Part D – Inheritance

Derived Classes & Custom I/O Operators

Workshop 7 (1.1 at_home)

In this workshop, you will work with classes that make up a hierarchal structure. The base or parent class in this case will be an ore that holds details that is prevalent to unrefined metals. This parent will then be derived into a few child classes that deepen the specification. In addition to this hierarchy, we will be incorporating custom input/output operators for these classes.

Learning Outcomes

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

- Create a base class from which others can derive from
- Create derived classes that expand upon a base
- Utilize custom input/output operators with these classes
- To describe to your instructor what you have learned in completing this workshop

Submission Policy

The workshop is divided into 2 sections;
in-lab - 50% of the total mark
To be completed before the end of the lab period and submitted from the lab.
at-home - 50% of the total mark
To be completed within 2 days after the day of your lab.

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).
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 Home 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 **at-home**/10 for the workshop.
-at-home 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**

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/WS07/in_lab -due<ENTER>
~profname.proflastname/submit 244/NXX/WS07/at_home -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 (50%)

ORE MODULE

The **Ore** module is a class that represent a generic ore.

To accomplish the above follow these guidelines:

Design and code a class named `Ore`.

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 the following constants in the **Ore** header:

`DEFAULT_WEIGHT` with a value of 300.50
This constant **double** number is used to set the default weight of the Ore.

`DEFAULT_PURITY` with a value of 20.
This constant **integer** number is used to set the default purity of an Ore.

`CLASS_LEN` with a value of 30.
This constant **integer** number is used to set the max length of the classification of an Ore.

PRIVATE MEMBERS:

An Ore will have the following data members:

- `weight`
  This is a **double value** representing the weight of the Ore.
- `purity`
  This is an **integer value** representing the purity of the Ore.
- `classification`
  This is a **char array** with a max length of representing the purity of the Ore. Recall the need for a **null byte**.

PUBLIC MEMBERS:

Constructors/Destructors

Ore 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 Ore 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 **double value** that represents the weight of the Ore
   b. An **integer value** representing the purity of the Ore
   c. A **const char pointer** that represents the classification of the Ore. A default value of "**Unknown**" should be used.

For each of the data members they should be validated like so:

- If the provided weight parameter is less than 1 set the **weight** of the Ore to the DEFAULT_WEIGHT. Otherwise set it as is.
- If the provided purity parameter is less than 1 or greater than 100, set the **purity** of the Ore to the DEFAULT_PURITY. Otherwise set it as is.
- The **classification** of the Ore should only copy the provided string value up to CLASS_LEN number of characters and no more.

Other Members

```
bool refine();
```

This member function refines the Ore. The meaning of refinement is the reduction of weight to increase purity. In this case the refinement process is done by **reducing the weight of the Ore by 20**. This then **increases the purity by 10**. The purity of an Ore cannot exceed 100 and thus if a refine call causes the purity to go over that, **set the purity to 100 instead**.

If an Ore is already at 100 purity then instead of performing the above, print out the following:

```
"Can no longer be refined" <newline>
```

If the refinement was successful, return **true**. Otherwise return **false**.

```
ostream& display(ostream& os) const;
```

This member function will display the current details of the Ore. Note that however this display function takes in a **parameter** of **ostream** type. This public member function will play a supporting role for our **I/O operator overloads** later on. As such when printing out the below output, *instead of directing the text to cout*, **direct it to the os parameter instead**. Consider **referencing previous examples of custom I/O** if difficulties arise.

If the Ore is in an **empty** state the following will be printed:

```
"This ore is imaginary" <newline>
```

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

```
"Weight: [weight]" <newline>
"Purity: [purity]" <newline>
"Classification: [classification]" <newline>
```

The weight should have two decimal point precision.

Refer to the sample output for details.

Lastly at the end of the function **return the parameter os**.

```
istream& input(istream& is);
```

The member function facilitates setting the values of an Ore through the use of user input. Rather than using cin specifically however, we will be making use of a more generic **istream** parameter **is**. Similar to the display function prior, instead of utilizing cin to take input from, use **is** instead. **Review the materials on custom input operators** for reference if difficulties arise.

The order in which input is supplied is as follows:

1. Create a temporary **double variable** to store a **weight**.
2. Print out the following:

   ```
   "Enter a value for this ore's weight: " <newline>
   ```

3. Accept user input from the **is** parameter into this temporary **double variable**.
4. Create a temporary **integer variable** to store a **purity**.
5. Print out the following:

   ```
   "Enter a value for this ore's purity: " <newline>
   ```

6. Accept user input from the **is** parameter into this temporary **integer variable**.
7. Create a temporary **char array variable** that can store a **classification** (consider the length needed).
   Print out the following:

   ```
   "Enter a classification for the ore (MAX 30 chars): "
   <newline>
   ```

8. Clear the buffer for the **is** parameter by using the **ignore**() function (similar to cin) then accept user input from the **is** parameter into this temporary **character array variable**. It may be advised to utilize the **getline**() functions (similar to cin) to extract a specific number of characters.
9. Utilize these three variables to then set the current **Ore** object. Recall that values like weight and purity had some validation that was applied in the **3-arg** constructor. **Keep in line with those guidelines here**. Consider the use of the 3-arg constructor here with temporary objects to set the current Ore.

Operator Overload Helpers

```
ostream& operator<<(ostream& os, const Ore& ore);
```
This operator overload allows for the use of language such as:

```
cout << ore1;
```

Where ore1 is an object of Ore type. This overload will perform the following:

1. First it prints out: "Ore" <newline>
2. Then it makes a call to the Ore class's **display** function through the **ore** parameter.
3. Lastly it returns the **os** parameter.

```
istream& operator>>(istream& os, Ore& ore);
```
This operator overload allows for the use of language such as:

```
cin >> ore1;
```

Where ore1 is an object of Ore type. This overload will perform the following:

1. It makes a call to the Ore class's **input** function through the **ore** parameter.
2. Lastly it returns the **is** parameter.

Recall that these operator overloads don't work specifically with cin or cout but more generically with ostream and istream objects.

IN-LAB MAIN MODULE

```
/*****************************************************************
// OOP244 Workshop 7: Dervied Classes & Custom I/O Operators
// File OreTester.cpp
// Version 1.0
// Date       2019/10/23
// Author     Hong Zhan (Michael) Huang
// Description
// Tests the Ore class and its use of custom I/O operators
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
// Michael
/////////////////////////////////////////////////////////////////
*****************************************************************/

#include <iostream>
#include "Ore.h"
#include "Ore.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 << "Ore default constr" << endl;
  line(64, '-') << endl;
  number(1) << endl;
  Ore o1;
  o1.display(cout) << endl;

  cout << "Ore 3 arg constr invalid args" << endl;
  line(64, '-') << endl;
  number(2) << endl;
  Ore o2(100, -100);
  Ore o3(10, 600);
  Ore o4(-100, 60);
  o2.display(cout) << endl;
  o3.display(cout) << endl;
  o4.display(cout) << endl;

  cout << "Ore 3 arg constr valid" << endl;
  line(64, '-') << endl;
  number(3) << endl;
```

```
  Ore o5(3000, 10, "Sedimentary");
  o5.display(cout) << endl;

  cout << "Ore refinement w/ custom output operator" << endl;
  line(64, '-') << endl;
  number(4) << endl;
  o5.refine();
  cout << o5 << endl;

  cout << "Ore maxed refinement w/ custom output operator" << endl;
  line(64, '-') << endl;
  number(5) << endl;
  Ore o6(600, 80, "Volcanic");
  o6.refine();
  o6.refine();
  o6.refine();
  cout << o6 << endl;

  cout << "Ore empty -> custom input operator" << endl;
  line(64, '-') << endl;
  number(6) << endl;
  Ore o7;
  cin >> o7;
  cout << endl << o7 << endl;

  cout << "Ore non-empty -> custom input operator" << endl;
  line(64, '-') << endl;
  number(7) << endl;
  cin >> o7;
  cout << endl << o7;

  return 0;

}
```

EXECUTION EXAMPLE RED VALUES ARE USER ENTRY

```
Ore default constr
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
This ore is imaginary

Ore 3 arg constr invalid args
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
Weight: 100.00
Purity: 20
Classification: Unknown

Weight: 10.00
Purity: 20
Classification: Unknown

Weight: 300.50
Purity: 60
Classification: Unknown
```

```
Ore 3 arg constr valid
-----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
Weight: 3000.00
Purity: 10
Classification: Sedimentary

Ore refinement w/ custom output operator
-----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
Ore
Weight: 2980.00
Purity: 20
Classification: Sedimentary

Ore maxed refinement w/ custom output operator
-----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
Can no longer be refined
Ore
Weight: 560.00
Purity: 100
Classification: Volcanic

Ore empty -> custom input operator
-----------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
Enter a value for this ore's weight: 900.99
Enter a value for this ore's purity: 2000
Enter a classification for the ore (MAX 30 chars): Magmatic

Ore
Weight: 900.99
Purity: 20
Classification: Magmatic

Ore non-empty -> custom input operator
-----------------------------------------------------------------
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
Enter a value for this ore's weight: 200.22
Enter a value for this ore's purity: 80
Enter a classification for the ore (MAX 30 chars): Granite

Ore
Weight: 200.22
Purity: 80
Classification: Granite
```

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 Ore module and the OreTester.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/WS07/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 (50%)

METAL MODULE

The **Metal** module will be a child of Ore. It will include some new data members and functions on top of what was established in Ore.

Create a class called **Metal** that derives from **Ore**.

In the **Metal** header include the following constant values:

DEFAULT_MOHS with a value of 2.5
This constant **double** number is used to set the default mohs of a Metal. Mohs is a scale of hardness for metals.

NAME_LEN with a value of 10
This constant **integer** number is used to determine the max length of the name for a metal.

PRIVATE MEMBERS:

Add the following data members to the Metal class:

- `name`
  This is a **character pointer** used to represent the name of the Metal.
- `mohs`
  This is a **double value** used to represent the mohs of the Metal.

PUBLIC MEMBERS:

Constructors/Destructors

1. Default constructor – This constructor should set a Metal 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. Only the **data members new** to Metal will need to be set. Rely on the **base class's default constructor** to set the data members inherited from **Ore**.
2. 5 Argument Constructor – This constructor will take in 5 parameters:
   a. A **double value** that represents the weight of the Metal
   b. An **integer value** representing the purity of the Metal
   c. A **const char pointer** that represents the classification of the Metal.
   d. A **const char pointer** that represents the name of the Metal.
   e. A **double value** that represents the mohs of the Metal.

   A little bit of validation is needed here. If the name of the Metal that is passed in is either **nullptr** or an **empty string** set the object to an empty state. Otherwise set the values normally.

   Note that the **name** is a pointer so utilizing **dynamic** memory there. The maximum length of a name is determined by NAME_LEN. **Allocate as much memory is needed for the name and no more.**

   If the **mohs** value passed in is **greater than zero** then set it as is. Otherwise use the DEFAULT_MOHS constant instead.

   Utilize the 3-arg constructor from Ore when coding this 5-arg constructor.

As there is a dynamic resource in the Metal class, deallocate that memory appropriately as needed.

Other Members

```
void refine();
```
This member function refines the Metal. It relies on the Ore **refine**() function. Attempt to call the Ore **refine**() first and if that operation returned **true**, increment the Metal's **mohs** value by 1. Otherwise do nothing.

```
ostream& display(ostream& os) const;
```
This member function will display the current details of the Metal. It will work similarly to the **display** function in Ore. It will also call the **display** function from Ore. Firstly it checks if the Metal is in an empty state.

If the Metal is in an **empty** state the following will be printed:

```
"This metal is imaginary" <newline>
```

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

```
“Name: [name]” <newline>
"Weight: [weight]" <newline>
"Purity: [purity]" <newline>
"Classification: [classification]" <newline>
“Mohs: [mohs]” <newline>
```

Notice that the lines in blue are elements new to Metal while the red are from Ore. Consider how you can call Ore's display to make this occur.

Refer to the sample output for details.

Lastly at the end of the function **return the parameter os**.

```
istream& input(istream& is);
```
The member function facilitates setting the values of a Metal through the use of user input. It goes through a process very similar to that of the Ore **input** function.

The order in which input is supplied is as follows:

1. Create a temporary **character array** to store a **name**. Use NAME_LEN to determine the size.
2. Print out the following:

```
"Enter a value for this metal's name: " <newline>
```

3. Accept user input from the **is** parameter into this temporary variable. Consider the use of the **getline**() functions as we are grabbing a string of a certain length.
4. Remembering that the name data member is pointer, **if the name already has memory allocated** to it, **deallocate** it.
5. Allocate new **memory** for the **name** as needed and then copy the value from the temporary variable into the name. Consider the **strcpy** functions and the position of the **nullbyte**.
6. Create a temporary **double** variable to store the **mohs** value.
7. Print out the following:

```
"Enter a value for this metal's mohs: " <newline>
```

8. Prior to accepting new input from the user, make sure to **clear the buffer** of any potential remnants. Consider the use of the **ignore**() and **clear**() functions that are normally applied to **cin** but here we will apply then to our **is** parameter. **Make sure to use the clear() function before ignore().**
9. Accept user input from the **is** parameter into this temporary **double variable**.
10. Recall that the mohs value if greater than zero is a valid state. But if it isn't then it should be set to the DEFAULT_MOHS (similar to the constructor). Do the same here when setting the mohs from the **temporary double variable**.
11. To accept user input to set the other data members inherited from Ore, **call Ore's input function here**.
12. Lastly return the parameter **is**.

Operator Overload Helpers

```
ostream& operator<<(ostream& os, const Metal& met);
```

This operator overload allows for the use of language such as:

```
cout << met1;
```

Where met1 is an object of Metal type. This overload will perform the following:

4. First it prints out: “Metal” <newline>
5. Then it makes a call to the Metal class’s **display** function through the **met** parameter.
6. Lastly it returns the **os** parameter.

```
istream& operator>>(istream& is, Metal& met);
```

This operator overload allows for the use of language such as:

```
cin >> met1;
```

Where met1 is an object of Metal type. This overload will perform the following:

3. It makes a call to the Metal class’s **input** function through the **met** parameter.
4. Lastly it returns the **is** parameter.

Recall that these operator overloads don’t work specifically with cin or cout but more generically with ostream and istream objects.

AT-HOME MAIN MODULE

```
/*************************************************************************
// OOP244 Workshop 7: Dervied Classes & Custom I/O Operators
// File MetalTester.cpp
// Version 1.0
// Date       2019/10/23
// Author     Hong Zhan (Michael) Huang
// Description
// Tests the Metal class and its use of custom I/O operators
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
// Michael
/////////////////////////////////////////////////////////////////
*************************************************************************/

#include <iostream>
#include "Metal.h"
#include "Metal.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 << "Metal default constr" << endl;
  line(64, '-') << endl;
  number(1) << endl;
  Metal m1;
  m1.display(cout) << endl;

  cout << "Metal 5 arg constr invalid args" << endl;
  line(64, '-') << endl;
  number(2) << endl;
  Metal m2(100, 200, "Stuff", nullptr, 0);
  Metal m3(100, 200, "Stuff", "", 0);
  m2.display(cout);
  m3.display(cout) << endl;

  cout << "Metal 5 arg constr valid args (default vals, vals as is)" << endl;
  line(64, '-') << endl;
  number(3) << endl;
  Metal m4(1300, 200, "Volcanic", "Gold", -1);
  Metal m5(1000, 10, "Magmatic", "Silver", 4.3);
  m4.display(cout) << endl;
  m5.display(cout) << endl;

  cout << "Metal refinement w/ custom output operator" << endl;
  line(64, '-') << endl;
  number(4) << endl;
  m5.refine();
  m5.refine();
  cout << m5 << endl;

  cout << "Metal empty -> custom input operator" << endl;
  line(64, '-') << endl;
  number(5) << endl;
  cin >> m1;
  cout << endl << m1 << endl;

  cout << "Metal non-empty -> custom input operator" << endl;
  line(64, '-') << endl;
  number(7) << endl;
  cin >> m4;
  cout << endl << m4 << endl;

  return 0;
```

}

EXECUTION EXAMPLE RED VALUES ARE USER ENTRY

```
Metal default constr
---------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
This metal is imaginary

Metal 5 arg constr invalid args
---------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
This metal is imaginary
This metal is imaginary

Metal 5 arg constr valid args (default vals, vals as is)
---------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
Name: Gold
Weight: 1300.00
Purity: 20
Classification: Volcanic
Mohs: 2.50

Name: Silver
Weight: 1000.00
Purity: 10
Classification: Magmatic
Mohs: 4.30

Metal refinement w/ custom output operator
---------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
Metal
Name: Silver
Weight: 960.00
Purity: 30
Classification: Magmatic
Mohs: 6.30

Metal empty -> custom input operator
---------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
Enter a value for this metal's name: Platinum
Enter a value for this metal's mohs: 3.5
Enter a value for this ore's weight: 200
Enter a value for this ore's purity: 110
Enter a classification for the ore (MAX 30 chars): Sediment

Metal
Name: Platinum
Weight: 200.00
Purity: 20
Classification: Sediment
Mohs: 3.50

Metal non-empty -> custom input operator
```

```
---------------------------------------------------------------
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
Enter a value for this metal's name: Mythril
Enter a value for this metal's mohs: 999
Enter a value for this ore's weight: 10
Enter a value for this ore's purity: 100
Enter a classification for the ore (MAX 30 chars): Mythic

Metal
Name: Mythril
Weight: 10.00
Purity: 100
Classification: Mythic
Mohs: 999.00
```

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 Ore & Metal modules and the MetalTester.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/WS07/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.