Part D – Polymorphism

Virtual Functions

Workshop 8

(V1.0 – updated specs for typos/clarity)

In this workshop, you are to implement an abstract definition of behavior for a specific type.

LEARNING OUTCOMES

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

- define a pure virtual function
- code an abstract base class
- implement behavior declared in a pure virtual function
- explain the difference between an abstract base class and a concrete class
- describe what you have learned in completing this workshop

SUBMISSION POLICY

The workshop is comprised of only 1 section;
lab - 100% of the total mark.

*The DIY parts are now replaced with the project milestones under the **OOP-Project** repository.*

Workshop 8 is to be submitted during the workshop period from the lab. To get the mark for this workshop you must attend the lab.

Start the workshop few days earlier and come to the workshop sessions with questions and attendance.

Remember: you must be present at the lab in order to get credit for the workshop.
If you attend the lab period and cannot complete the workshop during that period, ask your instructor for permission to complete after the period.

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 wrongdoing.

Late Submission Penalties:

Late submission: 1 to 6 days -50% after that submission rejected.
- If the content of sources.txt is missing, the mark for the whole workshop will be **0.**

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.):

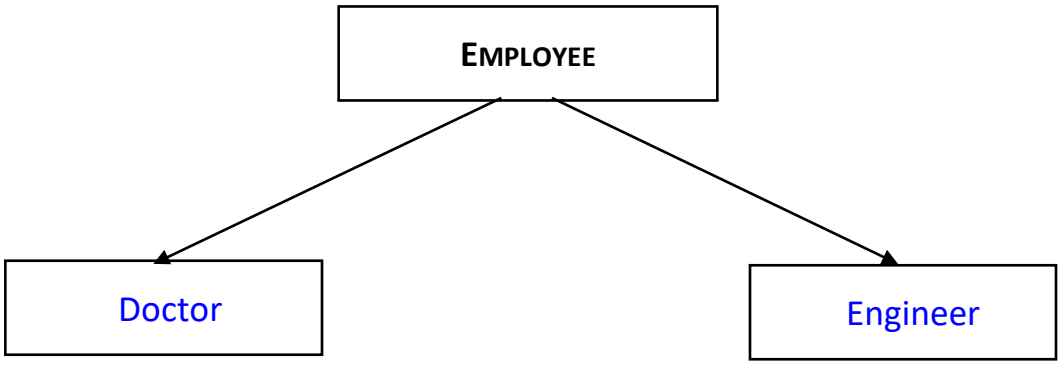
~profname.proflastname/submit 244/NXX/WS08/lab -due<ENTER>

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>

Utils Module

For those of you who would like to reuse their previously developed functions and classes in later workshops, an empty module is added to the workshop called `Utils` (`Utils.cpp` and `Utils.h`). These files are empty and will be compiled with your workshop.
If you don't have any interest in reusing your previous functions, leave these files as they are, and they will not have any effect in the workshop.
If you have anything that you would like to add to the project, place their code in the `".cpp"` file and the definitions and prototypes in the `".h"` file. This header file will be included in the tester program.
In any case when submitting your work, make sure that these two files (empty or not) are submitted along with your workshop.In-Lab (100%)



In this workshop we will be establishing the above class hierarchy. The **Employee** class will be an **abstract base class** from where the **Doctor** and **Engineer** classes will be derived.

EMPLOYEE MODULE

The Employee module is the interface to the hierarchy. It consists of only an *abstract base class (only header file no implementation)* and cannot be instantiated. It has no private members and only public pure virtual functions.

To accomplish the above follow these guidelines:

Design and code a class named **`Employee`**. Follow the usual rules for creating a module: (i.e. Compilation safeguards for the header file and the sdds namespace).

PUBLIC MEMBERS:

Pure Virtual Members

The definition of your **`Employee`** interface includes the following pure virtual member functions-

```
void setSalary(double);
This function will set the salary of the Employee

void bonus();
This function will calculate the bonus of the Employee

bool workHours();
A query that returns true if Employee is working for minimum hours.

ostream& display(ostream& os) const;
This function display details about the Employee.
```

DOCTOR MODULE

The **Doctor** class is derived from the Employee class and implements the Employee abstract base class.

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

MIN_W_HOURS with a value of 6.
This constant integer number is used to check the minimum working hours.

MAX_CHAR with a value of 20.
This constant number represents the maximum length for a **Doctor`s** type excluding the *nullbyte* (\0).

PRIVATE MEMBERS:

A Doctor will have the following data members:

- m_type
  A statically allocated **character array** that will hold the type of a Doctor. Doctor can be specialist or general. Its length is determined by MAX_CHAR.
- `m_salary`
  This is a **double value** that represents the salary of a doctor. A specialist doctor`s salary will be 2000 more than a general doctor.
- `m_whours`
  This is an **integer value that** represents the working hours of a doctor.
- `m_specialist`
  This is a **bool** value to indicate whethere a doctor is a specialist. This value is optional. If no value is passed it will be set as false.

PUBLIC MEMBERS:

Upon instantiation, a `Doctor` object either receives no information or it will receive information on all the above four values.

Default constructor – This constructor should set a Doctor object to **a safe empty state.**

4 Argument Constructor- This constructor will receive 4 parameters. The first parameter receives type of a doctor followed by salary and working hours. Last parameter receives a Boolean value. If the last argument is not provided, ***m_specialist*** is set to false.
The following validation needs to consider-

- If the salary and working hours are greater than zero, then constructor sets all the member variables except m_salary. Then it uses the setSalary() member function to set the value of m_salary.

- If either of working hours or salary are less than or equal to zero, the object is set to a safe empty state.

```
void setSalary(double);
```
This member function receives a double type salary and It will set the salary of a doctor depending on the value of the ***m_specialist***. If the doctor is a specialist, then the set value will be increased by 2000 dollars, otherwise it will set the salary to the incoming argument with no change.

```
bool workHours();
```
This member function returns true if a Doctor is working for MIN_W_HOURS

```
void bonus()
```
This member function calculates the bonus depends on the number of working hours then adds it to the salary. If the working hours is more than MIN_W_HOURS, the doctor will get 10% bonus otherwise 5%.

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

This member function will display the current details of a doctor. This public member function receives a reference to an ostream object.

- If a doctor`s salary and working hours is greater than zero it will print:

```
"Doctor details" <newline>
"Doctor Type: [type]" <newline>
"Salary: [salary]" <newline>
"Working Hours: [whours]" <newline>
```

  The salary should have two decimal place precision.

- If the class is in an empty state, then it will print:

```
"Doctor is not initiated yet"<newline>
```

- Lastly at the end of the function **return the parameter os**.
  Refer to the sample output for details.

ENGINEER MODULE

The **Engineer** class is derived from the Employee class and implements the Employee abstract base class.

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

MIN_HOURS with a value of 5.
This constant integer number is used to check the minimum working hours.
MAX_LEVEL with a value of 4.
This constant integer number is used to check the maximum level of an Engineer.

PRIVATE MEMBERS:

An **Engineer** will have the following data members:

- `m_esalary`
  This is a **double value** that represents the salary of an Engineer. A higher level (MAX_LEVEL) engineer`s salary will be 3000 more than other levels Engineer.
- `m_ewhours`
  This is an **integer value that** represents the working hours of an Engineer.
- `m_level`
  This is an **integer** value to indicate the level of a Engineer.

PUBLIC MEMBERS:

Upon instantiation, an `Engineer` object either receives no information or it will receive all of the above three values.

Default constructor – This constructor should set an Engineer object to **a safe empty state.**

3 Argument Constructor- This constructor will receive 3 parameters. The first parameter receives salary followed by working hours and level.

The following validation needs to be considered:

- If the salary, working hours and level is greater than zero, then the object first sets the level and the working hours and then call the setSalary() function to set the salary of the Engineer.
- If any of the data is invalid (less than or equal to zero), this function will set the object to a safe empty state.

```
void setSalary(double);
```
This member function receives a double type salary and sets the salary depends on the level of the Engineer. If the Engineer is at MAX_LEVEL, then the set value will be increased by 3000 dollars, otherwise it will set the salary to the incoming argument with no change.

```
bool workHours();
```
This member function returns true if an Engineer is working for MIN_HOURS.

```
void bonus()
```
This member function calculates the bonus depending on the working hours and the level, and then it adds the bonus to the salary. If an engineer works more than the MIN_HOURS, and is at the MAX_LEVEL, she will get a 10% bonus otherwise the bonus will be 5%.

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

This member function will display the current details of an Engineer. This public member function receives a reference to an ostream object.

- If an engineer`s salary and working hours is greater than zero it will print out the following:

```
"Engineer details" <newline>
"level: [level]" <newline>
"Salary: [salary]" <newline>
"Working Hours: [whours]" <newline>
```

  The salary should have two decimal place precision.

- If the object is in empty state, then it will print:

```
"Engineer is not initiated yet"<newline>
```

- Lastly at the end of the function **return the parameter os**.
  Refer to the sample output for details.

In-Lab Main Module

```
// OOP244 Workshop 8: virtual functions
// File: EmployeeTester.cpp
// Version: 1.0
// Date: 03/11/2020
// Author: Nargis Khan
// Description:
// Tests the Employee and its derived classes via virtual functions
////////////////////////////////////////////
// ----------------------------------------------------------
// Name Date Reason
// Nargis
////////////////////////////////////////////////////////////////
#include<iostream>
#include "Doctor.h"
#include "Doctor.h"
#include "Engineer.h"
#include "Engineer.h"

using namespace std;
using namespace sdds;

int main() {

   cout << "----------------------------------------------------------" << endl;
   cout << "*** checking Doctor & Engineer default constructor **** " << endl;
   Doctor d1;
   Engineer e1;
   d1.display(cout);
   e1.display(cout) << endl;
   cout << "----------------------------------------------------------" << endl;
   cout << "*** checking Doctor & Engineer parameterized constr (valid) *** " << endl;
   Employee* emp[4];

   emp[0] = new Doctor("General", 10100.00, 6);
   emp[1] = new Doctor("Specialist", 10100.00, 10, true);
   emp[2] = new Engineer(5595.00, 5, 1);
   emp[3] = new Engineer(5595.00, 8, 4);

   for (int i = 0; i < 4; ++i)
      emp[i]->display(cout) << endl;

   cout << "----------------------------------------------------------" << endl;

   cout << "*** checking bonus on current salary of Doctor ***" << endl;

   emp[0]->bonus();
   emp[1]->bonus();
   emp[0]->display(cout) << endl;
   emp[1]->display(cout) << endl;

   cout << "----------------------------------------------------------" << endl;
   cout << "*** checking bonus on current salary of Engineer ***" << endl;
   emp[2]->bonus();
   emp[2]->display(cout) << endl;;
   emp[3]->bonus();
```

```
   emp[3]->display(cout) << endl;
   // Cleaning up the memory
   for (int i = 0; i < 4; ++i)
      delete emp[i];
   return 0;
}

/*
-----------------------------------------------------------
*** checking Doctor & Engineer default constructor ****
Doctor is not initiated yet
Engineer is not initiated yet

-----------------------------------------------------------
*** checking Doctor & Engineer parameterized constr (valid) ***
Doctor details
Doctor Type: General
Salary: 10100.00
Working Hours: 6

Doctor details
Doctor Type: Specialist
Salary: 12100.00
Working Hours: 10

Engineer details
level: 1
Salary: 5595.00
Working hours: 5

Engineer details
level: 4
Salary: 8595.00
Working hours: 8

-----------------------------------------------------------
*** checking bonus on current salary of Doctor ***
Doctor details
Doctor Type: General
Salary: 10605.00
Working Hours: 6

Doctor details
Doctor Type: Specialist
Salary: 13310.00
Working Hours: 10

-----------------------------------------------------------
*** checking bonus on current salary of Engineer ***
Engineer details
level: 1
Salary: 5874.75
Working hours: 5

Engineer details
level: 4
Salary: 9454.50
Working hours: 8
```

```
*/
```

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 sources.txt, Doctor.cpp, Engineer.cpp and the EmployeeTester.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/WS08/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.

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.

The reflection should also contain any thoughts and learning from the final project milestones when possible.

Reflection Submission

You can submit your reflection 4 to 6 days after your lab session. Upload reflect.txt to your matrix account. Then, run the following command 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/WS08/reflect <ENTER>

and follow the instructions generated by the command.

To see the when you can submit your reflection and when the submission closes, (like any other submission) add **-due** to the end of your reflection submission command:

~profname.proflastname/submit 244/NXX/WS08/reflect -due<ENTER>