Part D – Polymorphism

Virtual Functions

Workshop 8

(V0.9 – on the day of your lab, before submission, make sure there are no new updates or corrections)

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

This workshop has two components:

- **lab**: to be completed by the end of the day of your lab;
- **Reflection**: to be completed by the end of the week of your lab (Sunday @ 23:59:59);

You must be present at the lab for the duration of the entire session in order to get credit for the *lab* portion.

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.

REFLECTION

After the workshop is fully completed (meaning the **lab** was already submitted), create a text file named **reflect.txt** that contains your detailed description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. Add any other comments you wish to make.

A section below provides some questions and expectations you should aim to answer at a minimum for the reflection. Aiming higher than that will be to your benefit as a poorly done reflection can incur a **mark reduction** (see Submission Penalties below).

This reflection is a mandatory part of the workshop submission.

LATE SUBMISSION PENALTIES:

If you do not follow the submission the policies, the following penalties will be applied:

- If the **lab** is submitted past the due date, it will be receive **0%.**
- If the submitted **reflection** is deemed to be insufficient in depth or does not demonstrate a strong understanding of the core concepts used in the Workshop, a **maximum reduction of 40%** may be applied the overall mark of the Workshop.
- If any of *lab* or *reflection* is missing on rejection date, the total mark of the workshop will be **0%**.

To see the dues for a deliverable, use:

```
~cornel.barna/submit 200/NXX/WSXX/lab -due<ENTER>
~cornel.barna/submit 200/NXX/WSXX/reflect -due<ENTER>
```

COMPILING AND TESTING YOUR PROGRAM

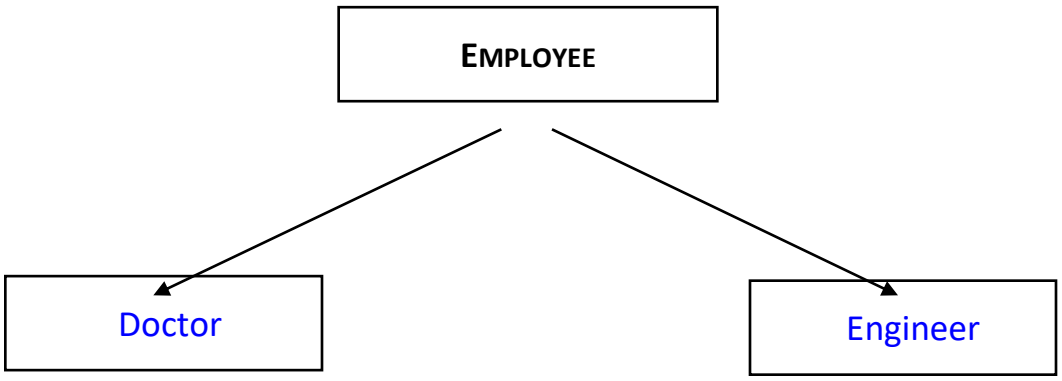
Compile all your code using this command on matrix:

```
g++ -Wall -std=c++11 -o ws file1.cpp file2.cpp ...<ENTER>
```

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



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.

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 200/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 should 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.

You can submit your reflection until Sunday @ 23:59:59 of the week of your lab session. Upload reflect.txt to your `matrix` account. Then, run the following command from your account (use your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~cornel.barna/submit 200/NXX/WSXX/reflect<ENTER>
```

and follow the instructions generated by the command.