Algorithmic Adventures: Into the Recursive Realms

Project 3 - The Tavern

It is now time to define the game setting. In this project you will implement the `Tavern` class, where your Characters will contend with each other.

The link to accept the GitHub Classroom assignment can be found on Blackboard

This project consists of two parts:

- ***You will modify the `Character` class***
- ***You will implement `Tavern`, as a subclass of `ArrayBag`*** that holds `Character` objects

---

Some additional resources

- ***Abstract Data Types:***
  - Geeks for Geeks

- Neso Academy
- ***Template Classes:***
  - CPP Manual
  - Geeks for Geeks
  - Tutorials Point
- ***Operator Overloading:***
  - CPP Manual
  - Geeks for Geeks
  - Programiz

Implementation

Work through the tasks sequentially (implement and test). Only move on to a task when you are positive that the previous one has been completed correctly. Remember that the names of classes and methods must exactly match those in this specification **(FUNCTION NAMES, PARAMETER TYPES, RETURNS, PRE AND POST CONDITIONS MUST MATCH EXACTLY)**.

Remember, you must thoroughly document your code!!!

Task 1: Modify the `Character` class

Define and implement the following additional *public* member functions:

```
/**
  @param      : A const reference to the right hand side of the == operator.
  @return     : Returns true if the right hand side character is "equal", false otherwise.                    Two
characters are equal if they have the same name, same race, same level                    and are either both an enemy
or not.
                NOTE: By this definition, only the aforementioned subset of the character's
attributes must be equal for two characters to be deemed "equal".

  Example: In order for character1 to be == to character2 we only need:
  - The same name
  - The same race
  - The same level
  - They must either be both an enemy or not
*/
operator==

/**
  @param     `: A const reference to the right hand side of the != operator.
  @return     : Returns true if the right hand side character is NOT "equal" (!=), false
```

```
                otherwise. Two characters are NOT equal if any of their name, race or level are
                not equal, or if one is an enemy and the other is not.
                NOTE: By this definition, one or more of the aforementioned subset of the
character's attributes only must be different for two characters to be
                deemed "NOT equal".
*/
operator!=

/**
  @post     : displays Character data in the form:
            "[name_] is a Level [level_] [race_]. \nVitality: [vitality_] \nMax Armor: [armor_]
             \n[They are / They are not] an enemy.\n"
*/
display
```

####

Task 2: Modify the `ArrayBag` class

Define and implement the following additional *public* member functions:

```
/** @param:   A const reference to another ArrayBag object
    @post:    Combines the contents from both ArrayBag objects, EXCLUDING duplicates.
    Example: [1, 2, 3] += [1, 4] will produce [1, 2, 3, 4]
*/
operator/=


/**
    @param:   A const reference to another ArrayBag object
    @post:    Combines the contents from both ArrayBag objects, including duplicates,
              adding items from the argument bag as long as there is space.
              Example: [1, 2, 3] += [1, 4] will produce [1, 2, 3, 1, 4]
*/
operator+=
```

Task 3: Implement the `Tavern` class as a subclass of `ArrayBag`

The `Tavern` is subclass of `ArrayBag` that stores `Character` objects

Data Types

The `Tavern` class **must** have the following *private member variables*:

```
- An integer sum of the levels of all the characters currently in the tavern
- An integer count of all the characters currently in the Tavern that are marked as enemies
```

The `Tavern` class **must** have the following *public member functions*:

Constructor

```
/**
    Default constructor.
    Default-initializes all private members.
*/
```

Unique Methods

```
/** @param:   A const reference to a Character entering the Tavern
    @return:  returns true if a Character was successfully added to the tavern (i.e. items_),                false
otherwise
    @post:    adds Character to the Tavern and updates the level sum and the enemy count
              if the character is an enemy.
 **/
enterTavern


/** @param:   A const reference to a Character leaving the Tavern
    @return:  returns true if a character was successfully removed from the tavern (i.e. items_),              false
otherwise
    @post:    removes the character from the Tavern and updates the level sum and the enemy count
              if the character is an enemy.
 **/
exitTavern



/**
    @return:  The integer level count of all the characters currently in the Tavern
 **/
getLevelSum



/**
    @return:  The average level of all the characters in the Tavern
    @post:    Computes the average level of the Tavern rounded to the NEAREST integer.
 **/
calculateAvgLevel



/**
    @return:  The integer enemy count of the Tavern
 **/
getEnemyCount
```

```
/**
    @return:  The percentage (double) of all the enemy characters in the Tavern
    @post:    Computes the enemy percentage of the Tavern rounded up to 2 decimal places.
 **/
calculateEnemyPercentage



/**
    @param:   A const reference to a string representing a character Race with value in
              ["NONE", "HUMAN", "ELF", "DWARF", "LIZARD", "UNDEAD"]
    @return:  An integer tally of the number of characters in the Tavern of the given race.
              If the argument string does not match one of the expected race values,
              the tally is zero. NOTE: no pre-processing of the input string necessary, only                  uppercase
input will match.
 **/
tallyRace


/**
@post:      Outputs a report of the characters currently in the tavern in the form:
            "Humans: [x] \nElves: [x] \nDwarves: [x] \nLizards: [x] \nUndead: [x] \n\nThe average             level is:
[x] \n[x]% are enemies.\n"
            Note that the average level should be rounded to the NEAREST integer, and the enemy             percentage
should be rounded to 2 decimal places.

            Example output:
            Humans: 3
            Elves: 5
            Dwarves: 8
            Lizards: 6
            Undead: 0

            The average level is: 7
            46.67% are enemies.
*/
tavernReport
```

Testing

How to compile with your **Makefile**:
In terminal, in the same directory as your *Makefile and your source files*, use the following command

```
make rebuild
```

This assumes you did not rename the `Makefile` and that it is the only one in the current directory.

You must always implement and test your programs **INCREMENTALLY!!!**
What does this mean? Implement and **TEST one method at a time**.
For each class:

- Implement one function/method and **test it thoroughly** (write a main file with multiple test cases + edge cases if applicable).

- Only when you are certain that function works correctly and matches the specification, move on to the next.
- Implement the next function/method and test in the same fashion.
  How do you do this? Write your own `main()` function to test your classes. Choose the order in which you implement your methods so that you can test incrementally: i.e. implement constructors then accessor functions, then mutator functions. Thoroughly test with valid and invalid input to check thet your function behaves as expected in each case. Pay special attention to edge cases. Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can **use stubs**: a dummy implementation that always returns a single value for testing. Don't forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Grading Rubric

Correctness 80% (distributed across unit testing of your submission)
Documentation 15%
Style and Design 5% (proper naming, modularity, and organization)

Important: You must start working on the projects as soon as they are assigned to detect any problems with submitting your code and to address them with us **well before** the deadline so that we have time to get back to you **before** the deadline.

There will be no negotiation about project grades after the submission deadline.

Submission:

We will grade the following :

`Character.hpp`
`Character.cpp`
`ArrayBag.hpp`
`ArrayBag.cpp`
`Tavern.hpp`
`Tavern.cpp`

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You MUST test and debug your program locally. To help you not rely too much on Gradescope for testing, we will only allow **5 submissions per day**. Before submitting to Gradescope you MUST ensure that your program compiles using the provided `Makefile` and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the "Programming Guidelines" document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you don't have through Gradescope. **"But it ran on my machine!" is not a valid argument for a submission that does not compile.** Once you have done all the above you submit it to Gradescope.

Due date:

This project is **due on October 13 at 11pm.**
No late submissions will be accepted.

Important

You must **start working on the projects as soon as they are assigned** to detect any problems and to address them with us **well before** the deadline so that we have time to get back to you **before** the deadline.
There will be no extensions and no negotiation about project grades after the submission deadline.

Help

Help is available via drop-in tutoring in Lab 1001B (see website for schedule). You will be able to get help if you start early and go to the lab early. We only have 2 UTAs in the lab, **the days leading up to the due date will be crowded and you will not be able to get much help then**.

Authors: Georgina Woo, Tiziana Ligorio