CS162 Assignment 3

Shoshana Abrass
abrasss@onid.oregonstate.edu
Feb 15, 2015

Discovering requirements

The requirements were given in the assignment, except for the choices and reflections documented below.

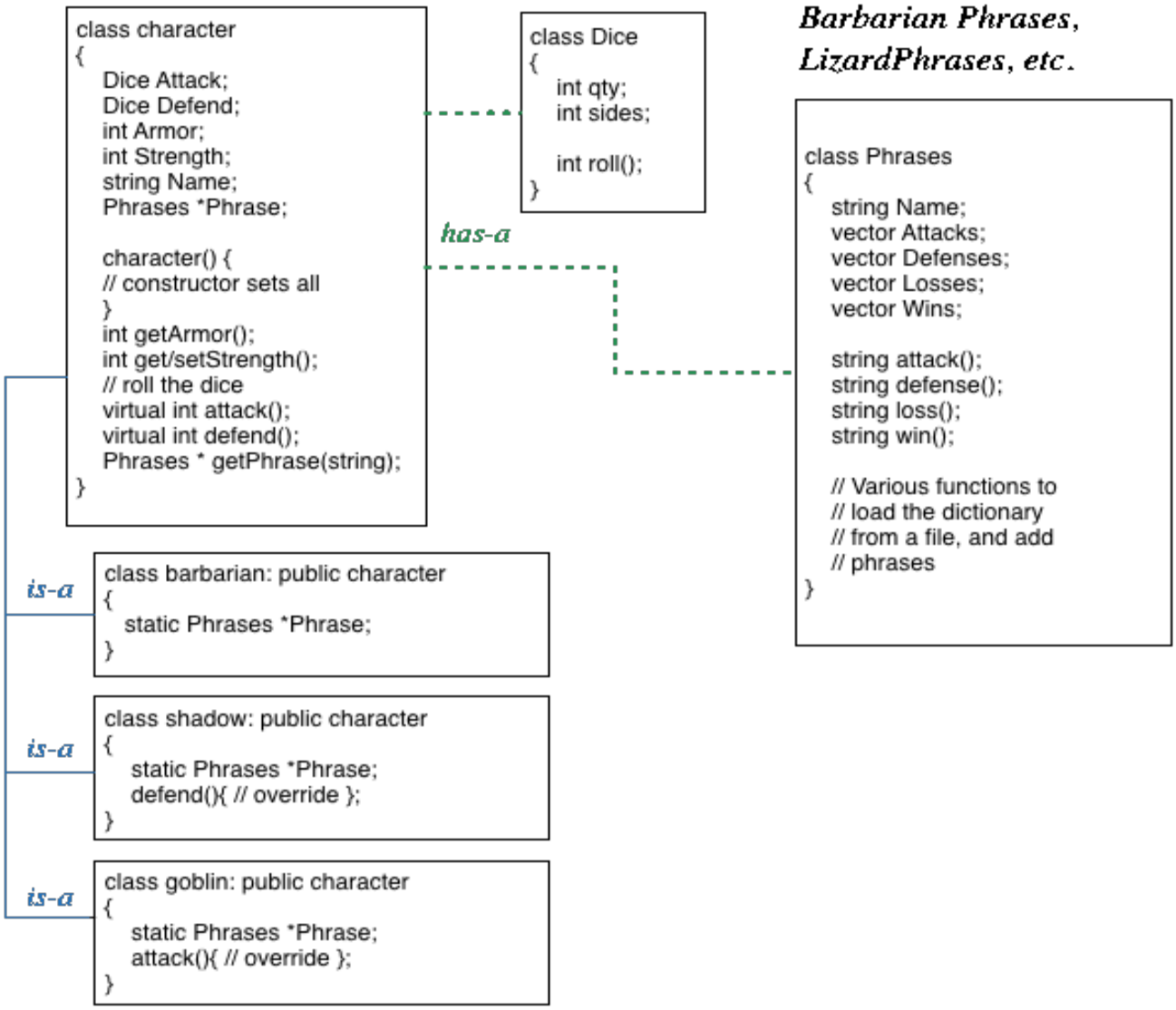
Design

Character class and derivatives

The basic class design here is straightforward. I'm confident that Dice, Character and the character derivatives are good although they may not be complete. In particular, I can see that for debugging or 'admin access' it would be useful to print all the character's statistics out, but I haven't put a full set of getters into the design.

As an extension to the assignment I was thinking about adding custom 'phrases' to each character type, so instead of printing "The Barbarian attacks. The Barbarian rolls a 9!", the program could print, "The Barbarian swings her war hammer. The Lizard is crushed!". This is an interesting challenge and I haven't finalized the design for how the different Phrase dictionaries will be identified and addressed. Right now the design calls for a static class member that will store the pointers to the various dictionaries, since I want them to be shared (ie, all barbarian instances will use the same pointer to the BarbarianPhrase dictionary). **`getPhrasePtr()`** will load the dictionary if the static pointer is NULL (so the first instance of each class will load that class' dictionary). I believe the character constructor may need to take an argument to the dictionary Name, but I'm not certain. However I'm happy with the interface between characters and phrases: ***`Character`*`.Phrase.Attack()`** will return an attack phrase from that character's dictionary. There may be a future need to support variant phrases for plural forms, although we won't use this in the first version.

I'm going to build out the basic class structure and gameplay drivers while I continue to think about the dictionary design.



test.cpp

Define character1, character2
do
    OneRound(character1, character2);
    OneRound(character2, character1);
until (one character has zero strength)
Declare a winner

// It's not yet clear whether we'll need to let the user choose characters;
// whether we'll need to support more than two characters;

// whether we'll want to allow for retreat; etc.

Reflection

- I originally planned to initialize the Dice by calling the constructor from inside the Character constructor; this doesn't work because the default constructor for Dice is automatically invoked, so I needed a public Dice::set method.
- I created a Phrases class as part of the design, but didn't implement it, since I don't know what the future requirements of this program will be
- In the design the Combat() function is shown as OneRound() function, but I decided to move both attack and defense rounds inside a common function. The Combat() function as implemented looks more like this:

```
Combat (Character &PlayerOne, Character &PlayerTwo)

 <flip a coin to see who goes first>
 while (both characters have strength > 0)
 {
        damage = attack - (defend+armor)
        <switch which character is attacking>
 }
```

- Because of the Goblin-vs-Goblin special case, I needed to be able to identify at least one sub-class of Character. The typeid() function isn't intended to be used this way, so I created an Character::Type member variable, as well as an "xfactor" variable which changes value when the Goblin has rolled a 12, causing the effect of an achilles injury (dividing attack points in half) if the opponent is not also a Goblin
  This is annoying since it breaks encapsulation a little, but since I assume we'll be given more requirements later, I still opted to leave Combat() where it is for now until I know where this is going.
- The biggest design decision in this program is where to put the Combat functionality. I decided on the simplest option, putting it inside main and passing both Character objects to it as references:
  `Combat (Character &PlayerOne, Character &PlayerTwo)`
  However there are also good arguments for having this be a Character method, where one of the players would call the function with the other player as an argument:
  `PlayerOne.Combat(Character &PlayerTwo)`
  This might be useful if the inter-player logic becomes more complicated. It would allow me to keep the goblin-vs-gobin logic - and any other special casing - encapsulated inside the class.
- If we need to add weapons to our characters, this can be done by creating a Weapon class and giving each Character a vector or array of Weapons. The effect of these can then be added to the results of the attack() function as needed. Again, if there are special cases such as "arrows don't work against goblins", I'll need to re-consider whether Combat() should be a member function.
- After creating a simple main loop in test.cpp, I added a few functions to fully automate testing. The results are shown below.

Testing

NB: I've written test code for testing the Combat() function with various Character subclasses. Type

```
make test
```

to compile and run these tests.

Percentage that this | character \V	wins against this character ----> Goblin	Barbarian	Reptile	BlueChix	Shadow
Goblin	49.95%	32.80%	0.02%	0.00%	8.26%
Barbarian	67.10%	49.80%	0.00%	0.00%	3.60%
Reptile	99.90%	100.00%	49.90%	12.19%	88.00%
BlueChix	99.90%	100.00%	87.80%	50.05%	91.90%
Shadow	91.70%	96.30%	11.90%	8.00%	49.90%