTER>
~profname.proflastname/submit 244/NXX/WS05/at_home -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 <ENTER>
```

In-Lab (30%)

Parts Module

The **Parts** module will contain classes that represent different parts of a boxing Robot. In our context we will keep number of parts minimum but may be subject to growth in the future. The first and only part for the **in_lab** portion is the **Arms**. As such we will be creating an **Arms** class in the **Parts** module.

To accomplish the above follow these guidelines:

Design and code a class named `Arms`.

Follow the usual rules for creating a module: (i.e. Compilation safeguards for the header file, namespaces and coding styles your professor asked you to follow).

Create a class called **Arms**.

Create the following constants in the **Parts** header:

`MIN_FORCE` with a value of 50.
This constant number is used to set the minimum force for the Arms.

PRIVATE MEMBERS:

A set of arms is composed of a single value:

- `force`
  This is an **integer** representing the amount of punch power in the arms.

PUBLIC MEMBERS:

Constructors/Destructors

Arms objects will be making use of **constructors** to handle their creation. There are two **constructors** that are required:

1. Default constructor – This constructor should set an Arms object to **a safe empty state**. The notion of what the safe empty state will be left up to your design but choose reasonable values.
2. 1 Argument Constructor – This constructor will take in 1 parameter:
   a. An **integer** representing the arms' force

   A little bit of validation needs to occur in the 1 arg constructor. **If the parameter's value is less than 1, set the respective force to be the value of**

MIN_FORCE. Otherwise set the data member to be the value of the parameter as they are.

Other Members

```
int getForce() const;
```
This is a query function that returns the force of the Arms.

Operator Overload Members

As a bit of pretext for operator overloads, consider the use of existing operators to define the overloads for the Arms class.

```
Arms& operator++();
```
This operator overload provides behavior for the Arms objects that maps to the normal notion of a **prefix** increment. The arms' force will be **incremented by 10**. Then **a reference to the current Arms object** is returned.

```
Arms operator++(int);
```
This operator overload provides behavior for the Arms objects that maps to the normal notion of a **postfix** increment. A **copy** of the Arms current object will be created. The arms' force of the current object will be **incremented by 10**. Then lastly the **copy** created earlier will be returned.

```
Arms& operator+=(int);
```

This operator overload provides behavior for the Arms objects that maps to the normal notion of a **plus equals** (assignment). The current object will increase the force on the arms based on the passed in **integer** parameter. **If the result of the operation causes the arms' force to be less than 1, set the value to 0 instead.**

After this operation, the **current object is then returned**.

Robot Module

The boxing **Robot** module will be composed partly of the classes from the earlier **Parts** module. In this **in_lab** that means this **Robot** has **Arms**. As such include the necessary headers in the Robot module.

Create the following constants in the **Robot** header:

`NICK_MAX_LEN` with a value of 10.
This constant number is used to set the maximum left of the nickname data member.

Private members:

A Robot is will have the following data members:

- `nickname`
  This is a **static character array** representing the nickname of the Robot. Utilize the NICK_MAX_LEN to set the size of this array. Remember to consider the **nullbyte**.
- `durability`
  This is an **integer** representing the durability of the Robot
- `arms`
  This is an instance of the **Arms** class representing the arms of the Robot

Public Members:

Constructors/Destructors

Robot objects will be making use of **constructors** as well to handle their creation. There are two **constructors** that are required:

1. Default constructor – This constructor should set a Robot object to **a safe empty state**. The notion of what the safe empty state will be left up to your design but choose reasonable values.
2. 3 Argument Constructor – This constructor will take in 3 parameters:

a. A **constant character pointer** that represents the nickname of the Robot
b. An **integer** representing the arms' force
c. An **integer** representing the durability of the Robot

A little bit of validation needs to occur in the 3 arg constructor. **If the provided constant character pointer** for the nickname is either nullptr or an empty string then set the Robot to **a safe empty state**.

Otherwise set the data members to be the values of the parameters as they are. If the provided string from the parameter is greater than **NICK_MAX_LEN** then simply copy over **NICK_MAX_LEN** characters. Remember to close off the string with a **nullbyte** at the last index.

If the durability value provided from the parameter is less than 1 then set the data member **durability to 100**.

Setting the value of the **arms** should incorporate the use of **assigning the Arms 1 arg constructor (with a value passed in)** to the respective data member.

Other Members

```
ostream& display() const;
```
This member function will display the current details of the Robot. If the Robot is in an **empty** state the following will be printed:

```
"This Robot isn’t operational" <newline>
```

If the Robot isn't empty then it will print:

```
"***Robot Summary***" <newline>
"Nickname: [nickname]" <newline>
"Arm Power [arms force]" <newline>
"Durability: [durability]" <newline>
```

Refer to the sample output for details.

Lastly at the end of the function **return the cout object**.

Operator Overload Members

`operator bool() const;`

This **boolean conversion** operator will define the behavior of converting a Robot into a bool value. For our context, we will use the notion of if the nickname or the durability is in an **empty** state then we will return **true**. Otherwise we return **false**.

IN-LAB MAIN MODULE

```
/**********************************************************************
// OOP244 Workshop 5: Member Operator Overloads IN LAB
// File BouquetTester.cpp
// Version 1.0
// Date       2019/10/05
// Author     Hong Zhan (Michael) Huang
// Description
// Tests the Robot and Arms classes through the use of their public members
// In particular their operators.
//
// Revision History
// -----------------------------------------------------------
// Name              Date              Reason
// Michael
/////////////////////////////////////////////////////////////////
**********************************************************************/

#include <iostream>
#include "Robot.h"
#include "Robot.h"
#include "Parts.h"
#include "Parts.h"

using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
       for (int i = 0; i < len; i++, cout << ch);
       return cout;
}
ostream& number(int num) {
       cout << num;
       for (int i = 0; i < 9; i++) {
              cout << " - " << num;
       }
       return cout;
}

int main() {

  cout << "Testing Empty Pair of Arms (default)" << endl;
       line(64, '-') << endl;
  number(1) << endl;
  Arms a1;
  cout << "a1 arms: " << a1.getForce();

  cout << "\nPrefix ++ on Empty Arms" << endl;
  line(64, '-') << endl;
  number(2) << endl;
  Arms a2 = ++a1;
  cout << "a1 arms: " << a1.getForce() << endl;
  cout << "a2 arms: " << a2.getForce() << endl;;

  cout << "\nPostfix ++ on Non Empty Arms" << endl;
  line(64, '-') << endl;
  number(3) << endl;
  Arms a3 = a1++;
  cout << "a1 arms: " << a1.getForce() << endl;
  cout << "a3 arms: " << a3.getForce() << endl;

  cout << "\nTesting Non Empty Arms (2 arg)" << endl;
  line(64, '-') << endl;
  number(4) << endl;
  Arms a4(0);
  Arms a5(7);
  cout << "a4 arms: " << a4.getForce() << endl;
  cout << "a5 arms: " << a5.getForce() << endl;

  cout << "\nTesting An Empty Robot" << endl;
  line(64, '-') << endl;
  number(5) << endl;
  Robot r1;
  r1.display();

  cout << "\nTesting An Non Empty Robot" << endl;
  line(64, '-') << endl;
  number(6) << endl;
  Robot r2("Mechanick", 0, 0);
  Robot r3("Ippo", 70, 600);
  r2.display() << endl;
  r3.display();

  cout << "\nTesting A Robot to Bool conversion" << endl;
  line(64, '-') << endl;
  number(7) << endl;
  if (r1) cout << "r1 results in a true value" << endl;
  if (!r2) cout << "r2 results in a false value" << endl;

  return 0;
}
```

EXECUTION EXAMPLE RED VALUES ARE USER ENTRY

```
Testing Empty Pair of Arms (default)
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
a1 arms: 0
Prefix ++ on Empty Arms
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
```

```
a1 arms: 10
a2 arms: 10

Postfix ++ on Non Empty Arms
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
a1 arms: 20
a3 arms: 10

Testing Non Empty Arms (2 arg)
----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
a4 arms: 50
a5 arms: 7

Testing An Empty Robot
----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
This Robot isn't operational

Testing An Non Empty Robot
----------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
***Robot Summary***
Nickname: Mechanick
Arm Power: 50
Durability: 100

***Robot Summary***
Nickname: Ippo
Arm Power: 70
Durability: 600

Testing A Robot to Bool conversion
----------------------------------------------------------------
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
r1 results in a true value
r2 results in a false value
```

IN-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 Arms and Robot modules and the RobotTester.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/WS04/in_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.

AT_HOME (35%)

PARTS MODULE (UPDATED)

The **Parts** module will be updated to include a new Part. We will be giving our Robot a pair of Legs so they may have a bit of footwork while boxing.

Create a class called **Legs**.

Create the following constants in the **Parts** header:

`MIN_SPEED` with a value of 25.
This constant number is used to set the minimum speed for the Legs.

PRIVATE MEMBERS:

For the purposes of the boxing Robot, the legs are there to provide a sense of speed of their movement. There will be one data member:

- `speed`
  This is an **integer** representing the amount speed in the Legs.

PUBLIC MEMBERS:

Constructors/Destructors

Legs objects will be making use of **constructors** to handle their creation. There are two **constructors** that are required:

1. Default constructor – This constructor should set a Legs object to **a safe empty state**. The notion of what the safe empty state will be left up to your design but choose reasonable values.
2. 1 Argument Constructor – This constructor will take in 1 parameter:

a. An **integer** representing the Leg's speed

A little bit of validation needs to occur in the 1 arg constructor. **If the provided parameter for the speed is less than 1, set the speed of the legs to be the value of MIN_SPEED**. Otherwise set the data members normally.

Other Members

```
int getSpeed() const;
```

This is a query function that returns the speed of the Legs.

Operator Overload Members (Legs)

As with the in_lab operator overloads, consider the use of existing operators to define the overloads for the Legs class.

```
Legs& operator-=(int);
```

This operator overload provides behavior for the Legs objects that maps to the normal notion of a **minus equals** (assignment). The current object will decrease the speed of the Legs based on the passed in **integer** parameter. **If the result of the operation causes the speed to be less than 1, set their values to 0 instead.**

After this operation, the **current object is then returned.**

Global Helper Operators (Arms, Legs)

The following two helper functions are operator overloads that will assist the Legs class. These will be defined in the **Parts** module **but won't be a part of the Arms or Legs classes.**

```
bool operator>(const Legs&, const Legs&);
```

This function will be used to determine whether one pair of Legs is quicker than another. It overloads the notion of the **greater than operator** but for use with **Legs**. If the first set of Legs is the nimbler one then this operator will return a Boolean value of **true** and **false** otherwise.

```
bool operator<(const Legs&, const Legs&);
```
This function will evaluate the opposite of the previous operator overload. If the first set of Legs is slower than the second, return **true**. Return **false** otherwise. **Consider the use of existing operators to define this one.**

```
bool operator>(const Arms&, const Arms&);
```
Follow the same idea as the above operators for Legs but this function will compare Arms. If the **force** are **greater** in the first set of Arms than the second return **true**. Return **false** otherwise.

```
bool operator<(const Arms&, const Arms&);
```
This function will evaluate the opposite of the previous operator overload. If the first set of Arms is weaker than the second, return **true**. Return **false** otherwise. **Consider the use of existing operators to define this one.**

ROBOT MODULE (UPDATED)

The **Robot** module will be updated to incorporate the new **Legs** parts.

PRIVATE MEMBERS:

There will be one data member added to the Robot:

- `legs`
  This is an **instance of the Legs class** representing the Legs of the Robot.

PUBLIC MEMBERS:

Constructors

Update the **3 arg constructor** into a **4 arg** one to reflect the addition of a new data member. The order of the parameters are as follows:

a. A **constant character pointer** that represents the nickname of the Robot
b. An **integer** representing the arms' force
c. An **integer** representing the speed of the legs [NEW]
d. An **integer** representing the durability of the Robot

Similar to the original 3 arg constructor, the assignment of the legs should incorporate the use of the **Legs constructor** and its **assignment** to the **legs** data member.

Other Members

```
ostream& display() const;
```
Update the **display** function to incorporate the Legs into the output in the case of a **non-empty** Robot:

```
"***Robot Summary***" <newline>
"Nickname: [nickname]" <newline>
"Arm Power: [arm force]" <newline>
"Durability: [durability]" <newline>
“Leg Speed: [leg speed]” <newline>
```

Add two new query functions:

```
Arms getArms() const;
```
Returns a copy of the Arms of the current Robot object.

```
Legs getLegs() const;
```
Returns a copy of the Legs of the current Robot object.

Operator Overload Members

```
Robot& operator-=(int);
```
This operator overload describes the behavior of the Robot class when it interacts with the **minus equals** operator. The current object will have its durability lowered by the **integer** parameter passed in. If the durability of the Robot drops below **one** then set the durability to **zero**.

Global Helper Function (Robot)

The following helper function will assist the Robot class. It will be defined in the **Robot** module **but won’t be a part of the Robot class.**

`void box(Robot&, Robot&);`

The box function will have two Robots passed in through the parameters face each other is a bout. **It is highly advised to make use of previously defined operators in this function as they may help to condense the code**. The function should work in the following way:

a. First print out a line:
   `"Attempting to begin a Robot boxing match" <newline>`

b. Now check if a bout can even begin. The conditions for a bout being possible is if both Robots are not in an **empty** state (meaning their durability is non-zero and that their nicknames are not nullptr).

   If this isn't the case then simply print out:

   `"At least one of the Robots isn't operational. No bout will be had." <newline>`

   And the function will end there.

c. In the case that both Robots are in the right condition to face one another then proceed with a print out:

   `"Both participants are operational... beginning the bout" <newline>`

d. Now begins the main combat loop. The two Robots will engage in continuous rounds of exchanges until one of the two is no longer **operational**.

   The actual combat itself will work as such:

   The **faster** Robot will get to attack first. An attack will be consisting of having the **force** of one Robot's Arms **reduce the durability** of the recipient of the attack. If this initial strike reduces the durability of the recipient to **0** then the recipient won't be able to fight back (This could be considered a KO). If it doesn't then the recipient will launch its own attack to **reduce the durability** of its opponent. After both attacks have potentially occurred, then check the **status** of each Robot. If **either of the two are no longer operational** then the loop will end.

e. Once the combat loop has ended, we now need to display who won the match.

   Firstly, print out the following:

   ```
   "The bout has concluded... the winner is: " <newline>
   ```

   After this depending on which Robot won, use the Robot **display** function to show the summary of their condition after the fight.

In addition to the above descriptions consider this **pseudo code** for the main portion of the box function:

```
If (Robot1 and Robot2 are operational / non-empty)
        print out the 'beginning the bout' message;

        while (Robot1 and Robot 2 are both operational)
                If (Robot 1 is faster than Robot 2)
                        Robot1 applies arm force on Robot2's durability
                        If (Robot 2 is still operational)
                                Robot2 applies arm force on Robot1's durability
                else (Robot 2 is faster than Robot 1)
                        Robot 2 applies arm force on Robot 1's durability
                        If (Robot1 is still operational)
                                Robot1 applies arm force on Robot2's durability

                check if either of the two Robots have been KO'd.

        while loop ends if the check results in a true value

        print out the winner of the match
else
        print out the "No bout will begin" message
```

Refer to the sample output for how everything should look.

AT-HOME MAIN MODULE

```
/**********************************************************************
// OOP244 Workshop 5: Member Operator Overloads AT HOME
// File RobotTester2.cpp
// Version 1.0
// Date       2019/10/05
// Author     Hong Zhan (Michael) Huang
// Description
// Tests the Robot, Arms, Legs classes through the use of their public members
// In particular their operators and helper functions.
//
// Revision History
```

```cpp
// -----------------------------------------------------------
// Name              Date            Reason
// Michael
/////////////////////////////////////////////////////////////////
*************************************************************************/

#include <iostream>
#include "Robot.h"
#include "Robot.h"
#include "Parts.h"
#include "Parts.h"

using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
       for (int i = 0; i < len; i++, cout << ch);
       return cout;
}
ostream& number(int num) {
       cout << num;
       for (int i = 0; i < 9; i++) {
              cout << " - " << num;
       }
       return cout;
}

int main() {

  cout << "Testing Creating Legs, Empty & Non-Empty" << endl;
       line(64, '-') << endl;
  number(1) << endl;
  Legs l1, l2(-13), l3(50);
  cout << l1.getSpeed() << " " << l2.getSpeed() << " " << l3.getSpeed() << endl;

  cout << "\n -= on Empty & Non-Empty Legs" << endl;
  line(64, '-') << endl;
  number(2) << endl;
  l1 -= 5; l2 -= 10; l3 -= 15;
  cout << l1.getSpeed() << " " << l2.getSpeed() << " " << l3.getSpeed() << endl;

  cout << "\nCompare the speed of Legs" << endl;
  line(64, '-') << endl;
  number(3) << endl;
  if (l1 < l3) cout << "l1 is slower than l3" << endl;
  if (l3 > l2) cout << "l3 is faster than l2" << endl;

  cout << "\nTesting Non-Empty Robot w/ Legs" << endl;
  line(64, '-') << endl;
  number(4) << endl;
  Robot r1("Miyata", 120, 120, 500), r2("Ippo", 150, 100, 600);
  r1.display(); r2.display();

  cout << "\nTesting -= on a Robot's durability" << endl;
  line(64, '-') << endl;
  number(5) << endl;
  Robot r3("Takamura", 200, 200, 1000);
  r3 -= 50;
  r3.display();
  cout << "\nTesting box function (no bout)" << endl;
  line(64, '-') << endl;
  number(6) << endl;
  Robot r4;
  box(r3, r4);

  cout << "\nTesting box function (bout)" << endl;
  line(64, '-') << endl;
  number(7) << endl;
  box(r1, r2);

  return 0;

}
```

Execution Example **Red values are user entry**

```
Testing Creating Legs, Empty & Non-Empty
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
0 25 50

 -= on Empty & Non-Empty Legs
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
0 15 35

Compare the speed of Legs
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
l1 is slower than l3
l3 is faster than l2

Testing Non-Empty Robot w/ Legs
----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
***Robot Summary***
Nickname: Miyata
Arm Power: 120
Durability: 500
Legs: 120
***Robot Summary***
Nickname: Ippo
Arm Power: 150
Durability: 600
Legs: 100

Testing -= on a Robot's durability
----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
***Robot Summary***
Nickname: Takamura
Arm Power: 200
Durability: 950
Legs: 200
```

```
Testing box function (no bout)
----------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
Attempting to being a Robot boxing match
At least one of the Robots isn't operational. No bout will be had.

Testing box function (bout)
----------------------------------------------------------------
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
Attempting to being a Robot boxing match
Both participants are operational... beginning the bout
The bout has concluded... the winner is:
***Robot Summary***
Nickname: Ippo
Arm Power: 150
Durability: 120
Legs: 100
```

AT-HOME SUBMISSION

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

If not on matrix already, upload Parts and Robot modules and the RobotTester2.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/at_home<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 (35%)

RobotSquad Module

The RobotSquad Module will contain a class **RobotSquad**. A RobotSquad consists of the following data:

- A **name** whose length is 10 characters at maximum. The default name if none is provided is "No Name".
- A roster of **Robots** of unknown length. Consider the need for dynamic memory.
- A **count** of how many Robots are in the roster

A RobotSquad should be able to **query** the above information at any point in time.

Remember to consider how the RobotSquad is constructed and if there is any need to clean up resources at the end of its life time.

Assembling a Roster

The main purpose of the RobotSquad is to **assemble a roster** of Robots.

A roster can be affected in two ways, we can add to the roster with a new Robot or we can remove a Robot from the roster.

The addition of a Robot to the roster should be expressed in the following manner:

RobotSquad += Robot;

The amount of memory allocated will need to be increased to allow for a new Robot to enter the roster. Consider how we can resize a dynamic roster.

The removal of a Robot from the Roster should be expressed in the following manner:

--RobotSquad;

When removing a Robot from the roster, we will opt to remove the **newest** Robot on the roster.

The amount of memory allocated will need to be decreased as we no longer require it after removing a Robot. Consider how we can resize a dynamic roster.

Other considerations for the removal:

- If a Squad's members is reduced to zero then the roster should be set to a safe empty state.
- If a Squad doesn't have a roster. Then there is no need to do anything except to print the following message:

```
"This squad **[Squad Name]** has no members or is uninitialized. Can't
do --." <newline>
```

Displaying a Roster

Displaying the details of a roster should be done as follows:

- If the roster is empty then display:
  `"Squad **[Squad Name]** has no members" <newline>`

- Otherwise a full summary of the Squad's roster will be displayed. It will first display the Squad's details such as their name, and roster count then each detail of its roster. Refer to the sample output for what this looks like.

DIY Main Module

```
/***********************************************************************
// OOP244 Workshop 5: Member Operator Overloads DIY
// File RobotSquadTester.cpp
// Version 1.0
// Date      2019/10/05
// Author    Hong Zhan (Michael) Huang
// Description
// Tests the RobotSquad class and its ability to resize a roster of Robots
//
// Revision History
// -----------------------------------------------------------
// Name              Date             Reason
// Michael
/////////////////////////////////////////////////////////////////
***********************************************************************/

#include <iostream>
#include "RobotSquad.h"
#include "RobotSquad.h"

using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
       for (int i = 0; i < len; i++, cout << ch);
       return cout;
}
ostream& number(int num) {
       cout << num;
       for (int i = 0; i < 9; i++) {
              cout << " - " << num;
       }
       return cout;
}

int main() {
  // Robots for creating squads with
  Robot r1("Joe", 200, 300, 700), r2("Ippo", 300, 200, 800), r3("Miyata", 200, 400, 500);
  Robot r4("Takamura", 400, 400, 1000), r5("Ricardo", 400, 400, 800), r6("Robo", 1, 1,
1);

  cout << "Empty Squad, default constr" << endl;
       line(64, '-') << endl;
  number(1) << endl;
  RobotSquad rs1;
  rs1.display();

  cout << "\nAdd Members to Empty Squad" << endl;
  line(64, '-') << endl;
  number(2) << endl;
  rs1 += r1;
  rs1.display();

  cout << "\nSubstract members from an Empty Squad" << endl;
  line(64, '-') << endl;
  number(3) << endl;
  RobotSquad rs2("Gold");
  --rs2;

  cout << "\nSubtract members from a Non-Empty Squad" << endl;
  line(64, '-') << endl;
  number(4) << endl;
  cout << "Before subtraction" << endl;
  line(64, '-') << endl;
  rs1.display();
  --rs1;
  cout << "\nAfter subtraction" << endl;
  line(64, '-') << endl;
  rs1.display();
  cout << "\nFill up Squads" << endl;
  line(64, '-') << endl;
  number(5) << endl;
  rs1 += r1;
  rs1 += r2;
  rs1 += r3;
  rs2 += r4;
  rs2 += r5;
  rs2 += r6;
  rs1.display();
  line(64, '-') << endl;
  rs2.display();

  cout << "\nBox one member from each Squad" << endl;
  line(64, '-') << endl;
  number(6) << endl;
  box(rs1.getRoster()[0], rs2.getRoster()[2]);
  box(rs1.getRoster()[1], rs2.getRoster()[0]);
  box(rs1.getRoster()[2], rs2.getRoster()[1]);

  cout << "\nDisplay the condition of each Squad after boxing" << endl;
  line(64, '-') << endl;
  number(7) << endl;
  rs1.display();
  rs2.display();

  return 0;

}
```

EXECUTION EXAMPLE RED VALUES ARE USER ENTRY

```
Empty Squad, default constr
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
Squad **No Name** has no members

Add Members to Empty Squad
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
***Squad Summary***
Name: No Name
Roster Count: 1
Roster:
***Robot Summary***
Nickname: Joe
Arm Power: 200
Durability: 700
Legs: 300

Substract members from an Empty Squad
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
This squad **Gold** has no members or is uninitialized. Can't do --.

Subtract members from a Non-Empty Squad
----------------------------------------------------------------
```

```
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
Before subtraction
----------------------------------------------------------------
***Squad Summary***
Name: No Name
Roster Count: 1
Roster:
***Robot Summary***
Nickname: Joe
Arm Power: 200
Durability: 700
Legs: 300

After subtraction
----------------------------------------------------------------
Squad **No Name** has no members

Fill up Squads
----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
***Squad Summary***
Name: No Name
Roster Count: 3
Roster:
***Robot Summary***
Nickname: Joe
Arm Power: 200
Durability: 700
Legs: 300
***Robot Summary***
Nickname: Ippo
Arm Power: 300
Durability: 800
Legs: 200
***Robot Summary***
Nickname: Miyata
Arm Power: 200
Durability: 500
Legs: 400
----------------------------------------------------------------
***Squad Summary***
Name: Gold
Roster Count: 3
Roster:
***Robot Summary***
Nickname: Takamura
Arm Power: 400
Durability: 1000
Legs: 400
***Robot Summary***
Nickname: Ricardo
Arm Power: 400
Durability: 800
Legs: 400
***Robot Summary***
Nickname: Robo
Arm Power: 1
Durability: 1
Legs: 1
```

```
Box one member from each Squad
----------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
Attempting to being a Robot boxing match
Both participants are operational... beginning the bout
The bout has concluded... the winner is:
***Robot Summary***
Nickname: Joe
Arm Power: 200
Durability: 700
Legs: 300
Attempting to being a Robot boxing match
Both participants are operational... beginning the bout
The bout has concluded... the winner is:
***Robot Summary***
Nickname: Takamura
Arm Power: 400
Durability: 700
Legs: 400
Attempting to being a Robot boxing match
Both participants are operational... beginning the bout
The bout has concluded... the winner is:
***Robot Summary***
Nickname: Ricardo
Arm Power: 400
Durability: 600
Legs: 400

Display the condition of each Squad after boxing
----------------------------------------------------------------
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
***Squad Summary***
Name: No Name
Roster Count: 3
Roster:
***Robot Summary***
Nickname: Joe
Arm Power: 200
Durability: 700
Legs: 300
This Robot isn't operational
This Robot isn't operational
***Squad Summary***
Name: Gold
Roster Count: 3
Roster:
***Robot Summary***
Nickname: Takamura
Arm Power: 400
Durability: 700
Legs: 400
***Robot Summary***
Nickname: Ricardo
Arm Power: 400
Durability: 600
Legs: 400
This Robot isn't operational
```

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 Robot, Parts and RobotSquad modules and the RobotSquadTester.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/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.