EE 1301
Lab 7: Introduction to Objects

Object-oriented programming is a powerful computational paradigm in which we approach problem solving from the perspective of how data objects interact. Our exploration begins with the construction of simple representations (classes) that C++ uses as "recipes" or "blueprints" for instantiating objects.

Warm-up

Complete the following paper/pencil exercises and discuss your results with your TA.
1) Consider the following class declaration:

```
1  class Point
2  {
3     public:
4     void showPoint( );
5     Point( );
6     Point(int, int);
7     int  xlocation;
8     int  ylocation;
9  } ;
```

a. Which line contains the default constructor for the class?

b. The data members are declared as public. Will this work?

c. Would this be a good idea for a large program?

d. Write the function definition for the constructor declared in line 6.

2) Now consider the following code segment, which uses the `Point` class definition from Warm-up 1, and answer the questions below.

```
1  int main( )
2  {
3     Point.xlocation = 3;
4     Point.ylocation = 10;
5
6     Point p1;
7     p1 = Point(5,6);
8
9     Point p2( );
10    return 0;
11 }
```

a. Describe the problem with the statements in lines 3 and 4.

b. Describe (in detail) what the statement in line 7 does (consult the cplusplus.com tutorial on classes if you are unsure).

c. What is the problem with the statement in line 9 (assuming you are trying to declare an object of the point class)?

d. What does the statement in line 9 currently declare?

e. Rewrite the statement in line 9 to declare a default instance of the `Point` class.

Stretch

1) Bug Class

Construct a user-defined object class named `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, the bug's position changes by one unit in the direction it is facing.

The Bug class will have data members `position` (type `int`) for the bug's position on the horizontal line and `dir` (type `int`) for the direction the bug is facing. A `dir` value of (positive) 1 indicates that the bug is moving to the right, and a value of -1 signifies that it is moving to the left. There are five member functions of the `Bug` class, described as follows.

- a default constructor that initializes the bug's `position` to 0 and `dir` to +1.
- a constructor with a single argument that initializes the initial `position` of the bug to a value set by the user (the initial value of `dir` should always be +1)
- `void move()` moves the bug one space in direction `dir`
- `void turn()` changes the direction the bug is facing
- `void display()` displays the data members of a `Bug` object on the terminal screen or writes them to a file (see example below)

Before writing any code, you should think about what the code for this class will look like. Discuss it with a neighbor or the TA. Make sure you have a good idea of what you need to do before you start to write the code for this class.

Write your bug class code in two files: an interface (or header) file named Bug.hpp, and an implementation (or definition) file named Bug.cpp. Remember that the implementation file will need to include the directive `#include "Bug.hpp"`. Also use `#ifndef`, `#define`, and `#endif` in your Bug.hpp file to make sure the `Bug` class is only defined once.

When you have your bug files completed, write a test program to create a `Bug` object at position 10, move the bug, turn the bug, and move the bug again. At each position, display

the bug's data members on the terminal screen. Your main program should be in an application file, i.e., a separate file from your bug files. That application file should also have a directive `#include "Bug.hpp"`. Remember that to compile your program you will need to include both the application file name and the file name Bug.cpp in the compilation command. For example, if you named your application file buggy.cpp, then the following command would compile your code.

```
g++ Bug.cpp buggy.cpp -o buggy
```

Here is a sample of what the output should look like.

```
position = 10, direction = 1
position = 11, direction = 1
position = 11, direction = -1
position = 10, direction = -1
```

2) Complex Class

A complex number has the form a + b*i*, where a and b are real numbers and $i = \sqrt{-1}$. For example, 2.4 + 5.2*i* and 5.73 - 6.9*i* are complex numbers. Here, a is called the real part of the complex number, and b*i* the imaginary part.

In this exercise, you will create a class named `Complex` to represent complex numbers. (Some languages, including C++, have a complex number library; in this problem, however, you will write the complex class yourself.) The members of this class are as follows.

Private data members:

`real` and `imag` of type `double` – represent the real and imaginary part of the complex number (`real + imag i`)

Public member functions:

- A default constructor that initializes `real` and `imag` to zero
- `void input(istream&);` //Input values from the specified stream for `real` and `imag`
- `void output(ostream&);` // Output the Complex number to the specified stream in this form: `2.3 + 4.6i`
- `double getReal();` // Accessor function that returns the value of data member `real`
- `double getImag();` //Accessor function that returns the value of data member `imag`
- `void setReal(double);` // Mutator function that sets the value of `real`
- `void setImag(double);` // Mutator function that sets the value of `imag`

Before writing any code, discuss with a neighbor or the TA what the code for this class will look like. Make sure you have a good idea of what you need to do before you start writing the code for this class.

Write a `main()` driver function that calls the appropriate member functions to do the following:

- Declare two `Complex` objects: `c1` and `c2`.
- Prompt the user and use the input function to input complex values for the two objects, `c1` and `c2`.
- Display `c1` and `c2` using the output function.
- Use `setReal()` to change the value of `real` for object `c2` to 22.2.
- Use `getReal()` to obtain the value of `real` for object `c2`. Display the value on the terminal screen.
- Display the updated `c2` value using the `output()` function.

Example (user input is underlined):

```
Enter values for real and imaginary coefficients: 17.4 3.9
Enter values for real and imaginary coefficients: 10.2 16.8
17.4 + 3.9i
10.2 + 16.8i
Enter a new value for the real coefficient: 22.2
The new real coefficient is 22.2
22.2 + 16.8i
```

Workout

1) Playing Card Class

In this problem, you will implement a `DeckOfCards` class that simulates a deck of playing cards. The class will utilize an important algorithm to "shuffle" the simulated card deck.

Part 1: Create an object class named `DeckOfCards` that models a randomized deck of playing cards. Individual "cards" will be represented by integer values as follows.

```
0 1 2 3 4 5 6 7 8  9 10 11 12 13 14 15 16 . . . 51
A 2 3 4 5 6 7 8 9 10  J  Q  K  A  2  3  4 . . .  K
```

The value 0 represents the Ace of the "first" suit, 1 is the 2 of the first suit, 13 is the Ace of the "second" suit, and so on. The actual "suit" of a card (spades, hearts, etc.) will not be needed for now and can be ignored. Also note that in this problem we start counting with 0, rather than with 1.

The `DeckOfCards` class should include the following **private** data members:

- A 52-element integer array to represent the shuffled deck of cards
- An integer to maintain the index of the "next card" to be dealt from the deck

The `DeckOfCards` class must also provide the following **public** (member) functions:

- A default constructor that initializes the values of the card deck array with integer values 0...51 and then "shuffles" the card deck (see description below)

- A member function `dealCard()` that will return the "next" card value from the shuffled deck. If no cards remain, then the deck should be reshuffled and reset.
- A member function `shuffle()` that takes no arguments and "shuffles" (randomizes) the values in the card "deck" using the Knuth Shuffle algorithm. Given an array `a` with `n` elements, the Knuth Shuffle algorithm works as follows.

FOR i = n-1 TO 1 REPEAT the following two lines:
    j = a pseudo-random integer from the interval 0 <= j <= i
    exchange a[i] and a[j]

Part 2: Write a `main()` driver function to verify the correct operation of all `DeckOfCards` member functions. At a minimum, do the following:

- Instantiate a `DeckOfCards` object
- Create an array that will represent a 4-card "hand" of dealt cards
- Write a user-defined function (not part of any class) that will take as arguments (a) an array of cards representing a single "hand", and (b) the number of cards in the array. The function should display the card values of the hand on the terminal in the following format:

```
A 2 4 K
```

- Using the `DeckOfCards` object, deal 13 separate 4-card hands and display them on the terminal, each hand on a separate line. When all is said and done, verify that you've dealt exactly 4 aces, 4 twos, 4 threes, etc. in a "reasonably" random fashion.

Example:

```
A 4 3 5
10 8 6 Q
Q 5 2 Q
Q J 6 A
4 3 K J
7 5 A 2
9 8 2 J
2 K 7 6
J 5 10 K
9 6 4 7
10 9 3 9
10 8 8 K
7 A 3 4
```

2) Casino Blackjack

(This is a long problem, but it provides good practice with classes and objects.)

In this problem, you will use the `DeckOfCards` class created in the previous exercise to construct a simulation program that determines how frequently the dealer will go "bust" when playing the casino game Blackjack. The simulation involves dealing a large number of Blackjack hands according to the dealer rule "stand on soft-17" and reporting the frequencies of various scoring outcomes.

In this version of Blackjack, a player initially receives 2 cards and optionally draws 1, 2, or 3 more in an attempt to bring the total value of the cards as close as possible to 21 without going over. To determine the value of the hand, the cards labeled 2 through 10 are scored as 2 through 10 points, respectively, "face cards" (jack, queen, and king) are scored as 10 points, and the ace can count as either 1 or 11, whichever is better for the player. If the score is over 21, the player loses (the player is "busted"), regardless of what the dealer does. When played in a casino, the dealer plays according to fixed rules. For this simulation, we'll assume the casino employs the "stand on soft-17" rule that is more advantageous for the player.

Regardless of the player's score, the dealer is required to continue drawing cards until his/her hand achieves a value of 17 or more, or goes bust. Since an ace counts as either 1 or 11 points, each ace results in two possible scores for a hand. A "**soft**" score is a score that includes an ace that count as 11 points. For example, ace-ace-3-2 could be one of three possible scores: 27 (busted), 17 (soft-17), or 7. The actual score of a soft hand is the largest score that is less than or equal to 21. In this example, the score is 17, and it is a soft-17, since it contains an ace that counts as 11 points. Therefore, ace-ace-3-2 is an example of a "soft-17" and would require the dealer to stop taking cards (stand). As another example, the hand 10-7 also has a score of 17, but it is not a soft-17. Also, the hand 10-5-ace-ace has a score of 17, but it is not a soft-17 because the aces both count as 1 point. In all these cases, whether the 17 is a soft-17 or not, the dealer must stand at 17.

Write a program that does the following:

- Includes a function, `scoreDealer()` that takes as arguments an array of integers representing a Blackjack hand and an integer representing the number of cards in the hand and returns the dealer's score. If the hand is a "soft hand" (i.e., it contains an ace that counts as 11 points), your function must determine the largest soft score less than 21. If that is not possible, your function should return the lowest score over 21 with all the aces in the hand counted as 1.

- Using the `DeckOfCards` object, deal 10,000 Blackjack hands with the "stand on soft-17" rule (i.e. for each hand, continue to deal cards until the score reaches 17 or more). Your program must call the `scoreDealer()` function to determine the score of each hand as it is dealt.

- For each of the 10,000 hands, count the number of occurrences of the following. **Note: Natural blackjacks are counted both in the "natural blackjack" and the "dealer scores 21" categories)**:
  - dealer scores 17
  - dealer scores 18
  - dealer scores 19
  - dealer scores 20
  - dealer scores 21
  - dealer "busted"
  - "natural" blackjack (score equal to 21 using only the first two cards dealt)

- Display each of these statistics on the terminal as both frequency (count) and the percentage of total hands played.

Example:

```
Number of natural blackjack hands: 490 (4.9%)
hands of 17: 1443 (14.43%)
hands of 18: 1396 (13.96%)
hands of 19: 1372 (13.72%)
hands of 20: 1681 (16.81%)
hands of 21: 1234 (12.34%)
dealer bust: 2874 (28.74%)
```

Note: The theoretical probability for a dealer bust is ~29%.

Challenge

1) Card Games

Using the `DeckOfCards` class and the "stand on soft-17" rule for the dealer, write a complete program that plays the game of Blackjack. The computer should act as the dealer and allow the player to indicate if she/he wants to hit or stand.

Example (user input underlined):

```
Your hand: K 2
My hand: 7 3
Hit or Stand? (h/s): h
Your hand: K 2 3
Hit or Stand? (h/s): h
Your hand: K 2 3 4
Hit or Stand? (h/s): s
My hand:
7 3 8
You win!
Play again? (y/n): n
```