Lab 9 - Classes and Objects

Directions

- The labs are marked based on attendance and effort.
- It is your responsibility to ensure the TA records your progress by the end of the lab.
- Do each step of the lab in order.
- While completing these labs, you are encouraged to help your classmates and receive as much help as you like. Assignments, however, are individual work.
  You may not work on assignments during your lab time.
- If you complete the lab early, you should experiment with C++; however, you may leave if you prefer.
- If you do not finish the lab exercises during your lab time, you are encouraged to complete them later to finish learning the material. You will still receive full marks if you arrived on-time and put in your best effort to complete the lab.

1. Using a class: Cookies!

1. Create a new project for this lab, and copy the `cookie.cpp` file provided on the webpage into your project.
   - This file provides you a fully implemented `Cookie` class and space for you to complete a `main()` function to use the class. The `Cookie` class's member functions are implemented below `main()`.
   - **Do not edit the `Cookie` class during this lab.**
     Complete this section by adding/changing code in the `main()` function.
2. Start by asking the user what flavour of cookie they want.
   - Store the answer in a `string`. Capture the whole line of text using `getline()` because they may want a "deluxe chocolate chip with nuts" `Cookie`!
   - Your prompt may look like:
     ```
     Welcome, Cookie lover!
     Please enter the cookie flavour you would like to try.
     : Chocolate Chip
     ```
3. Instantiate the `Cookie` object and set the flavour.
   - Choose a good name for your `Cookie;` for example, `firstCookie.`
   - Read the `Cookie` class's interface (defined in the `class` code at the top of the file) for what public member function you should call to set the Cookie's flavour.
   - You may, if you like, read the implementation of the functions as well, which are at the bottom of the file; however, due to encapsulation, you should be able to just read the `class` section and be able to use it.

4. Output the current state of the cookie.
   - Use getter member functions to find out how much of the `Cookie` is left, its flavour, and display them to the screen. The getter functions will each **return** a value, you must then use `cout` to **output** those values to the screen.
   - Also use (call) the member function which **returns** a description, and then **output** that text to the screen as well.
   - Your new output may look like the following. Underlined text shows what came from the cookie's getter functions:
     ```
     You now have 100% of a Chocolate Chip cookie.
     Your cookie describes itself as: All of a Chocolate Chip cookie.
     ```
     - If you are having trouble getting the output to look exactly like the above, start by using `cout` to display the result of each getter, and get description function. Put each result on its own line to see exactly what value it gives you.
     - Based on understanding what each member function gives you, you should be able to construct the above output.
5. Create a loop which, while there is some cookie left, takes a bite out of the `Cookie`.
   - Continue looping while the percentage left of the cookie is > 0.
     Hint: What function tells you how much is left?
   - During the loop:
     - Take a bite out of the cookie (using one of the provided member function);
     - Display a message that you have eaten some of the cookie; and
     - Use the member function to get the full description, and print that to the screen.
   - Your new output may look like:
     ```
             Num Num! Cookies good!
     Cookie info after bite: Most of a Chocolate Chip cookie.
             Num Num! Cookies good!
     Cookie info after bite: Most of a Chocolate Chip cookie.
             Num Num! Cookies good!
     Cookie info after bite: A little of a Chocolate Chip cookie.
             Num Num! Cookies good!
     Cookie info after bite: A little of a Chocolate Chip cookie.
             Num Num! Cookies good!
     Cookie info after bite: NONE  of a Chocolate Chip cookie.
     ```


6. Now, create a second `Cookie` to show that we can work with two `Cookies` at once!
   - Near the top of `main()`, just after you create your first `Cookie`, create a second `Cookie` named `secondCookie`.
   - Initialize this cookie to the flavour `"Healthy oat and bran"`.
   - Output the description of this cookie once it's created.
   - At the end of the program, show the description again.
   - Your output may look like:

```
Welcome, Cookie lover!
Please enter the cookie flavour you would like to try.
: Chocolate Chip
You now have 100% of a Chocolate Chip cookie.
Your cookie describes itself as: All of a Chocolate Chip cookie.
Your other cookie describes itself as: All of a Healthy oat and
bran cookie.
        Num Num! Cookies good!
Cookie info after bite: Most of a Chocolate Chip cookie.
        Num Num! Cookies good!
Cookie info after bite: Most of a Chocolate Chip cookie.
        Num Num! Cookies good!
Cookie info after bite: A little of a Chocolate Chip cookie.
        Num Num! Cookies good!
Cookie info after bite: A little of a Chocolate Chip cookie.
        Num Num! Cookies good!
Cookie info after bite: NONE  of a Chocolate Chip cookie.
Your other cookie is: All of a Healthy oat and bran cookie.

No cookies were harmed in the making of this program... yet!
```

7. Try out your program with no flavour name (just press enter when asked for the flavour). Look at the implementation for the flavour setter function to see how this is handled.

8. **Understanding:**
   - How to understand the public interface for a simple class.
   - How to create an object which is an instance of an class.
   - How to call member functions of the object.
   - Understand that each object is independent of other objects of the same type (class). This is shown by eating one cookie does not change the other cookie.

2. Writing a Class: PetRock

1. Create a new program called `petRock.cpp`.
2. Create a class called `PetRock`. It's UML diagram is shown below. The following steps will guide you through implementing and testing the class.

PetRock
- size : int - mood : string
+ setSize(newSize : int) + getSize() : int + setMood(newMood : string) + getMood() : string + drop() : void + hug() : void + drawToScreen() : void

3. Implement the `size` member variable, and its getter and setter member functions.
   - In `PetRock`, add a private member variable of type `int`, called `size;`
   - In `PetRock`, implement as inline functions, the getter and setter methods corresponding to the two prototypes:
     ```
     void setSize(int newSize);
     int getSize();
     ```
4. Test the functions you have just implemented.
   - In the `main()` function, instantiate the `PetRock` as a variable called `rocky.`
   - Use the setter function to set `rocky`'s `size` to 6.
   - Use the getter function to output to the screen `rocky`'s current `size`.
     - Remember that the getter functions **return** a value; they don't **output** to the screen. You'll need to call the getter function from within a `cout` statement. For example:
       ```
       cout << "Rocky's size is: " << rocky.getSize() << endl;
       ```
   - Your output may look like this:
     ```
     Rocky's size is: 6
     ```
5. Implement the `PetRock mood` attribute, and its setter and getter functions.
   - Test your new attribute and functions. Use the mood `"Bouncy"`.
   - Your output should look like this:
     ```
     Rocky's mood is: Bouncy
     ```


6. Implement the `PetRock drawToScreen()` member function.
   - This function should draw a box of `'#'` signs on the screen.
     - Make box's height half the rock's `size;` the width equal to the `size.`
     - Use nested for loops.
   - After you have draw the rock-box, your `drawToScreen()` method should also output the rock's `mood`.
   - In `main(),` output a message, and then call the `drawToScreen()` function:
     ```
     cout << "Currently, he looks like: " << endl;
     rocky.drawToScreen();
     ```
   - Your new output, for a `PetRock` of size 6 might look like:
     ```
     Currently, he looks like:
     ######
     ######
     ######
     Pet rock is: Bouncy
     ```
7. Implement the `PetRock drop()` member function.
   - This function should change the rock's mood to being `"Unhappy"`.
   - Test this by having your code in `main()` do the following:
     - call the `drop()` function,
     - output a message stating you dropped him and then
     - redrawing rocky to the screen.
   - Your new output should look like:
     ```
     Oops! I dropped him:
     ######
     ######
     ######
     Pet rock is: Unhappy
     ```
8. Implement the `PetRock hug()` member function which is identical to `drop(),` but makes the rock `"Happy".`
   - Add more code (which mirrors the code for testing `drop()`).
   - Your new output may look like:
     ```
     But now I hugged him:
     ######
     ######
     ######
     Pet rock is: Happy
     ```
9. **Understanding:**
   - Describe how you would convert a UML diagram into a class.
   - Why do we not have to pass arguments to the drawToScreen() function? Where does it get its information from?

3. Expanding the PetRock

1. Add a new attribute to the `PetRock` class to store a `name`.
   - Add appropriate member functions to support the `name` attribute.
   - Modify the `drawToScreen()` function to display the name.
   - And modify your code in `main()` to demonstrate the new functionality you added.
2. Change the `drawToScreen()` function into a non-inline member function.
   - Give it a correct comment block once it's a non-inline function (provide the function implementation outside the `class` block.
   - Generally, only very short functions (one line of code) are left as inline functions.
3. **Understanding:**
   - For each member variable, describe what functions you would expect to in a class.
   - When moving a member-function from being inline to non-inline, what do you need to do?

4. Extra Challenges

- Change `drawToScreen()` so that the rock's shape changes based on its mood.
  - For example, if the mood is happy, draw it as a diamond; unhappy, draw it as a triangle; unhip, draw it as a square. You should support at least three different shapes (including the current rectangle). Be creative!
  - Update your `main()` function to test your shapes. What happens when you set the mood to a string which you are not explicitly checking for? (it should display something).
- Change the `PetRock` class so that all of the setter functions check the validity of the values being passed in.
  - For `size`, enforce the size to be between 1 and 50.
  - For `mood`, enforce that the new mood is one of a few "supported" moods.
  - Update the `main()` function to test each of your modified setters. Be sure to test as though looking for bugs, so try the boundary conditions.

5. Skills and Understanding

You should now be able to answer all the "understanding" questions in the previous sections. Complete the following to get credit for the lab:

- Show the TA the following:
  - Your operational programs which complete all of the above tasks.
  - The TA may ask you to explain any section of the lab.
- **Nothing** is to be submitted electronically or in hard-copy for this lab.

