Algorithmic Adventures: Into the Recursive Realms

Project 1 - The **Character** class: A Review of OOP

This semester we will have some fun setting up our very own role playing game (inspired by the Divinity series). We hope you'll enjoy this adventure as much as we did setting it up for you!

This is a baseline project whose objective is to **get you acquainted with the platforms we will use in this course** and to refresh your knowledge of **basic OOP**. You will **implement the `Character` class.** In order to successfully complete this project, we strongly recommend that you look back to your CSCI 135 coursework as a reference.

First of all, this project will introduce you to **GitHub Classroom** so you can work with git version control. All projects in this course will be distributed via GitHub Classroom and submitted to Gradescope via GitHub. We truly hope you will start establishing **best practices of version control, INCREMENTAL coding, testing and debugging** (i.e. code, test and debug one function at a time); you will need it in the near future, so better start now!

Part 1 - getting started with GitHub Classroom:

- If you don't already have one, go to https://github.com/ and create a GitHub account. You will likely use your GitHub account professionally in the future, so choose a username you will want to keep.
- Next, watch this video to brush-up on or learn the basics of git and GitHub:
  https://www.youtube.com/watch?v=MJUJ4wbFm_A
- For this project we will use GitHub Classroom. The following video will guide you through the entire process - from accepting an assignment to submitting your solution:
  https://www.youtube.com/watch?v=AHDCokfgcSo
  - *Although the video is about a different course, the instructions are the same (with different repo and file names). The only difference is that we will not add a distribution branch, so you can ignore the part where it says to execute the two git commands in the readme file; there are not extra instructions in the readme file on our repo).*

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

The above video will also show you how to submit to Gradescope via GitHub. Make sure to refer back to these instructions when it's time to submit.

Some additional resources if you need to brush up on basic OOP

- Code Beauty on constructors and class methods
- thenewboston on classes and objects
- McProgramming on .hpp and .cpp files

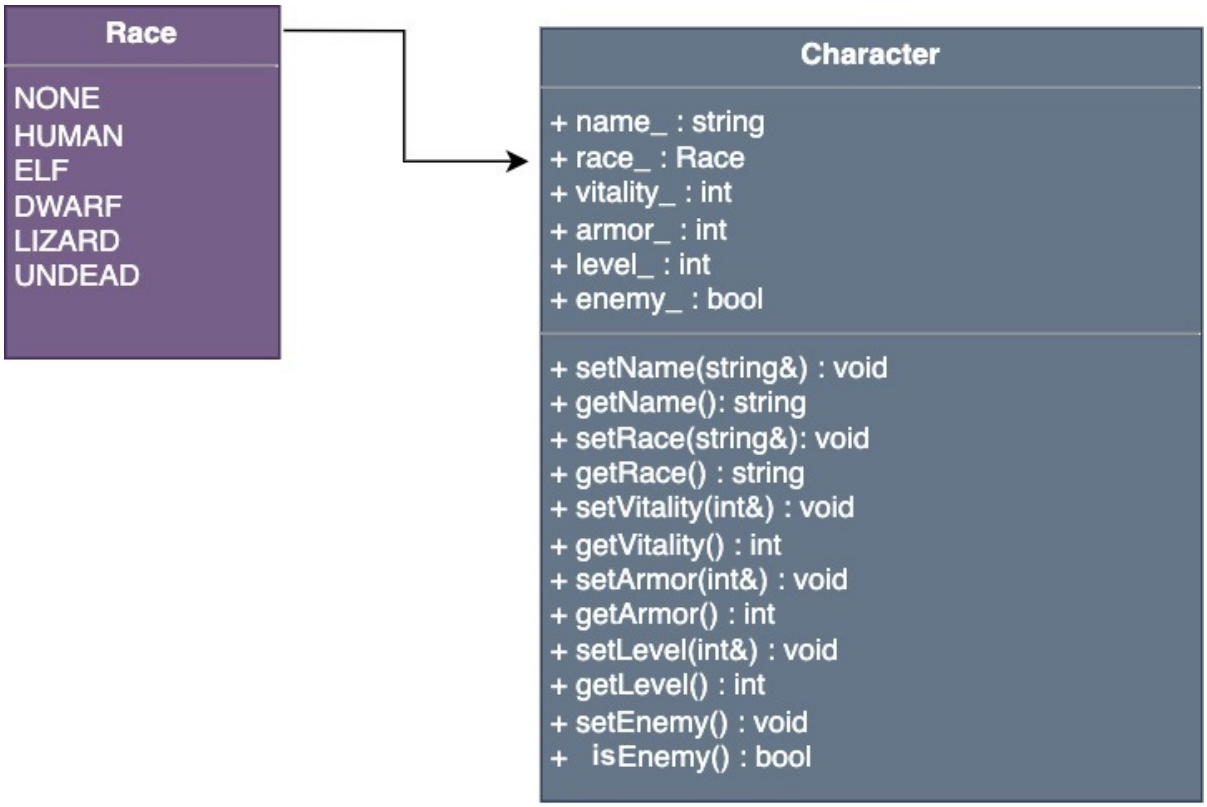
Part 2 - The `Character` Class :

You will implement the `Character` class.

You must always separate interface from implementation(Character.hpp and Character.cpp), and you ONLY EVER include a class' interface (.hpp) This will be an implicit assumption in this course going forward. 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)**. This class has only accessor and mutator functions for its public data members. Recall that accessor functions (e.g. `getName()`) are used to access the private data members (e.g. all `getName()` will do is return `name_`, the private data member) and are therefore declared `const`, while mutator functions (e.g. `setName()`) give a value to the data members, and do not modify it's parameters, which will be passed by `const` reference.

Remember, you must thoroughly document your code!!!

Task: The Character class



Every Character has a Name, Race, Health, Armor, Level, and a boolean if the character is an enemy.

The Character class must define the following types:

- An **enum** named `Race` with values `{NONE, HUMAN, ELF, DWARF, LIZARD, UNDEAD}`

The Character class must have the following *private member variables*:

```
private:
- The name of the character (a string in UPPERCASE)
- The race of the character (an enum)
- The character's vitality (a non-negative integer)
- The character's max armor level (a non-negative integer)
- The character's level (a non-negative integer)
- A flag indicating whether the character is an enemy
```

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

Constructors

```
/**
     Default constructor.
     Default-initializes all private members.
     Default character name: "NAMELESS".
     Booleans are default-initialized to False.
     Default enum value: NONE.
     Default Vitality, Max Armor, and Level: 0.
  */
```

```
 /**
 Parameterized constructor.
 @param      : The name of the character (a string in UPPERCASE)
 @param      : The race of the character (a string)
 @param      : The character's vitality (a non-negative integer) , with default
               value 0
 @param      : The character's max armor level (a non-negative integer), with
               default value 0
 @param      : The character's level (non-negative integer), with default value 0
 @param      : A flag indicating whether the character is an enemy, with default
               value false
 @post       : The private members are set to the values of the corresponding
               parameters.
 */
```

Hint: Notice the default argument in the parameterized constructor.

Accessors (get) and Mutators (set)

```
   /**
            @param  : the name of the Character
            @post   : sets the Character's title to the value of the parameter,
                      in UPPERCASE. Only alphabetical characters are allowed.
                      For example, attempting to create a character named
                      "TW3EDLEDUM2" should create a character named "TWEDLEDUM".
                    : If the given parameter does not have any valid alphabetical
                      characters, the character's name should be set to "NAMELESS".
*/
 setName


  /**
      @return : the name of the Character
  */
 getName


   /**
       @param  : the race of the Character (a string)
       @post   : sets the Character's race to the value of the parameter.
                 If the given race was invalid, set race_ to NONE.
  */
 setRace


  /**
      @return : the race of the Character (a string)
   */
 getRace


  /**
    @param  : an integer vitality
    @pre    : vitality >= 0 : Characters cannot have negative health
```

```
   @post   : sets the vitality private member to the value of the parameter
             if vitality is negative, do nothing.
 */
setVitality


 /**
     @return : the value stored in vitality_
  */
getVitality

 /**
   @param  : an integer armor level
   @pre    : armor >= 0 : Characters cannot have negative armor
   @post   : sets the armor private member to the value of the parameter
             if armor is negative, do nothing.
 */
setArmor


 /**
     @return : the value stored in armor_
  */
getArmor

 /**
   @param  : an integer level
   @pre    : level >= 0 : Characters cannot have a negative
   @post   : sets the level private member to the value of the parameter
             if level is negative, do nothing.
 */
setLevel


 /**
     @return : the value stored in level_
  */
getLevel



 /**
  @post   : sets the enemy flag to true
 */
setEnemy


  /**
   @return true if the character is an enemy, false otherwise

   Note: this is an accessor function and must follow the same convention as all accessor
         functions even if it is not called getEnemy
  */
  isEnemy
```

Submission:

You will submit your solution to Gradescope via GitHub Classroom (see video linked above). The autograder will grade the following files only:
`Character.hpp Character.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.

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 (we will provide one with this first project, but you can always add to it). In this course we will not grade your test program, but you must always write one 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. 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 Rubrics

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

Due date:

This project is **due on September 8 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