COSC 1437 (DL) - Fall 2016
Program Set #4

See 1437 Grading /Program Guide Sheet for directions and grading/submission information.

1. Implement Programming Exercise #9-Chapter 13 (Gaddis), pp.805. Use all the member variables/functions as mentioned in the text. Place all code into one file. Output should look similar to below.

Sample Run:

```
Enter total population: 230000
Enter number of births per year: 230
Enter number of deaths per year: 30

Population Statistics
        Population:   230000
        Birth Rate:   0.0010
        Death Rate:   0.0001
```

Name the program: TestPopulationXX.cpp, where XX are your initials.

2. Implement a Polynomial class with polynomial addition, subtraction, and multiplication. A variable in a polynomial only acts as a placeholder for the coefficients. Therefore, an array of coefficients and the corresponding exponent are needed. One way to implement is to use an array of doubles to store the coefficients. The index of the array is the exponent of the corresponding term. If a term is missing, it simply has a zero coefficient. Provide a default constructor, a copy constructor, and a parameterized constructor that enables an arbitrary polynomial to be constructed. Also, supply an overloaded operator = and a destructor. Provide for the following operations.

- Polynomial + polynomial
- Constant + polynomial
- Polynomial + constant
- Polynomial - polynomial
- Constant - polynomial
- Polynomial - constant
- Polynomial * polynomial
- Constant * polynomial
- Polynomial * constant

Supply functions to assign and extract coefficients, indexed by exponent as well. You will need to decide whether to implement the functions above as members, friends, or stand-alone functions. Write a C++ test program to test your class. Place all code into one file. Output should be user friendly.

Name the program: TestPolyXX.cpp, where XX are your initials.

3. Declare and define a class for a set of integers. A set is a collection of data without repetition or ordering. The class should only have two private data members: a pointer to a dynamically allocated

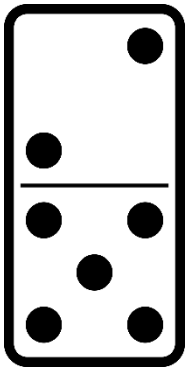
array of integers and an integer that holds the size of the set. The class should have the following methods:

- A constructor to create an empty set.
- A copy constructor
- A destructor
- A function to add an element to the set. It must check for duplicates.
- A function to remove elements from the set.
- A function to count the number of elements in the set.
- A binary friend function to determine the intersection of two sets. An intersection of two sets is another set with all elements common to the two sets.
- A binary friend function to determine the union of two sets. A union of two sets is another set that contains all elements that are in either set or in both. The union set can have no duplicates.
- A binary friend function to determine the difference of two sets. A difference of two sets is another set that has all elements that are in the first set but not in the second.
- A function that determines if an element is in a set. It should return true if the element is present and false if it is not.

Write a C++ test program to test your class. Place all code into one file. Output should be user friendly.

Name the program: `TestSetXX.cpp`, where XX are your initials.

4 (**). A domino is a small tile that represents the roll of two dice. The tile, commonly called a bone, is rectangular with a line down the center. Each end of the tile contains a number. In the most popular domino set, the double-six, the numbers vary from 0 (or blank) to 6. This produces 28 unique tiles. Dominoes are referred to by the number of dots (or pips) on each end, with the lower number usually listed first. Thus, a tile with a 2 on one end and a 5 on the other is referred to as a "2-5". The domino represented as [2 | 5] is shown to the right.



A tile with the same number on both ends is called a "double", so a "6-6" is referred to as "double-six". Tiles which have ends with the same number of dots are members of the same "suit". In a double-six set, there are seven suits (blank, 1, 2, 3, 4, 5, 6), each with seven members (0-5, 1-5, 2-5, 3-5, 4-5, 5-5, & 5-6) make up the "fives" suit, for instance. Except for the doubles, each tile belongs to two suits. Write an OO C++ game that plays the game of dominoes. Here the game will be played by the user and the computer.

When the play begins the dominoes must be randomly shuffled, the collection of shuffled tiles is called the boneyard. When play begins, the player and computer select 6 dominoes from the set of dominoes provided from the boneyard, and the holder of the "heaviest" domino goes first as the start of the "train". Typically, this is the double-six. If no one holds the double-six, then the double-five is played, and so on. At each turn the player tries to add to the train by placing one of his dominoes with a matching value at the head (left hand side) or the tail (right hand side) of the train. Note that it may be necessary to "flip" a domino so that it may be added to the train. For example, if the train is

```
[4 | 5][5 | 7][7 | 3]
```

then the player may either:

- place a domino that has a 4 on one half (such as `[4 | 6]` or `[0 | 4]`) at the head of the train, creating for example

  `[6 | 4] [4 | 5] [5 | 7] [7 | 3]` (Note that `[4 | 6]` was "flipped")

  OR

  `[0 | 4] [4 | 5] [5 | 7] [7 | 3]`

- place a domino with 3 on one half (such as `[3 | 0]` or `[2 | 3]`) at the tail of the train, creating for example
  `[4 | 5] [5 | 7] [7 | 3] [3 | 0]`

  OR

  `[4 | 5] [5 | 7] [7 | 3] [3 | 2]` (Note that `[2 | 3]` was "flipped")

When it is the computer's turn to play, the program will search its dominoes and play the first domino it finds that can be added to either the head or the tail of the train. In accordance with the rules described above. If the player (user or computer) cannot add one of his dominoes to the train, he chooses a domino from the remaining dominoes in the bone yard and adds it to his dominoes. If the boneyard is empty, the player simply passes his turn. The next player then takes his turn. A game ends either when a player (user or computer) plays all his tiles, or when a game is blocked. A game is blocked when no player is able to add another tile to the layout. A player wins if they are the first to play all their tiles.

Implementation:

- When it is the user's turn to play, you will present the user with a menu.
- You must enforce all game rules and perform all necessary error checking, responding with appropriate error messages and (if possible) continuing the game.
- If they select a tile not in their hand or cannot be played, they should be re-prompted for another tile.
- Print user's dominoes -- this choice prints all of the user's dominoes in the format shown in the example above. Each domino is represented as 2 integers.
- Add a domino to the head of the train -- the user specifies the domino by entering the domino's 2 values (on the same line, separated by a space). The values may be entered in either order. If the specified domino does not belong to the user, an error is produced and the user must choose from the menu again. If the domino does belong to the user, it is added to the head of the train (being automatically flipped if necessary) and removed from the user's set of dominoes. This choice ends the user's turn.

- Add a domino to the tail of the train -- same as adding a domino to the head of the train, but the domino is added to the tail. This choice ends the user's turn.
- Draw a domino from the bone yard -- the first domino in the boneyard is removed from the boneyard and added to the user's set of dominoes. If the boneyard is empty, an appropriate message will be displayed and the user will choose another option from the menu. This choice will end the user's turn.
- Pass this turn -- It's now the system's turn to play. It is an error for the user to pass his turn if the boneyard is not empty. This choice ends the user's turn.
- Whenever a domino is added to the train (by either the user or the system), your program will print the new train.
- When a player (user or system) chooses a domino to add to the train, your program will automatically "flip" the domino if necessary. See the example in the description above.

Some classes that will be needed are specified below and additional classes can be used. It is up to the student to figure out what the responsibilities (methods) of each class will be. Those classes are:

- `Domino` (object) - to represent the entire game. Creating a `Domino` object should start the game.
- `Player` (object) - to represent a single player.

Also, no object should directly examine or alter the variables of any other object; all access should be done by talking to (calling methods of) the other object. The student should figure out what the various classes and objects should be and what their instance variables and their methods should be. There is no single "correct" design; any reasonable design will do. The better the design, the easier the programming will be. Use the objects defined to play a game of Dominoes. The computer player must play legally, but does not need to play well. Try to make the computer player "smart," so the human player does not always win. Place all classes into one file. Output should look similar to below.

Sample Run: `partial`

```
Domino Menu

1. Print your dominoes
2. Add a domino to the head of the train
3. Add a domino to the tail of the train
4. Pick a domino from the boneyard
5. Pass your turn (only if bone yard is empty)
6. Print this menu
7. Quit

…

Please enter a choice: 2
Enter domino values: 3 5

The train (2): [5 | 3] [3 | 3]

System's turn.....System drew from the boneyard
System has 5 dominoes left
```

```
The train (2): [5 | 3] [3 | 3]

System has 1 dominoes left [2 | 0]

The train (9): [3 | 1] [1 | 2] [2 | 2] [2 | 4]
[4 | 5] [5 | 3] [3 | 3] [3 | 0] [0 | 1]

Your turn....(Enter 6 for menu)
Please enter a choice: 1

Your Dominoes (1): [5 | 5]

Your turn....(Enter 6 for menu)
Please enter a choice: 4

Choosing from boneyard...[2 | 5]

The train (9): [3 | 1] [1 | 2] [2 | 2] [2 | 4]
[4 | 5] [5 | 3] [3 | 3] [3 | 0] [0 | 1]

System's turn.....System drew from the boneyard
System has 2 dominoes left

The train (9): [3 | 1] [1 | 2] [2 | 2] [2 | 4]
[4 | 5] [5 | 3] [3 | 3] [3 | 0] [0 | 1]

…

System's turn....System played [0 | 2]
System has 0 dominoes left

The train (16): [0 | 2] [2 | 3] [3 | 1] [1 | 2]
[2 | 2] [2 | 4] [4 | 5] [5 | 3] [3 | 3]
[3 | 0] [0 | 1] [1 | 4] [4 | 0] [0 | 2]
[2 | 5] [5 | 5]

Oops!!...Computer won
Thank you for playing the game.
```

Name the program: `OODominoesXX.cpp`, where `XX` are your initials.