Algorithmic Adventures: Into the Recursive Realms

Project 2 - Pick your role: Inheritance

You will now define different roles for your character. You will implement **4 new classes that derive from the `Character` class** but also have additional attributes and methods specific to their type:

- **Mages** are scholars of magic and spells, and are able to manipulate the world around them with elemental magic. Each Mage has an affinity to a specific *school of magic*, a choice of *weapon*, and an optional choice of summoning an *incarnate*.
- **Scoundrels** live by their wits, take chances when they must, and land on their feet against all odds. Every Scoundrel has a set of *daggers* made from a certain *material* (e.g. wood, bronze, iron, steel, mithril, adamant, and rune) *faction* that they belong to: Cutpurse, Nightblades, or Silvertongue (e.g, thieves, assassins, and swindlers). Scoundrels may or may not have a *disguise*.

- **Rangers** specialize in hunting the monsters that threaten the edges of civilization. Each Ranger is equipped with a *crossbow* that fires from a limited source of *ammunition*. Arrows can come with different effects, so a ranger's ammunition source includes a range of different *arrows*. A Ranger may or may not be able to recruit an *animal companion*. Each Ranger also has a list of *affinities and advantages*, which increases the damage dealt with an arrow of the same element.
- **Barbarians** are the archetypal warriors who use brute strength and raw fury to excel in combat. A Barbarian can wield either a *two-handed weapon*, or a *one-handed weapon* and a *shield*. They may or may not be *enraged*, which allows them to deal double damage.

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

---

Some additional resources on Inheritance, enums and struct

- Code Beauty on inheritance programming
- Caleb Curry on base and subclass inheritance
- enums
- struct

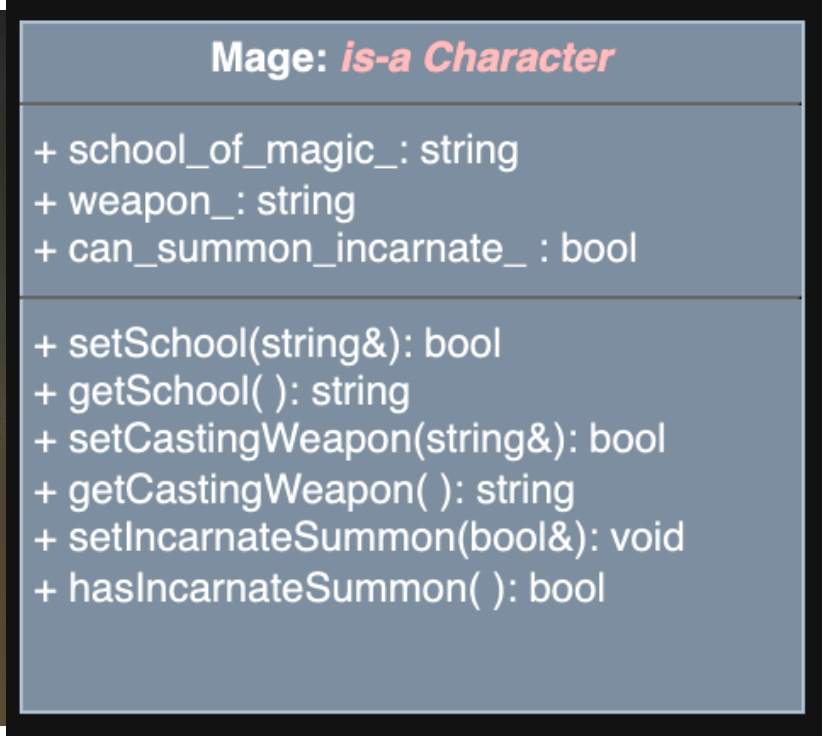
Implementation

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: Sublcasses

The `Mage` class



Data Types

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

```
- A string that represents their school of magic
- A string that represents their weapon
- A boolean indicating if they are able to summon an incarnate
```

Constructors

```
/**
  Default constructor.
  Default-initializes all private members.
  Default character name: "NAMELESS". Booleans are default-initialized to False.
  Default school of magic and weapon: "NONE".
*/


/**
  Parameterized constructor.
  @param      : The name of the character (a const string reference)
  @param      : The race of the character (a const string reference)
  @param      : The character's vitality (an integer). Default to 0
  @param      : The character's max armor level (an integer). Default to 0
  @param      : The character's level (an integer). Default to 0
  @param      : A flag indicating whether the character is an enemy. Default to false.
  @param      : The character's school of magic (a const string reference). Valid schools:
                [ELEMENTAL, NECROMANCY, ILLUSION].
                String inputs can be in lowercase, but must be converted to
                uppercase when setting the variable.
                If the school name is invalid, set it to "NONE"
```

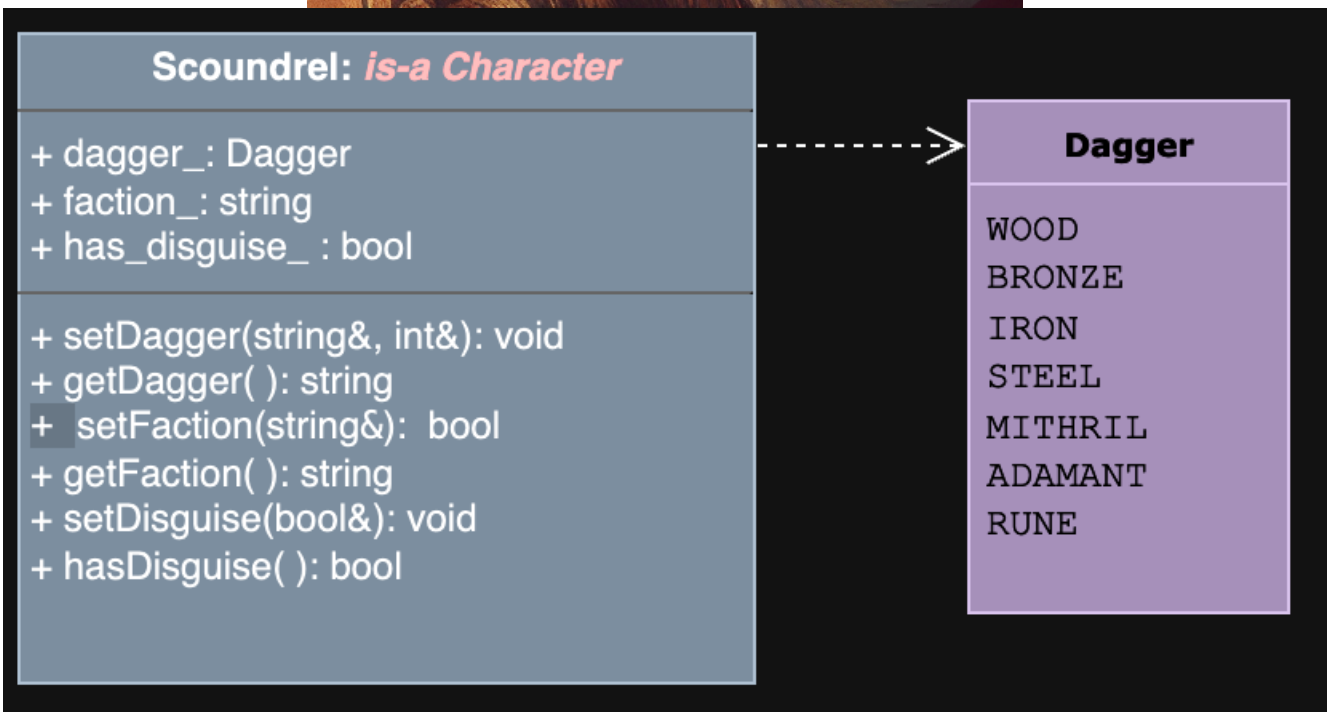
```
  @param      : The character's choice of weapon (a const string reference).
                Valid weapons: [WAND, STAFF]
                String inputs can be in lowercase, but must be converted to
                uppercase when setting the variable.
                If the weapon is invalid, set it to "NONE"
  @param      : A flag indicating whether the character is able to summon an
                incarnate. Default to false.
  @post       : The private members are set to the values of the corresponding
                parameters.
                REMEMBER: If the school of magic or weapon is not provided or invalid, the
                variables should be set to "NONE".
*/
```

Unique methods

```
/**
    @param  : a reference to a string representing the school of magic
    @post   : sets the private member variable to the value of the parameter. If
              the provided school of magic is not one of the following: [ELEMENTAL,
              NECROMANCY, ILLUSION], do nothing and return false.
              String inputs can be in lowercase, but must be converted to
              uppercase when setting the variable.
    @return  : true if setting the variable was successful, false otherwise.
**/
setSchool


/**
  @return  : the string indicating the character's school of magic
**/
getSchool


/**
    @param  : a reference to a string representing the character's weapon
    @post   : sets the private member variable to the value of the parameter.
              String inputs can be in lowercase, but must be converted to
              uppercase when setting the variable.
              If the provided weapon is not one of the following:
              [WAND, STAFF], do nothing and return false.
    @return  : true if setting the variable was successful, false otherwise.
**/
setCastingWeapon


/**
  @return  : the string indicating the character's weapon
**/
getCastingWeapon

/**
  @param  : a reference to boolean
  @post   : sets the private member variable indicating whether the character can
            summon an incarnate
**/
setIncarnateSummon
```

```
/**
  @return  : the summon-incarnate flag
**/
hasIncarnateSummon
```

The `Scoundrel` class



Data Types

The `Scoundrel` class **must** have the following *type*:

```
- an enum Dagger with the values {WOOD, BRONZE, IRON, STEEL, MITHRIL, ADAMANT, RUNE}
```

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

```
- A dagger
- A string that represents their Faction
- A boolean indicating if they have a disguise
```

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

Constructors

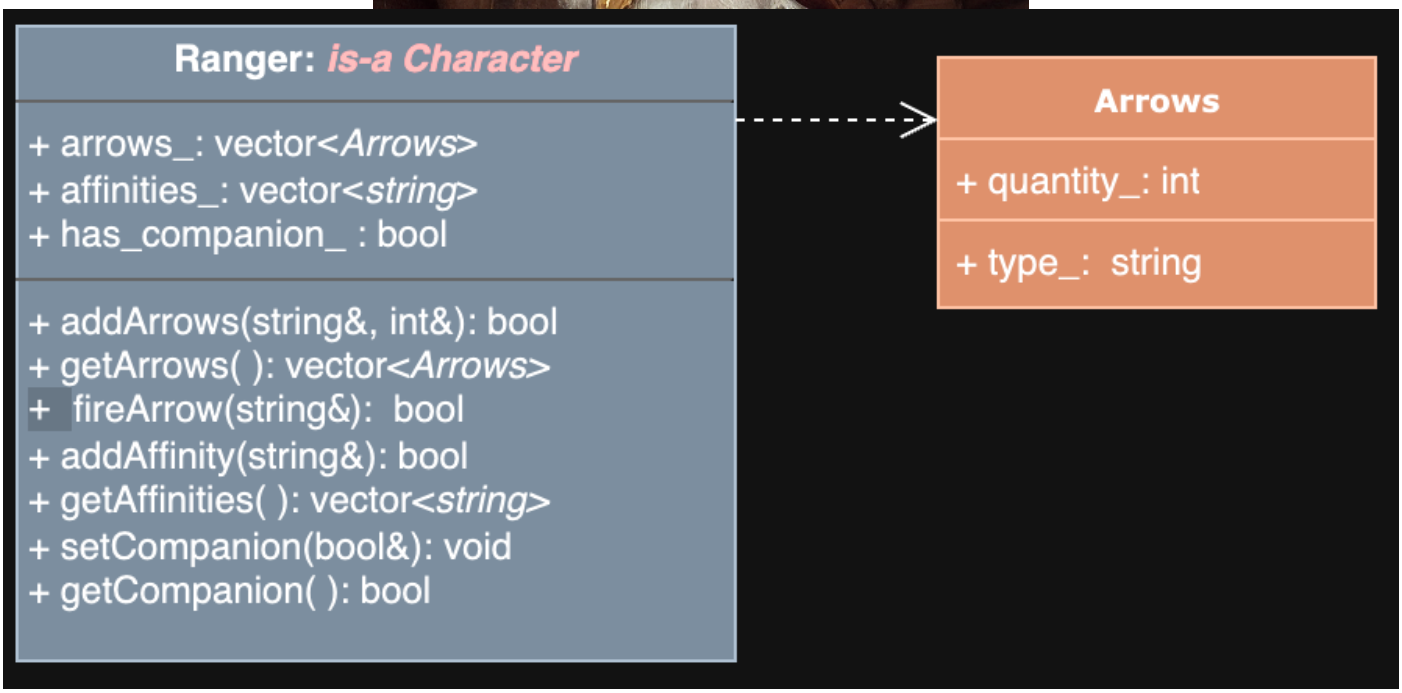
```
/**
    Default constructor.
    Default-initializes all private members.
    Default character name: "NAMELESS". Booleans are default-initialized to False.
    Default dagger: WOOD. Default faction: "NONE".
*/


/**
    Parameterized constructor.
    @param      : The name of the character (a const string reference)
    @param      : The race of the character (a const string reference)
    @param      : The character's vitality (an integer). Default to 0
    @param      : The character's max armor level (an integer). Default to 0
    @param      : The character's level (an integer). Default to 0
    @param      : A flag indicating whether the character is an enemy. Default to false.
    @param      : The character's dagger type (a const string reference).
                  String inputs can be in lowercase, but must be converted to
                  uppercase when setting the dagger enum. Default to WOOD
    @param      : The character's Faction (a const string reference).
                  Valid Factions: [CUTPURSE, SHADOWBLADE, SILVERTONGUE]
                  String inputs can be in lowercase, but must be converted to
                  uppercase when setting the variable. Default to "NONE"
    @param      : A flag indicating whether the character has a disguise. Default to false
    @post       : The private members are set to the values of the corresponding
                  parameters.
                 If the dagger type is not provided or invalid, the variable should
                  be set to WOOD.
                 If the Faction is not provided or invalid, the variable should be
                  set to "NONE".
*/
```

Unique Methods

```
/**
   @param     : a reference to a string representing the dagger type
   @post      : sets the private member variable to the value of the parameter.
                String inputs can be in lowercase, but must be converted to
                uppercase when setting the variable.
```

```
                If the dagger type is not valid (i.e, is one of the following:
                [WOOD, BRONZE, IRON, STEEL, MITHRIL, ADAMANT, RUNE]),
                the variable should be set to WOOD.
**/
setDagger



/**
  @return     : the string indicating the character's dagger type
**/
getDagger


/**
    @param    : a reference to a string representing the character's Faction
    @post     : sets the private member variable to the value of the parameter.
                String inputs can be in lowercase, but must be converted to
                uppercase when setting the variable.
                If the provided faction is not one of the following:
               [NONE, CUTPURSE, SHADOWBLADE, SILVERTONGUE],
               do nothing and return false.
    @return   : true if setting the variable was successful, false otherwise.
**/
setFaction



/**
  @return  : the string indicating the character's Faction
**/
getFaction



/**
  @param  : a reference to boolean
  @post   : sets the private member variable indicating whether the character has a disguise
**/
setDisguise


/**
  @return  : the visual aid flag
**/
hasDisguise
```

The `Ranger` class



Data Types

The `Ranger` class **must** have the following *type*:

```
struct Arrows
{
    string type_;
    int quantity_;
};
```

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

```
- A vector of arrows
- A vector of affinities (strings)
- A boolean indicating if they are able to recruit an animal companion
```

Constructors

```
/**
    Default constructor.
    Default-initializes all private members. Default character name: "NAMELESS".
    Booleans are default-initialized to False.
*/


/**
   Parameterized constructor.
  @param      : The name of the character (a const string reference)
  @param      : The race of the character (a const string reference)
  @param      : The character's vitality (an integer). Default to 0
  @param      : The character's max armor level (an integer). Default to 0
  @param      : The character's level (an integer). Default to 0
  @param      : A flag indicating whether the character is an enemy. Default to false
  @param      : A vector of arrows. Valid arrow types are: [WOOD, FIRE, WATER, POISON, BLOOD]
                Lowercase valid arrow types are retained but converted to uppercase.
                Invalid arrows are those with non-positive quantities or invalid types.
                If the vector contains invalid arrows, those arrows are discarded.
                Default to empty vector
  @param      : A vector of affinities. Valid Affinities: [FIRE, WATER, POISON, BLOOD]
                String inputs can be in lowercase, but must be converted to uppercase.
                If the vector contains invalid affinities, those affinities are discarded.
                Default to empty vector
  @param      : A flag indicating whether the character is able to recruit an animal companion.
                Default to false
  @post       : The private members are set to the values of the corresponding parameters
*/
```

Unique Methods

```
/**
  @return     : a vector of the Character's arrows
**/
getArrows

/**
    @param    : a reference to string representing the arrow type
    @param    : a reference to an integer quantity
    @post     : If the character already has that type of arrow, the quantity in the vector
                is updated. If not, the arrow is added to the vector.
                Valid arrow types are: [WOOD, FIRE, WATER, POISON, BLOOD]
                Lowercase valid arrow types are retained but converted to uppercase.
                Quantity of arrows must be greater than 0
                Invalid arrows are those with non-positive quantities or invalid types.
```

```
                If the arrows are invalid, they are not added.
    @return   : True if the arrows were added successfully, false otherwise
**/
addArrows

/**
    @param    : a reference to string representing the arrow type
    @post     : If the character has the listed arrow AND enough arrows to fire one,
                the quantity of remaining arrows in the vector is updated.
    @return   : True if the character had the listed arrow AND enough arrows, False otherwise.
**/
fireArrow

/**
    @param  : a reference to string representing an affinity
    @post   : If the affinity does not already exist in the vector, add it to the vector.
              Valid Affinities: [FIRE, WATER, POISON, BLOOD]
              String inputs can be in lowercase, but must be converted to uppercase when
              setting the variable.
              There should be no duplicate affinities.
              If the affinity is invalid, it is NOT added.
    @return : True if the affinity was added successfully, false otherwise
**/
addAffinity


/**
  @return     : a vector of the Character's affinities
**/
getAffinities

/**
    @param    : a reference to a boolean indicating whether the character is able to recruit
                an animal companion
    @post     : sets the private member variable to the value of the parameter.
**/
setCompanion

/**
    @return   : a boolean indicating whether the character is able to recruit an animal companion
**/
getCompanion
```

##

Note: This class uses a **struct** type. Struct is just short for "structure" and it is a user-defined type, much like a class. It is not a full fledge class, however. We use it to bundle data items with no or very few functions (non in this case). When initializing a new review you may use `review` as the data type. For example for `int x`, `int` is our primitive data type. If we want to make a review called x we would write `review x`.

- More on structs here: https://www.educative.io/edpresso/what-is-a-cpp-struct

##

The `Barbarian` class

Barbarian: *is-a Character*

+ main_weapon_: string
+ secondary_weapon_: string
+ enraged_ : bool

+ setMainWeapon(string&): bool
+ getMainWeapon( ): string
+ setSecondaryWeapon(string&): bool
+ getSecondaryWeapon( ): string
+ setEnrage(bool&): void
+ getEnrage( ): bool
+ toggleEnrage( ): void

Data Types

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

```
- A string representing their main weapon
- A string representing their offhand weapon
- A boolean indicating if they are enraged
```

Constructors

```
/**
    Default constructor.
    Default-initializes all private members. Default character name: "NAMELESS".
    Booleans are default-initialized to False.
    Default weapons: "NONE".
*/


/**
    Parameterized constructor.
    @param      : The name of the character (a const string reference)
    @param      : The race of the character (a const string reference)
    @param      : The character's vitality (an integer). Default to 0
    @param      : The character's max armor level (an integer). Default to 0
    @param      : The character's level (an integer). Default to 0
    @param      : A flag indicating whether the character is an enemy. Default to false
    @param      : The character's main weapon (a string).
                  String inputs can be in lowercase, but must be converted to uppercase
                  when setting the variable. Only alphabetical characters are allowed.
                  Default to "NONE"
    @param      : The character's offhand weapon (a string).
```

```
                String inputs can be in lowercase, but must be converted to uppercase
                when setting the variable. Only alphabetical characters are allowed.
                Default to "NONE"
    @param      : A flag indicating whether the character is enraged. Default to false
    @post       : The private members are set to the values of the corresponding parameters.
                : If the main and secondary weapons are not provided or invalid, the variables                    are set
to "NONE".
*/
```

Unique Methods

```
/**
    @param    : a reference to a string representing the Character's main weapon
    @post     : sets the private member variable to the value of the parameter.
                Only alphabetical characters are allowed.
                String inputs can be in lowercase, but must be converted to uppercase when
                setting the variable.
                If the given input is invalid (i.e contains numbers), do nothing and return false
    @return   : true if setting the variable was successful, false otherwise.
**/
setMainWeapon


/**
    @return   : a string of the Character's main weapon
**/
getMainWeapon

/**
    @param    : a reference to a string representing the Character's secondary weapon
    @post     : sets the private member variable to the value of the parameter.
                Only alphabetical characters are allowed.
                String inputs can be in lowercase, but must be converted to uppercase when
                setting the variable.
              : If the given input is invalid (i.e contains numbers), do nothing and return false
    @return   : true if setting the variable was successful, false otherwise.
**/
setSecondaryWeapon


/**
   @return    : a string of the Character's secondary weapon
**/
getSecondaryWeapon

/**
  @param      : a reference to a bool
  @post       : sets the private member variable to the value of the parameter.
**/
setEnrage


/**
  @return  : a boolean of whether the Character is enraged
**/
getEnrage
```

```
/**
  @post   : sets the enraged variable to the opposite of what it was
**/
toggleEnrage
```

Task 2: Testing

To give you some feel and guide into testing, we have added a testing task to this project. To be clear, the task below is not a thorough test of your classes. You must test every function you implement with valid and invalid values, and make sure it behaves as specified. This is only intended to motivate you into building a methodical and incremental testing practice.

Write a testing program as described below and name the file `test.cpp`.
Instantiate the following character objects that derive from the Character class. For each character object, print out the character's information - including private member variables that are unique to their subclass - using the appropriate getter functions in the format: "[NAME] is a Level [LEVEL] [RACE]. \nVitality: [VITALITY] \nArmor: [ARMOR] \nThey are [an enemy/not an enemy]\n[SUBCLASS ATTRIBUTE] : [SUBCLASS ATTRIBUTE VALUE]...",

where \n[SUBCLASS ATTRIBUTE] : [SUBCLASS ATTRIBUTE VALUE]... means that you add any additional subclass-specific variable name and its value after a colon, each on a new line.

```
2.1.1 Instantiate a default Mage
  - Sets its name, race, vitality, and armor levels using the appropriate setter functions

2.1.2 Instantiate a Mage with the parameterized constructor with the following character details:
Name: SPYNACH
Race: ELF
Level: 4
Vitality: 6
Enemy: FALSE

2.1.3 Set SPYNACH's unique private member variables to the following:
School of Magic: Illusion
Weapon: Wand
Summon Incarnate: FALSE

2.2.1 Instantiate a default Scoundrel
  - Sets its name, race, vitality, and armor levels using the appropriate setter functions

2.2.2 Instantiate a Scoundrel with the parameterized constructor with the following character details:
Name: FLEA
Race: DWARF
Level: 7
Vitality: 12
Enemy: FALSE

2.2.3 Set FLEA's unique private member variables to the following:
Dagger: Adamant
Faction: Cutpurse
Disguise: TRUE
```

```
2.3.1 Instantiate a default Ranger
  - Sets its name, race, vitality, and armor levels using the appropriate setter functions

2.2.2 Instantiate a Ranger with the parameterized constructor with the following character details:
Name: MARROW
Race: UNDEAD
Level: 6
Vitality: 9
Enemy: TRUE

2.2.3 Set MARROW's unique private member variables to the following:
Vector of arrows: Wood, 30, Fire, 5, Water, 5, Poison, 5
Affinities: Fire, Poison
Animal Companian: TRUE

2.4.1 Instantiate a default Barbarian
  - Sets its name, race, vitality, and armor levels using the appropriate setter functions

2.4.2 Instantiate a Barbarian with the parameterized constructor with the following character details:
Name: BONK
Race: HUMAN
Level: 5
Vitality: 11
Enemy: TRUE

2.4.3 Set BONK's unique private member variables to the following:
Main Weapon: MACE
Offhand Weapon: ANOTHERMACE
Enraged: TRUE
```

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 :

`Barbarian.hpp`
`Barbarian.cpp`
`Mage.hpp`
`Mage.cpp`
`Ranger.hpp`
`Ranger.cpp`
`Scoundrel.hpp`
`Scoundrel.cpp`
`test.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 September 22 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