NATIONAL UNIVERSITY OF COMPUTER & EMERGING SCIENCES
ISLAMABAD CAMPUS
COMPUTER PROGRAMMING (CS103) - FALL 2018
ASSIGNMENT-6

Due Date: November 28, 2018 (05:00 pm)

Instructions:

1. *Make sure that you read and understand each and every instruction.*
2. *Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded straight zero in this assignment or all assignments.*
3. *Submit a single '.zip' file for your assignment and each problem solution must be provided in a folder and three separate CPP file. For instance, you must name the folder containing solution of first problem as 'q1' and second as 'q2' and so on.*
4. *You have to write each class in header and .cpp files Name the files as per your question. For instance, folder named as q1 will have three files. q1.h, q1.cpp and q1main.cpp.No code will be accepted without these files.*
5. *Note: You have to follow the submission instructions to the letter. Failing to do so can get a zero in assignment.*

Q1) Define a pure abstract base class called BasicShape. The BasicShape class should have the following members:

Private Member Variable:

- area, a double used to hold the shape's area.

Public Member Functions:

- getArea. This function should return the value in the member variable area.
- calcArea. This function should be a pure virtual function.

Next, define a class named Circle. It should be derived from the BasicShape class. It should have the following members:

Private Member Variables:

- centerX, a long integer used to hold the x coordinate of the circle's center.
- centerY, a long integer used to hold the y coordinate of the circle's center.
- radius, a double used to hold the circle's radius.

Public Member Functions:

- constructor accepts values for centerX, centerY, and radius. Should call the overridden calcArea function described below.
- getCenterX returns the value in centerX.
- getCenterY returns the value in centerY.
- calcArea calculates the area of the circle (area = 3.14159 * radius * radius) and stores the result in the inherited member area.

Next, define a class named Rectangle. It should be derived from the BasicShape class. It should have the following members:

Private Member Variables:

- width, a long integer used to hold the width of the rectangle.
- length, a long integer used to hold the length of the rectangle.

Public Member Functions:

- constructor accepts values for width and length. Should call the overridden calcArea function described below.
- getWidth returns the value in width.
- getLength returns the value in length.
- calcArea calculates the area of the rectangle (area = length * width) and stores the result in the inherited member area.

Demonstrate the classes in a program that has an array of BasicShape pointers. The array elements should be initialized with the addresses of dynamically allocated Circle, and Rectangle objects. The program should then step through the array, calling each object's calcArea function.

Q2) Design a generic class to hold the following information about a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following member functions:

- **Constructor**: Accepts arguments for the balance and annual interest rate.
- **deposit**: A virtual function that accepts an argument for the amount of the deposit. The function should add the argument to the account balance. It should also increment the variable holding the number of deposits.
- **withdraw**: A virtual function that accepts an argument for the amount of the withdrawal. The function should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.
- **calcInt**: A virtual function that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:

  Monthly Interest Rate = (Annual Interest Rate / 12)
  Monthly Interest = Balance * Monthly Interest Rate
  Balance = Balance + Monthly Interest
- **monthlyProc**: A virtual function that subtracts the monthly service charges from the balance, calls the calcInt function, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a savings account class, derived from the generic account class. The savings account class should have the following additional member:

- status (to represent an active or inactive account)

If the balance of a savings account falls below 25000, it becomes inactive. (The status member could be a ag variable.) No more withdrawals may be made until the balance is raised above 25000, at which time the account becomes active again. The savings account class should have the following member functions:

- **withdraw**: A function that checks to see if the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the base class version of the function.
- **deposit**: A function that checks to see if the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above 25000, the account becomes active again. The deposit is then made by calling the base class version of the function.

- **monthlyProc**: Before the base class function is called, this function checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of 100 for each withdrawal above 4 is added to the base class variable that holds the monthly service charges. (Don t forget to check the account balance after the service charge is taken. If the balance falls below 25000, the account becomes inactive.)

Next, design a checking account class, also derived from the generic account class. It should have the following member functions:

- **withdraw**: Before the base class function is called, this function will determine if a withdrawal (a check written) will cause the balance to go below 0. If the balance goes below 0, a service charge of 150 will be taken from the account. (The withdrawal will not be made.) If there isn't enough in the account to pay the service charge, the balance will become negative and the customer will owe the negative amount to the bank.
- **monthlyProc**: Before the base class function is called, this function adds the monthly fee of 50 plus 0.10 per withdrawal (check written) to the base class variable that holds the monthly service charges.

Write a complete program that demonstrates these classes by asking the user to enter the amounts of deposits and withdrawals for a savings account and checking account. The program should display statistics for the month, including beginning balance, total amount of deposits, total amount of withdrawals, service charges, and ending balance.

Q3) Board Game: The board game comes with a board, divided into 40 squares, a pair of six-sided dice, and can accommodate 4 players. It works as follows:

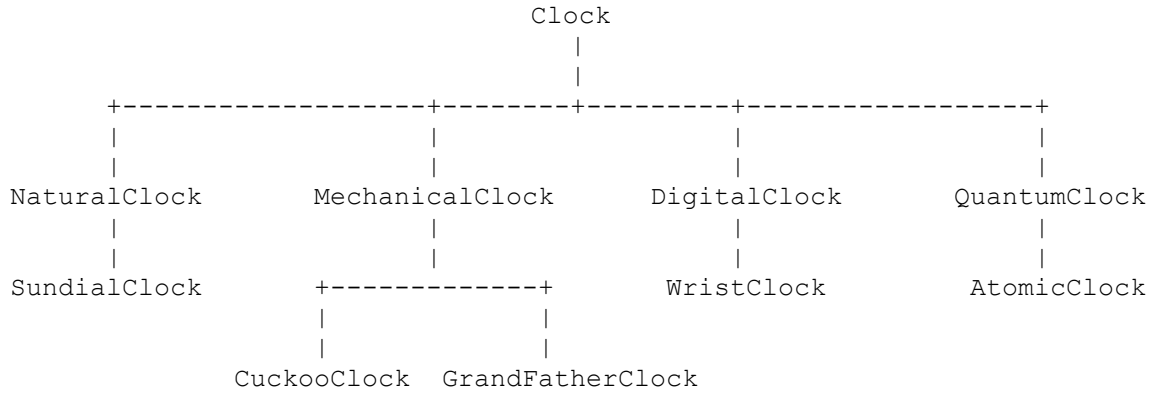
- Each Player has a name, money and the current position on the board
- All players start from the first square.
- One at a time, players take a turn: roll the dice and advance their respective tokens on the board.
- A round consists of all players taking their turns once.
- Players have money. Each player starts with 7 cents.
- The amount of money changes when a player lands on a special square:
  - ✓ Squares 5, 15, 25, 35 are bad investment squares: a player has to pay 5 cents. If the player cannot afford it, he gives away all his money.
  - ✓ Squares 10, 20, 30, 40 are lottery win squares: a player gets 10 cents.
- The winner is the player with the most money after any player advances beyond the 40th square.

Bad investment and lottery win squares are special cases of squares, which differ in a way they affect players. To model this you can introduce class SQUARE and then use inheritance and polymorphism to implement the behavior of special squares. You can store squares of all kinds in a single polymorphic container (e.g. array of squares pointers) and let dynamic binding take care of which special behavior applies for each square.

Q4) Write a program to simulate the simultaneous running of a collection of clocks for a period of one (1) week, where each drifts a specified amount of time per second.

Class Hierarchy:

The following classes are related in a class hierarchy and represent a family of clock types:



HIERARCHY NOTES:

- Classes "Clock", "NaturalClock", "MechanicalClock", "DigitalClock", and "QuantumClock" are abstract base classes.
- Classes "SundialClock", "CuckooClock", "GrandFatherClock", "WristClock", and "AtomicClock" are concrete derived classes.
- Each class in the hierarchy has a constructor which takes the same arguments as those in the constructor for abstract base class "Clock" i.e hours, min, sec, secondsPerTick and driftPerSecond. Make sure the arguments passed to a concrete derived class's constructor on instantiation are passed up through the hierarchy via base class constructor invocation.
- The constructor defined in abstract base class "Clock" is the only one with an implementation. The constructors in all other classes are empty.
- The destructor defined in abstract base class "Clock" is the only one with an implementation. The destructors in all other classes are empty.
- The following virtual methods are defined in each class:
  - ✓ void reset ();
  - ✓ void tick ();
  - ✓ void displayTime ();

- The virtual methods noted in (6) are only implemented in the concrete derived classes while the implementation of these methods in the abstract base classes is empty.

Clock Collection Management:

The collection of clocks is comprised of one (1) dynamically allocated instance of each concrete derived class in the hierarchy i.e. array of clock pointers of size 5.

Simulation Workflow:

What follows is a characterization of the workflow governing the clock simulation:

- Create collection of clocks, initializing each as follows:
    1. Time = [00:00:00]
    2. Seconds per tick = 1
    3. Drift per second = Value defined in Table 1

- Display time of each clock before clocks run
- Perform 604,800 ticks (1 week) per clock in the collection
- Display time of each clock after clocks run
- Destroy collection of clocks

Clock Drifts:

Each clock has a specified drift, quantified in terms of an amount of drift per second. See Table 1 for an enumeration of clock drifts.

Output:

Reported clock times after resetting:

```
Sundial Clock          time [00:00:00] - total drift = 0 seconds
Cuckoo Clock           time [00:00:00] - total drift = 0 seconds
Grandfather Clock      time [00:00:00] - total drift = 0 seconds
Wrist Clock            time [00:00:00] - total drift = 0 seconds
Atomic Clock           time [00:00:00] - total drift = 0 seconds
```

Running the clocks for one (1) week...

Reported clock times after running:

```
Sundial Clock          time [24:00:00] - total drift = 0 seconds
Cuckoo Clock           time [24:00:00] - total drift = 420 seconds
Grandfather Clock      time [24:00:00] - total drift = 210 seconds
Wrist Clock            time [24:00:00] - total drift = 20.9999 seconds
Atomic Clock           time [24:00:00] - total drift = 0 seconds
```

Table 1 - Clock Drifts

Clock Type	Drift (amount per second)	Comment
Sundial Clock	0.0	No drift
Atomic Clock	0.0	No drift
Cuckoo Clock	0.000694444	60 seconds per day
Grandfather Clock	0.000347222	30 seconds per day
Wrist Clock	0.000034722	3 seconds per day