Alex Hoffer

Test Plan

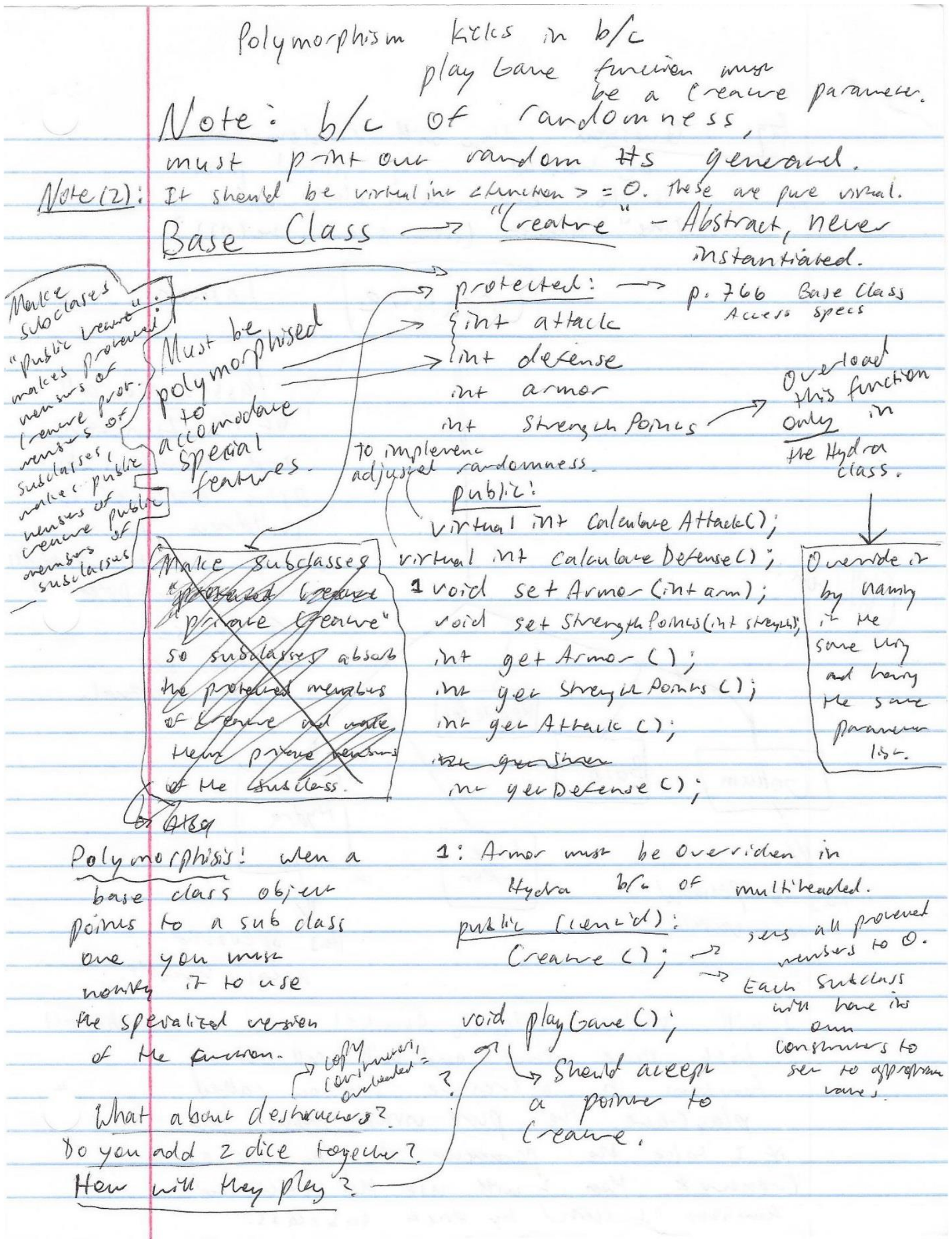
Value	**Expected**	**Actual**	**Comments**
Hydra’s special CAN occur.	Can occur, and occurs correctly.	Yes- after running the simulation many times, the message that the special is enabled is printed and the special takes effect.	
Gollum’s special CAN occur.	Can occur, works correctly.	Yes- after running the simulation many times, the message that the special is enabled is printed and the special takes effect.	
BlueMen wins most of the time.	BlueMen wins most of the time.	Yes- in general, BlueMen’s extra abilities mean that they win most of the time they play.	
ReptilePeople wins most of the time.	ReptilePeople wins most of the time.	Yes- in general, ReptilePeople’s extra abilities mean that they win most of the time they play.	
Gollum loses most of the time.	Gollum loses most of the time.	Yes- in general, Gollum’s underwhelming abilities mean he loses most of the time he plays. The special attack, when activated, makes a small difference.	
Entering 0 for the number of rounds exits the program.	Yes.	Yes- the program recognizes this is not a valid value.	
Entering a negative number for the number of rounds exits the program.	Yes.	Yes- the program recognizes this is not a valid value.	

Entering an extremely high number of rounds guarantees somebody wins eventually.	Since the battle accepts a number of rounds but ends whenever a player dies, entering a large number of rounds means that one of the characters is guaranteed to die.	Yes- in general, entering 20 as the number of rounds will guarantee that one of the characters dies.	
A subclass can battle the same subclass.	An example: Gollum can battle Gollum. Since there is randomization at work, they will not tie.	Yes. Gollum can lose to Gollum, and Gollum can win to Gollum.	
Entering a number besides 0 or 1 when asked if the user wants to play again will be met with an error message.	There is an else statement that contains all possibilities not 0 or 1, prints error if alternative value is entered.	Yes. You can’t enter anything but 0 or 1 in this field without the program exiting.	

Design Documentation

All included documents here were drafted before coding began. There are many differences between these drafts and the final product.

1. Base Class Draft:



Polymorphism kicks in b/c play Game function must be a Creature parameter.

Note: b/c of randomness, must print out random #s generated.

Note (2): It should be virtual int <function> = 0. These are pure virtual.

Base Class → "Creature" – Abstract, never instantiated.

Make subclasses "public Creature": makes protected members of Creature prot. members of subclasses, makes public members of Creature public members of subclasses

protected: → p. 766 Base Class Access specs

int attack

int defense

int armor

int Strength Points → Overload this function only in the Hydra class.

Must be polymorphised to accomodate special features.

to implement adjusted randomness.

public:

virtual int Calculate Attack();

virtual int Calculate Defense();

1 void set Armor (int arm);

void set Strength Points (int strength);

int get Armor ();

int get Strength Points ();

int get Attack ();

~~int get Stren~~

int get Defense ();

Override it by naming it the same way and having the same parameter list.

~~Make Subclasses "protected Creature" "private Creature" so subclasses absorb the protected members of Creature and make them private members of the subclass.~~

→ 4869

Polymorphisis! when a base class object points to a sub class one you must notify it to use the specialized version of the function.

1: Armor must be Overriden in Hydra b/c of multiheaded.

public (cont'd):

Creature (); → sets all protected members to 0. → Each Subclass will have its own constructors to set to appropriate values.

void playGame(); → Should accept a pointer to Creature.

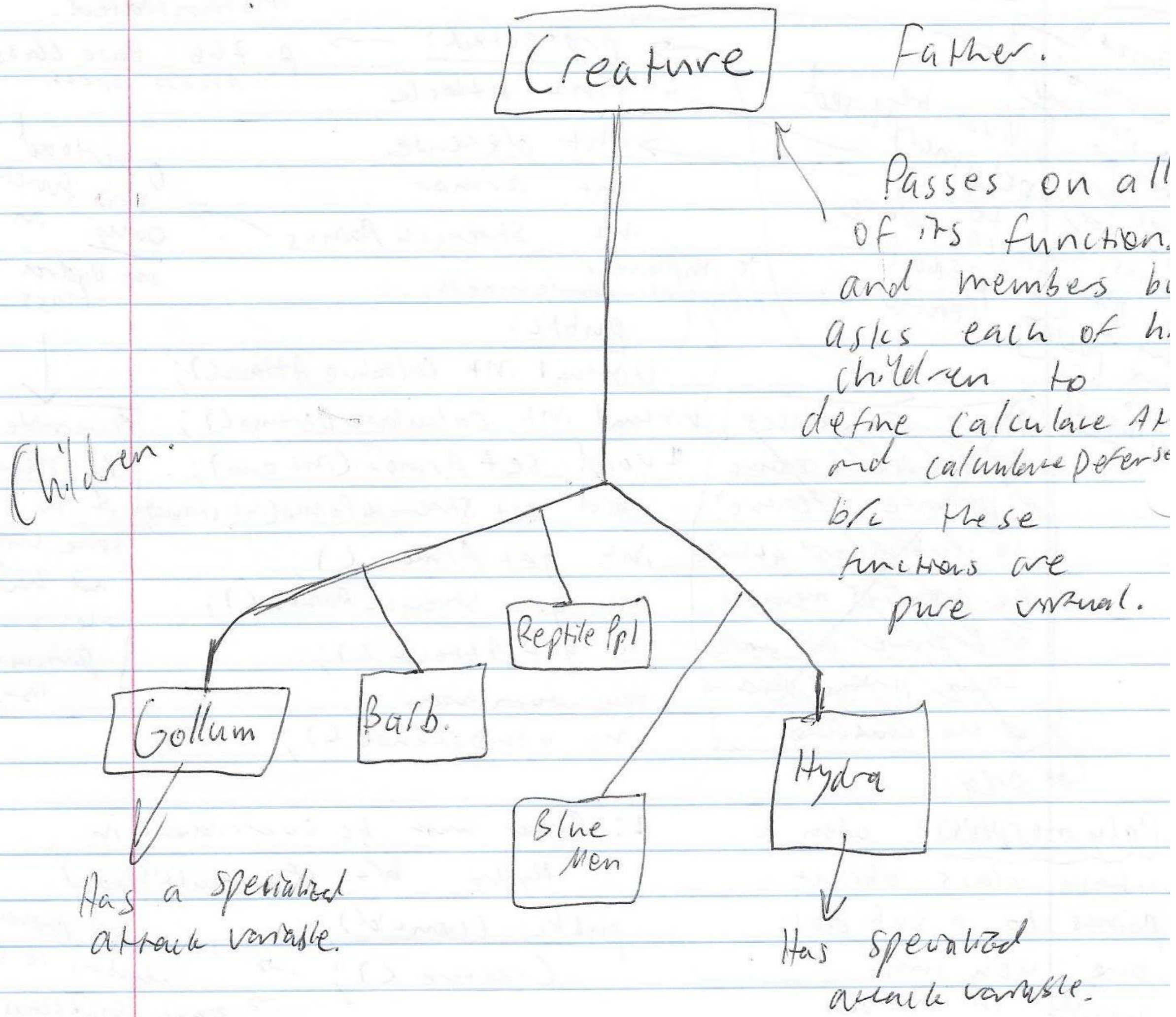
What about destructors? → copy constructors, overloaded = ?

Do you add 2 dice together?

How will they play?

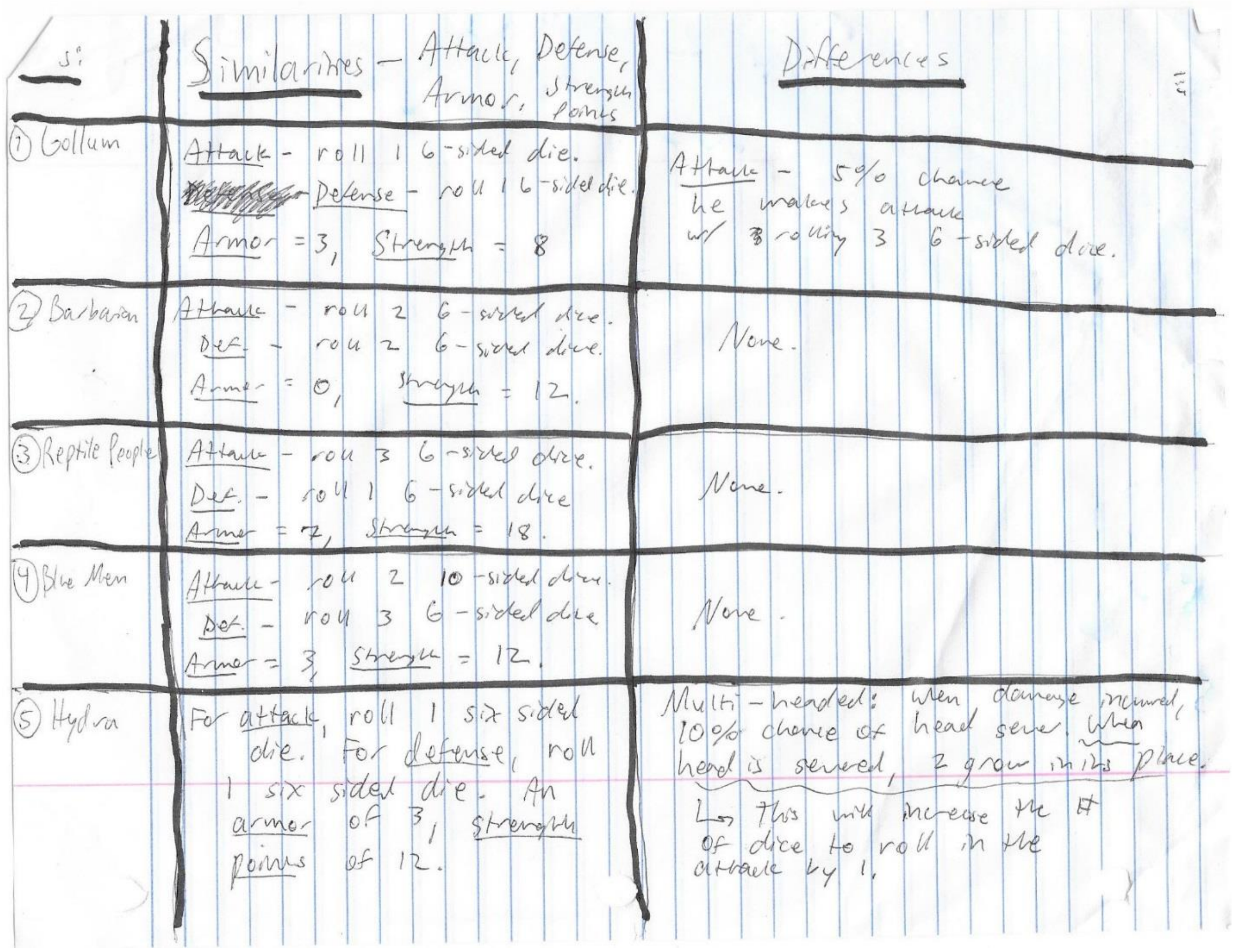
2. Tree of Base Classes, Subclasses:

Pg Question: How will randomness be handled? My initial plan is to seed each class w/ "Time" header (i.e. srand(time(0))).



I will make calculateAttack() and calculateDefense both pure virtual and will call these functions in a Creature function called playGame. The pure virtual means that if I take the parameter of playGame as a Creature* then it will use the specialized functions as defined by each subclass.

3. Crucial differences between subclasses: (i.e. where polymorphism is required)



S:	Similarities – Attack, Defense, Armor, strength points	Differences
① Gollum	Attack - roll 1 6-sided die. Defense - roll 1 6-sided die. Armor = 3, Strength = 8	Attack - 5% chance he makes attack w/ ~~3~~ rolling 3 6-sided dice.
② Barbarian	Attack - roll 2 6-sided dice. Def. - roll 2 6-sided dice. Armor = 0, Strength = 12.	None.
③ Reptile People	Attack - roll 3 6-sided dice. Def. - roll 1 6-sided dice Armor = 7, Strength = 18.	None.
④ Blue Men	Attack - roll 2 10-sided dice. Def. - roll 3 6-sided dice Armor = 3, Strength = 12.	None.
⑤ Hydra	For attack, roll 1 six sided die. For defense, roll 1 six sided die. An armor of 3, strength points of 12.	Multi-headed: when damage incurred, 10% chance of head sever. When head is severed, 2 grow in its place. ↳ This will increase the # of dice to roll in the attack by 1.

4. Subclass Draft (pt 1):

```
Subclass: (1) Gollum : public Creature
        (absorbs protected members, sets them to protected)
        protected:
        bool calculateChance();

        public:
        virtual int calculateAttack();
        virtual int calculateDefense();
            Gollum()
            {
              armor = 3;
              strength = 8;
            }
```

Gollum has 5% chance of rolling 3 6-sided dice. This function will return true if 5% condition is met.

Calls calculateChance() if it returns true, he rolls 3 6-sided dice. If it returns false, he rolls 1 6 sided dice.

~~Rolls 1 6 sided~~ Roll 1 6 sided die, let defense = result.

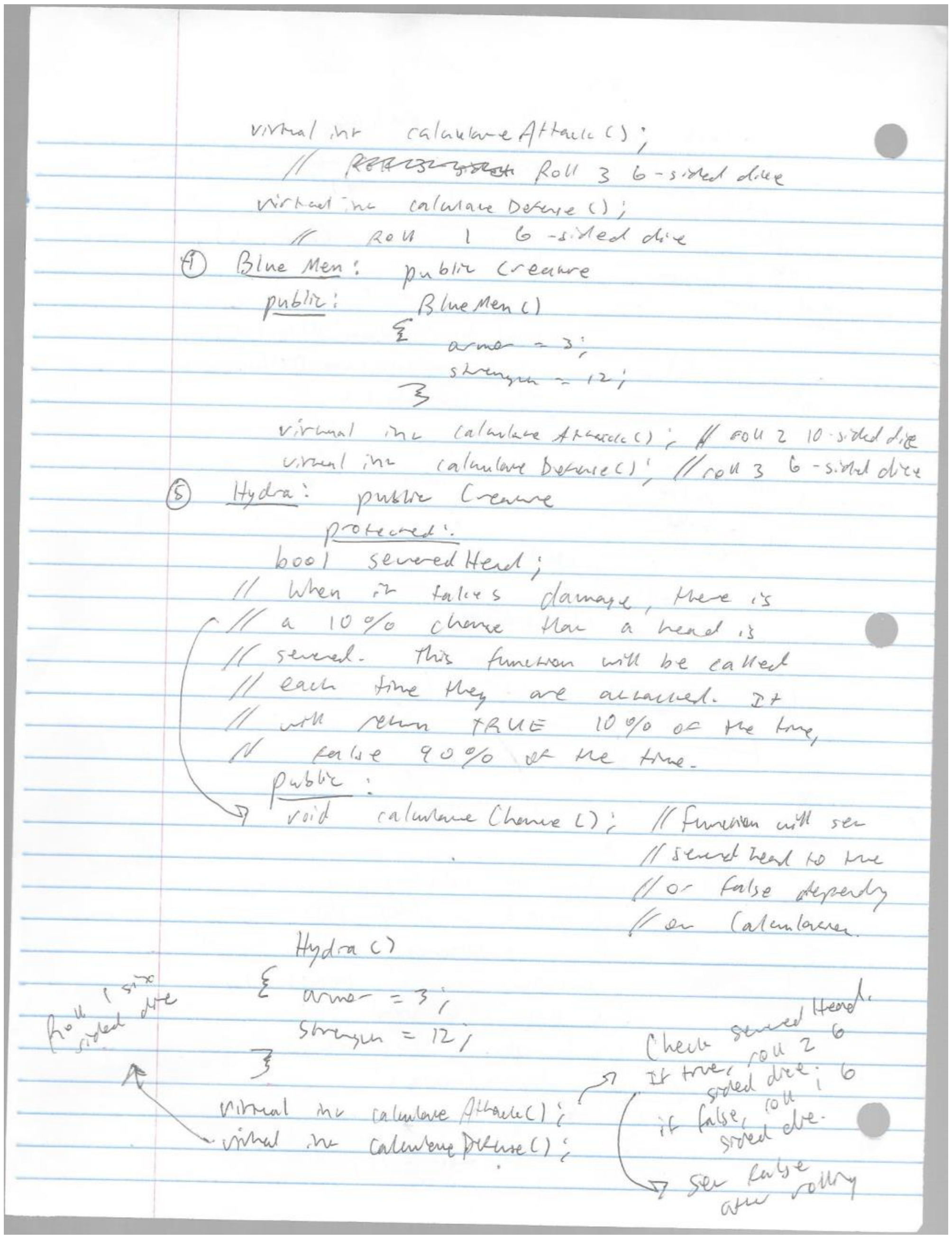
```
(2) Barbarian : public Creature
    public:   Barbarian()
              {
                 strength = 12;
                 // keep armor = 0 from default
              }

          virtual int calculateAttack();
          { // Roll 2 6-sided dice.
            // No contingencies, nothing special.
          virtual int calculateDefense();
          { // Roll 2 6-sided dice.
```

Q: Do calculateAttack/ calculateDefense need to return an int?

```
(3) Reptile People : public Creature
       public:  ReptilePeople()
                {
                   armor = 7;
                   strength = 18;
                }
```

5. Subclass Draft (pt. 2):



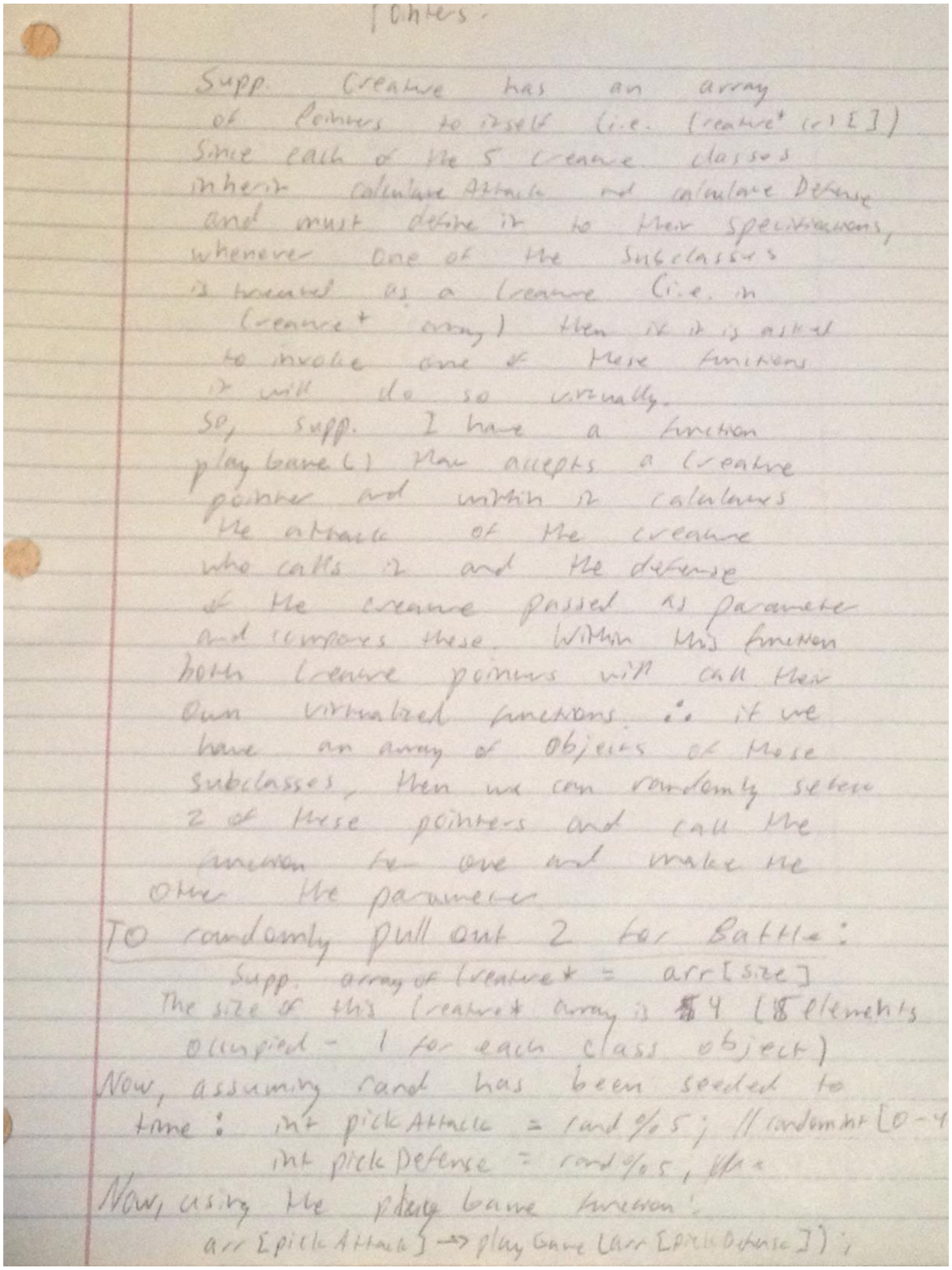
```
virtual int   calculateAttack();
   // [illegible] Roll 3 6-sided dice
virtual int   calculateDefense();
   //    Roll  1  6-sided dice
(4) Blue Men:   public Creature
   public:       BlueMen()
                 {
                     armor = 3;
                     strength = 12;
                 }

   virtual int   calculateAttack(); // roll 2 10-sided dice
   virtual int   calculateDefense(); // roll 3 6-sided dice
(5) Hydra:   public Creature
         protected:
   bool  severedHead;
 // When it takes damage, there is
 // a 10% chance that a head is
 // severed. This function will be called
 // each time they are attacked. It
 // will return TRUE 10% of the time,
 // false 90% of the time.
   public:
   void   calculateChance();  // function will set
                              // severedHead to true
                              // or false depending
                              // on calculation.

      Hydra()
    {  armor = 3;
       strength = 12;
    }
  virtual int  calculateAttack();     Check severedHead.
  virtual int  calculateDefense();    If true, roll 2 6
                                      sided dice; 
                                      if false, roll 1 6
                                      sided dice.
                                      set false
                                      after rolling

roll 1 six
sided dice
```

6. Ideas for Main: Randomly Selecting 2 Subclasses Draft:



Pointers:

Supp. Creature has an array of Pointers to itself (i.e. Creature* cr[ ])
Since each of the 5 creature classes inherit calculate Attack and calculate Defense and must define it to their specifications, whenever one of the subclasses is treated as a Creature (i.e. in Creature* array) then if it is asked to invoke one of these functions it will do so virtually.
So, supp. I have a function playGame( ) that accepts a Creature pointer and within it calculates the attack of the creature who calls it and the defense of the creature passed as parameter and compares these. Within this function both Creature pointers will call their own virtualized functions ∴ if we have an array of objects of these subclasses, then we can randomly select 2 of these pointers and call the function for one and make the other the parameter

TO randomly pull out 2 for Battle:

Supp. array of Creature* = arr[size]
The size of this Creature* array is ~~5~~ 4 (8 elements occupied - 1 for each class object)
Now, assuming rand has been seeded to time: int pickAttack = rand%5; // random int [0-4
int pickDefense = rand%5; // [illegible]
Now, using the playGame function:
arr[pickAttack] → playGame(arr[pickDefense]);

Reflection:

I made many changes to the original draft that it would be quite difficult to describe each and every one. Some were miniscule (i.e. I added a string member called name to Creature for identification purposes). These will take a backseat to the more important ones, as in I will not be able to remember each one and so some will not be mentioned.

That being said, here are the key differences:

1. Creating static members for each subclass that correspond to their number of times killed and number of times they have been killed. This way, each object of a subclass holds their win-loss count and it doesn't reset if the user plays many rounds.

2. Added pure virtual functions to Creature for increasing the number of times killed and increasing the number of times the creature has been killed. This way, each subclass can increase their, and only theirs, static members.

3. Created a static bool member called hasPlayed which is set to true immediately when an object of a subclass is thrown into combat. This way, whenever the win-loss counts are printed only the information of subclasses who have actually fought are included.

4. Pure virtual accessor function added to Creature that returns the bool hasPlayed to be used for the purposes described in 3.

5. Pure virtual accessor functions for returning the number of times killed and the number of times been killed.

6. Added a print record function which uses many of the above functions to print out the specialized win-loss information for each subclass object that has been in combat.

7. The Creature function for playing the game has changed- it used to accept no parameters, now it accepts a pointer to a Creature object.

8. Gollum’s function calculateChance() is now called backJump, it sees if the 5% case has occurred and if it has, returns true, if it hasn’t, returns false.

9. Hydra’s special has been expanded into three components: 1) severedHead() which sees if the 10% case has occured, and if it has sets specialAttack to true, 2) the member variable specialAttack which will be true if and only if the 10% case has occurred, 3) the accessor getSpecialAttack() which returns specialAttack’s truth value.

All in all, I am satisfied with my original drafts because they laid a useful ground work for the final product, and the largest changes I made were the static variables and their corresponding Creature functions and these were done mostly just to make the program more visually appealing and easy to follow, not necessarily a requirement.