Benjamin R. Olson, May 15, 2014
CS 162 – Introduction to Computer Science II
Instructor: Terry Rooker

Project 4 Design: Modification and Extension of Project 3, Combat Game

Modifications to the Original:

1. The parent class, character, will have at least one pure virtual function, modifyRoll() in order to make it an abstract class. The character class will also have virtual function calcDamage().
2. I need to move calcDamage() into the character class and change the formal parameter list as follows:
Old function declaration: void calcDamage(character &a, character &b, int attack, int defense);
New function declaration: virtual void character::calcDamage(attacker &a, int attack_roll) = 0;
implementation guidelines:
    - calcDamage() is to be called on the defending object
    - local variables: int defense_roll
    - reference to the attacking object is passed in via the attacker &a argument in case the defender's roll is greater than the attacker's roll (so the attacker may possibly lose its own strength points in this case)
    - The new function will call rollDice() and modifyRoll(). The call to modifyRoll() demonstrates polymorphism.
3. I will move rollDice() into the character class and change it as follows:
Old function: int rollDice(character &c, bool attacker)
New function: int character::rollDice(bool attacker), with basically the same implementation details
4. I will add a pure virtual function, modifyRoll(int delta), which will be implemented differently in each subclass (see details of #5).

5. Each subclass of character will be modified by overriding the virtual function calcDamage() in the parent class, with the following class hierarchy as an example:

```
character ( contains virtual function calcDamage() )
        reptile ( overrides calcDamage() )
                dragon ( overrides calcDamage() of the reptile class )
```

Details of subclass modifications, where modifyRoll() is defined differently in each:

- **reptile**: calcDamage() incurs less damage than the corresponding calculation in the parent class would incur, according to the **tailWhip()** function added, which does the following:
    - If dragon is attacking, tailWhip() adds 2 to the dragon's roll. If dragon is defending, tailWhip() adds 3 to the dragon's roll.
- **barbarian**: adds int shield and int sword to barbarian, that increase the value of defense by the shield value or increase the value of attack rolls by the sword value).
- **goblin**: calcDamage() incurs 4 less damage than the corresponding calculation in the parent class would incur.
- **blueMan**: calcDamage() is the exact same algorithm in original code from Project 3 – inherited from base class and not overridden.

6. choosePair() function will also be modified: see below...

Extensions of the Original:

1. The main flow of the program will include the functionality of having the user select players and instantiate objects from subclasses (descendants) of the character class, using pointers to prevent slicing and so that the function choosePair() can still be used as objects are passed in by reference. However, the definition of choosePair() will change to set the attacker and defender based on the user input rather than randomly generated numbers.

2. reptile subclass extended (dragon extends reptile), and **dragon.tailWhip() redefined in reptile**:

- If dragon is attacking, **tailWhip()** adds 2 to the roll. If dragon is defending, tailWhip() nullifies the attacker's roll if attacker rolls even, otherwise does not change anything.
- When attacking, if dragon rolls odd, dragon breathes fire and kills defender.
- If dragon rolls even (whether attacking or defending), twice the armor is applied in calcDamage(). If dragon rolls odd, half the armor is applied in calcDamage().
- A helper function will be added (declared and defined) to the dragon class to determine which of these cases to apply and how that may change the armor and strength points of both the attacker and defender. This function will be called outside of calcDamage() (but still be part of the dragon object) because it will be called on the dragon object not only when it is defending itself but also while attacking.
    - Algorithm:
        - if dragon rolls even:
            - armor applied in damage calculation = armor * 2
        - if dragon rolls odd:
            - if dragon is attacker:
                - defender's strength = 0
            - else (where dragon is defender):
                - armor applied in damage calculation = armor / 2

3. **Main functionality added to the original:** The main containment structure for instantiated objects needs to change, so that objects are instantiated and destroyed dynamically (during runtime).

- Original design: Objects that were instantiated at compile time were added to the character vector.
- New design: A new function will be introduced that instantiates a character object (child or descendant of character) based on user input. initVector() will thus be removed in place of the new function.
    - Function: createPlayer(int choice, std::vector<character> &c), where int choice determines which character to instantiate and add to the vector.

Initial Thoughts on Testing:

Test cases to be mainly implemented in loops that iterate over arrays of function arguments, where each loop iteration (1) calls a function with array element x, an element that is the arguments to that function and (2) displays the return value or information about the post-conditions of the function call. To hopefully cut down on testing time, output will provide a formatted report of expected and actual values so the results do not have to be hand-typed into a chart as I've done previously.