CS162-400

Assignment 3- Polymorphism

Step 1: Identifying Classes and Objects — Finding the nouns.

Describe the problem: There are dark times in the world of Polytron, where battles are fought every day between Goblins, Barbarians, Reptile People, Blue Men, and a create known as The Shadow. (added drama: We must see through the outcomes of these attack so we as humans can prepare are own defense.) Each of these creatures has their own individual attacks, defense methods, level of Armor, and Strength Points. Each creature will be able to battle both one from its own race, or another from the four different races.

FAQ:

Are there any special powers?: Yes, as a matter of fact their are. *The Shadow* has a special "Armor" trait, in that it can deceive its enemies and dodge their attacks 50% of the time. *The Goblin* also has a special in the form of its attack, in that in extreme cases (rolling a 12), it may cut the Achilles of opponent (if not Goblin opponent). This attack will cause the opponent to be weakened, and its attacks will only be worth half.

Who will attack first?: This will be randomly chosen.

Can there be a draw?: No battlefield on Polytron goes unclaimed. There will be no draws, as the first creature to run out of Strength Points will be automatically eliminated.

Who will fight: There are no true alliances found in this world, and therefor any two creatures can fight, and a creature may also fight itself.

I will approach this program by having a base class, creature, and 5 subclasses the represent the individuals creatures. Base Class: Creature Derived Class(s): blue_men, reptile, barb, gob, and shadow. I will create another class called Battlefield that will have these classes composed within. The Base class will be an abstract class as it will have pure virtual functions.

The class Battlefield will have a pointer to an array (of two) of the base class (creature) that will dynamically assign user chosen creatures to battle; this will automatically type-cast the individual create class (ex. gob) into the base class type, and keep track of the actual class using dynamic binding at run time.

— That was a mouth full—

I have drawn out what I believe to be a program that will allow all of what needs to be accomplished. Let me discuss my approach to major design implementation:

Member Variables in Creature Class:

Int armor, int strength, int defense, int attack, string name, static int damage, static int goblin.

Member Functions:

void attack() { if(name==goblin){static damage=a) else{ static damage=a/static int goblin) }

int damage ()=0;

void defense() { static damage= defend() ..if(name!=shadow)… else{damage -= getArmor();… Strength-=damage}

int defend()=0;

int getStrength () const {return this->strength)

Creature usable functions, and pure virtual functions:

The base class will be able to call functions attack and defense. Within the functions, it will call setDamage(damage ();); and defend(); correspondingly, which will be pure virtual functions. These functions will return int values that are determined by the individual sub-creature member.

Set Damage will take an integer that is determined by running the attack() function the accurate sub-creature member function.

Accounting for Goblin Special Achilles:

I use a function call to setDamage in the base class function attack, (with the input of damage) to account for the goblin special. If the name of the attacker is Goblin, then the attack will always be worth the full value (achilles never works against goblins). If the attacker is not a goblin, I divide their attack by the static goblin value (which will either be 1 (not activated) or 2 (activated) and will divide their attack in half).

Accounting for cross object Attack and Defend:

Moving from attacking with one object, and then defending with another requires a way to communicate between the two values as you bounce from one sub-creature object to the next. I choose to address this by using a static member variable, in the base class, that is updated with the damage of the attack, the defense of the attack, the armor of the sub-creature, and is then that value is subtracted from the strength points (if greater than 0).

What about The Shadow Special (Teleport):

Using the name of the defender, I call a specific function if the defenders name is shadow (int getArmor ( );) The armor for shadow will either be 0 or will block full attack. The Max value an attack could be is by either the Blue Men or The Shadow would be 20. The minimum value of the defense for the shadow is one. Therefore, we must account for the maximum damage which is 19. The special, when implemented, will cause an armor reduction of 19 (note we could have set it to 1000) because the function defense() will buffer out damage-armor if it is less than zero.

How do you know when someone dies?:

Using the get strength function of the base class, I will call after each set of attack and defend and if strength points have gone below zero, then that sub-creature is dead!

Which creatures will battle?:

I have decided to make this a menu option, and allow the user to place two sub-creatures into the pointer array of the base class. This will be done using a for loop, and matching the section of 1-5 (for each sub-creature), with an if statement and setting the first element in array and then the second.

How is this program random?:

I will use the library <ctime> , the call srand(time(0)) , and the rand()% function to address random numbers. The dice rolls are random (attack and defense): I will stick with the probabilities of the dice rolls and the number of dice. Who goes first: A 50/50 chance of who will go first. srand()%1 (0 or 1).

Step 2: Define Each Class Attribute

We will have seven classes. The parent class called creature, the five subclasses: blue_men, reptile, barb, gob, and shadow. There is also a class battleField, that has composition of all previously described classes.

Class name : **creature**

Attributes :

<string> **name** // blueMen, barb, gob, reptile, shadow

<int> **armor** // set upon initialization

<int> **strength** // set upon initialization

<int> **attack** // holds temp attack values

<int> **defense** // holds temp defense values

static <int> **damage** // used to store attack-defense-armor

static <int> **goblin** // if goblin has special, will divide attack of opponent by 2

Class name : **blue_men, reptile, barb, gob, and shadow**

No member variables, rely on the derived member variables (specialized functions though).

Class name : **battleField**

Attributes :

<creature> *type[2]; // A pointer array of the creature data type, holds two values.

<shadow> shady;

<bool> death;

<int> choice1,choice2; //stores creature selection.

Step 3: Class behavior and function.

The battlefield class will be the main driver of this entire program, as it composes the other classes. This relationship makes sense in OOP, because I battlefield would potentially have various creatures within it. The base class, creature, will have a constructor that iniaiate its member variables. Note that this class cannot have an instance created by itself, but it can be pointed to. The sub-create classes will pass in pre-constructed variables for (name, armor, and strength points) as these are no going to change (despite the shadow armor exception). The battlefield will use a user defined constructor, but only to call the function mainMenu( ). Here are what the default constructs will look like for the other classes.

+creature(string name, int armor, int strength); ,
+reptile :public creature("Reptilian", 7,18); ,
+barb() :public creature("Barbarian", 0,12); ,
+gob():public creature("Goblin", 3,8); ,
+shadow():public creature("The Shadow", 0,12);
+blue_men():public creature("Blue Man", 3,12)
+battleField(){ mainMenu (); }

---

creature behavior: Described above on page 2

- void attack()
- int damage ()=0;
- void defense()
- int defend()=0;
- int getStrength ()
- int rollSix ();
- int rollTen();

---

reptile behavior:

- Rolls three 6 sided die in order to generate attack value.

+ damage ( ): int;

- Generates a defense amount by rolling one six sided dice.

+ defense ( ): int;

barb behavior:

- Rolls two 6 sided die in order to generate attack value.

+ damage ( ): int;

- Generates a defense amount by rolling two six sided dice.

+ defense ( ): int;

---

gob behavior:

- Rolls two 6 sided die in order to generate attack value. If a 12 is rolled, set goblin special.

+ damage ( ): int;

- Generates a defense amount by rolling one six sided dice.

+ defense ( ): int;

---

shadow behavior:

- Rolls two 10 sided die in order to generate attack value.

+ damage ( ): int;

- Generates a defense amount by rolling one six sided dice.

+ defense ( ): int;

---

blue_men behavior:

- Rolls two 10 sided die in order to generate attack value.

+ damage ( ): int;

- Generates a defense amount by rolling one six sided dice.

+ defense ( ): int;

battleField behavior:

- Acts a a main menu to allow for creature choice.

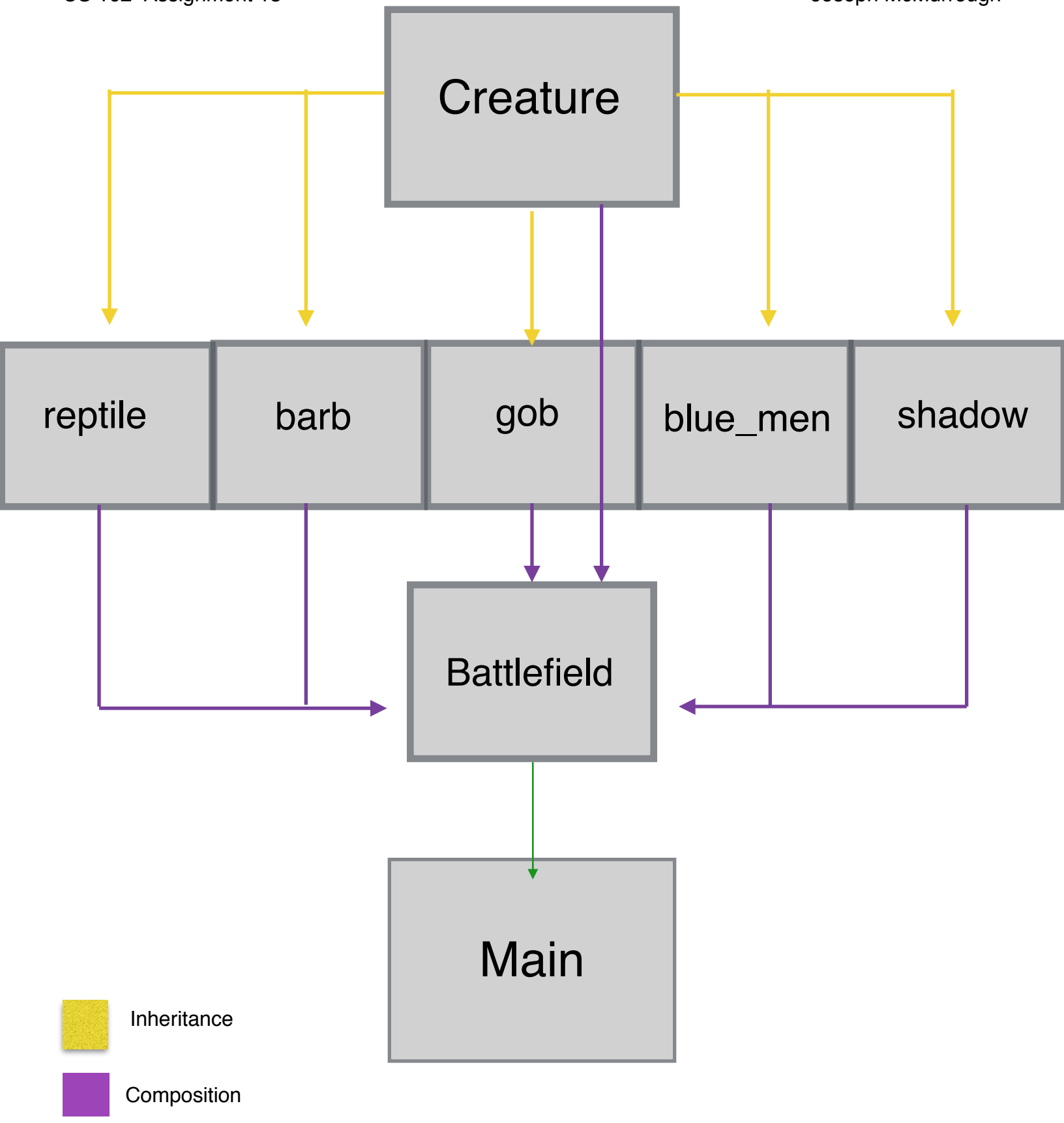
+ mainMenu ( ): void;

- Randomly selects the first to attack.

+ random ( ): int;

- Check for death

+ checkDeath ( ): bool;


Creature
reptile
barb
gob
blue_men
shadow
Battlefield
Main
Inheritance
Composition

Step 4: Main ( )

Our main function will simply create an object for a battleField , and use its composed class's functions along with its own personal functions.

Final Considerations — Modifications — Notes:

- **CHANGE:** I was to have a third static function called static int shadowArmor that was virtual and called upon when the name was equal to shadow. I failed to realized that if this was to be virtual, each individual subclass would have to over-ride the function, when it is specific to The Shadow, or else it makes the subclass abstract. I will fix this by adjusting "The Shadows" call to the virtual function defend(); instead of only outputting a defense amount, it will output defense plus armor (special). There will be a bypass when subtracting armor from a normal in the base class function.
- **CHANGE**: Got ride of shadowArmor function.
- **CHANGE:** Static members from private to protected so they can be accessed by derived class directly.
- **CHANGE:** Added in member variable in battleField rr, for random input holder.
- **CHANGE:** Added getName function for display purposes.
- **CHANGE:** Added sayItatk and sayItdef to add a dialog to the program.
- **CHANGE:** Added loser variable in battleField to keep track of who lost for future use.
- **CHANGE**: Included wait fiction (part of unix library), to create more true random numbers.
- **CHANGE**: Noted error in testing was that the static goblin special variable was not being reset after a game, and attacks were being divided by 2 in a new match if continued without program restarting. Made goblin static variable set to 1 in the constructor of the Goblin class.

Reflection: I really enjoyed this project. I did make a few changes along the way, but noting major really. I really only added details. I enjoyed the testing phase as well, although it was kind of tedious. Though I found myself rooting for the underdog, and hoping for surprising victories!