Project 2

● Graded

Student

Devin Chen

Total Points

90 / 100 pts

Autograder Score

70.0 / 80.0

Failed Tests

Checks that test.cpp tests Creature functions (0/10)

Passed Tests

Test compiles (5/5)
Tests Dragon class default constructor (2.5/2.5)
Tests Ghoul class default constructor (2.5/2.5)
Tests Mindflayer class default constructor (2.5/2.5)
Tests Dragon class parameterized constructor for default values (2.5/2.5)
Tests Ghoul class parameterized constructor for default values (2.5/2.5)
Tests Mindflayer class parameterized constructor for default values (2.5/2.5)
Tests Dragon class parameterized constructor (5/5)
Tests Ghoul class parameterized constructor (5/5)
Tests Mindflayer class parameterized constructor (5/5)
Tests Dragon class mutator and accessor functions (10/10)
Tests Ghoul class mutator and accessor functions (10/10)
Tests Mindflayer class mutator and accessor functions (10/10)
Test for read-only functions and parameters (5/5)

Question 2

Style & Documentation

20 / 20 pts

✓ **+ 5 pts** Style

✓ **+ 5 pts** Indicates name and and description in comment preamble at top of file

✓ **+ 10 pts** Has function preambles with @pre, @post, @param, @return where appropriate

+ 20 pts No-Compile Adjustment

+ 7 pts No-Compile Adjustment for partial (1/3) implementation

+ 14 pts No-Compile Adjustment for partial (2/3) implementation

+ 0 pts Insufficient submission

Autograder Results

Test compiles (5/5)

Your program compiles!

Tests Dragon class default constructor (2.5/2.5)

Your program passed.

Tests Ghoul class default constructor (2.5/2.5)

Your program passed.

Tests Mindflayer class default constructor (2.5/2.5)

Your program passed.

Tests Dragon class parameterized constructor for default values (2.5/2.5)

Your program passed.

Tests Ghoul class parameterized constructor for default values (2.5/2.5)

Your program passed.

Tests Mindflayer class parameterized constructor for default values (2.5/2.5)

Your program passed.

Tests Dragon class parameterized constructor (5/5)

Your program passed.

Tests Ghoul class parameterized constructor (5/5)

Your program passed.

Tests Mindflayer class parameterized constructor (5/5)

Your program passed.

Tests Dragon class mutator and accessor functions (10/10)

Your program passed.

Tests Ghoul class mutator and accessor functions (10/10)

Your program passed.

Tests Mindflayer class mutator and accessor functions (10/10)

Your program passed.

Checks that test.cpp tests Creature functions (0/10)

```
test.cpp compiles!

test.cpp is not testing every function of the derived classes.

Test Failed: 'Incorrect Implementation' != ''
- Incorrect Implementation
+
```

Test for read-only functions and parameters (5/5)

Your program has functions and parameters as read-only (const) where appropriate.

Submitted Files

▾ .gitignore

Download

```
.DS_Store
.vscode
*.log
*.o
main
```

Creature.cpp Download

```cpp
/*
CSCI235 Spring 2024
Project 1 - Creature Class
Georgina Woo
Nov 13 2023
Creature.cpp defines the constructors and private and public function implementation of the Creature class
*/

#include "Creature.hpp"

/**
    Default constructor.
    Default-initializes all private members.
    Default creature name: "NAMELESS".
    Booleans are default-initialized to False.
    Default enum value: UNKNOWN
    Default Hitpoints and Level: 1.
*/
Creature::Creature(): name_{"NAMELESS"},  category_{UNKNOWN}, hitpoints_{1}, level_{1}, tame_{false}
{

}

/**
    Parameterized constructor.
    @param      : A reference to the name of the creature (a string). Set the creature's name to NAMELESS if the provided string contains non-alphabetic characters.
    @param      : The category of the creature (a Category enum) with default value UNKNOWN
    @param      : The creature's hitpoints (an integer) , with default value 1 if not provided, or if the value provided is 0 or negative
    @param      : The creature's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
    @param      : A flag indicating whether the creature is tame, with default value False
    @post       : The private members are set to the values of the corresponding parameters.
    Hint: Notice the default arguments in the parameterized constructor.
*/
Creature::Creature(const std::string& name, Category category, int hitpoints, int level, bool tame): category_{category}
{
    if(!setName(name))
    {
        name_ = "NAMELESS";
    }

    if(!setHitpoints(hitpoints))
    {
        hitpoints_ = 1;
    }
    if(!setLevel(level))
    {
        level_ = 1;
    }
    tame_ = tame;

}

/**
    @param : the name of the Creature, a reference to string
    @post  : sets the Creature's name to the value of the parameter in UPPERCASE.
             (convert any lowercase character to uppercase)
             Only alphabetical characters are allowed.
           : If the input contains non-alphabetic characters, do nothing.
    @return : true if the name was set, false otherwise
*/
bool Creature::setName(const std::string& name)
{
    if (name.length() == 0)
    {
        return false;
    }
    else
    {
        std::string nameUpper = name;
        for (int i = 0; i < name.length(); i++)
        {
            if (!isalpha(name[i]))
            {
                return false;
            }
            else
            {
                nameUpper[i] = toupper(name[i]);
            }
        }
        name_ = nameUpper;
        return true;
    }
}

/**
     @return : the name of the Creature
*/
std::string Creature::getName() const
{
    return name_;
}


/**
```

```
    @param  : the category of the Creature (an enum)
    @post   : sets the Creature's category to the value of the parameter
*/
void Creature::setCategory(const Category& category)
{
    category_ = category;
}


/**
     @return : the category of the Creature (in string form)
*/
std::string Creature::getCategory() const
{
    switch(category_)
    {
        case UNDEAD:
            return "UNDEAD";
        case MYSTICAL:
            return "MYSTICAL";
        case ALIEN:
            return "ALIEN";
        default:
            return "UNKNOWN";
    }
}

/**
    @param  : an integer that represents the creature's hitpoints
    @pre    : hitpoints > 0 : Creatures cannot have 0 or negative hitpoints (do nothing for invalid
input)
    @post   : sets the hitpoints private member to the value of the parameter
    @return : true if the hitpoints were set, false otherwise
*/
bool Creature::setHitpoints(const int& hitpoints)
{
    if (hitpoints > 0)
    {
        hitpoints_ = hitpoints;
        return true;
    }
    else
    {
        return false;
    }
}


/**
     @return : the value stored in hitpoints_
*/
int Creature::getHitpoints() const
```

```
{
    return hitpoints_;
}

/**
    @param  : an integer level
    @pre    : level >  0 : Characters cannot have 0 or negative levels (do nothing for invalid input)
    @post   : sets the level private member to the value of the parameter
    @return : true if the level was set, false otherwise
*/
bool Creature::setLevel(const int& level)
{
    if (level > 0)
    {
        level_ = level;
        return true;
    }
    else
    {
        return false;
    }
}


/**
     @return : the value stored in level_
*/
int Creature::getLevel() const
{
    return level_;
}


/**
    @param  : a boolean value
    @post   : sets the tame flag to the value of the parameter
*/
void Creature::setTame(const bool& tame)
{
    tame_ = tame;
}


/**
    @return true if the creature is tame, false otherwise
    Note: this is an accessor function and must follow the same convention as all accessor functions
even if it is not called getTame
*/
bool Creature::isTame() const
{
    return tame_;
}
```

```cpp

/**
   @post     : displays Creature data in the form:
   "[NAME]\n
   Category: [CATEGORY]\n
   Level: [LEVEL]\n
   Hitpoints: [Hitpoints]\n
   Tame: [TRUE/FALSE]"
*/
void Creature::display() const
{
   std::cout << name_ << std::endl;
   std::cout << "Category: " << getCategory() << std::endl;
   std::cout << "Level: " << level_ << std::endl;
   std::cout << "Hitpoints: " << hitpoints_ << std::endl;
   std::cout << "Tame: " << (tame_ ? "TRUE" : "FALSE") << std::endl;
}
```

Creature.hpp Download

```cpp
/*
CSCI235 Spring 2024
Project 1 - Creature Class
Georgina Woo
Nov 13 2023
Creature.hpp declares the Creature class along with its private and public members
*/
#ifndef CREATURE_HPP_
#define CREATURE_HPP_
#include <iostream>
#include <string>
#include <cctype>


class Creature
{
    public:
        enum Category {UNKNOWN, UNDEAD, MYSTICAL, ALIEN};
        /**
            Default constructor.
            Default-initializes all private members.
            Default creature name: "NAMELESS".
            Booleans are default-initialized to False.
            Default enum value: UNKNOWN
            Default Hitpoints and Level: 1.
        */
        Creature();

        /**
            Parameterized constructor.
            @param      : The name of the creature (a string)
            @param      : The category of the creature (a Category enum) with default value UNKNOWN
            @param      : The creature's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
            @param      : The creature's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
            @param      : A flag indicating whether the creature is tame, with default value False
            @post       : The private members are set to the values of the corresponding parameters.
            Hint: Notice the default arguments in the parameterized constructor.
        */
        Creature(const std::string& name, Category category = UNKNOWN, int hitpoints = 1, int level = 1, bool tame = false);

        /**
             @param : the name of the Creature, a string
             @post  : sets the Creature's name to the value of the parameter in UPPERCASE (convert any lowercase character to upppercase
                      Only alphabetical characters are allowed.
                    : If the input contains non-alphabetic characters, do nothing.
            @return : true if the name was set, false otherwise
        */
        bool setName(const std::string& name);

        /**
             @return : the name of the Creature
        */
        std::string getName() const;


        /**
            @param  : the category of the Creature (an enum)
            @post   : sets the Creature's category to the value of the parameter
        */
        void setCategory(const Category& category);


        /**
             @return : the race of the Creature (in string form)
        */
        std::string getCategory() const;

        /**
            @param  : an integer that represents the creature's hitpoints
            @pre    : hitpoints > 0 : Creatures cannot have 0 or negative hitpoints (do nothing for invalid input)
            @post   : sets the hitpoints private member to the value of the parameter
            @return : true if the hitpoints were set, false otherwise
        */
        bool setHitpoints(const int& hitpoints);


        /**
             @return : the value stored in hitpoints_
        */
        int getHitpoints() const;

        /**
            @param  : an integer level
            @pre    : level > 0 : Creatures cannot have 0 or negative levels (do nothing for invalid input)
            @post   : sets the level private member to the value of the parameter
            @return : true if the level was set, false otherwise
        */
        bool setLevel(const int& level);


        /**
             @return : the value stored in level_
        */
        int getLevel() const;


        /**
            @param  : a boolean value
            @post   : sets the tame flag to the value of the parameter
        */
        void setTame(const bool& tame);


        /**
            @return true if the Creature is tame, false otherwise
            Note: this is an accessor function and must follow the same convention as all accessor functions even if it is not called getTame
        */
        bool isTame() const;

         /**
            @post     : displays Creature data in the form:
            "[NAME]\n
            Category: [CATEGORY]\n
            Level: [LEVEL]\n
            Hitpoints: [Hitpoints]\n
            Tame: [TRUE/FALSE]"
        */
        void display() const;

    private:
        // The name of the creature (a string in UPPERCASE)
        std::string name_;
        // The category of the creature (an enum)
        Category category_;
        // The creature's hitpoints (a non-zero, non-negative integer)
        int hitpoints_;
        // The creature's level (a non-zero, non-negative integer)
        int level_;
        // A flag indicating whether the creature is tame
        bool tame_;

};

#endif
```

Dragon.cpp Download

```cpp
/**
 * @file Dragon.cpp
 * @author Devin Chen
 * @brief Dragon Class
 * @date 2/19/2024
*/
#include "Dragon.hpp"

/**
 * Default constructor.
 * Default-initializes all private members.
 * Default Category: MYSTICAL
 * Default element: NONE
 * Default number of head(s): 1
 * Booleans are default-initialized to False.
*/
Dragon::Dragon(): Creature(), affinity_{NONE}, num_heads_{1}, can_fly_{false} {
    setCategory(MYSTICAL);
};

/**
  Parameterized constructor.
  @param      : The name of the Dragon (a reference to string)
  @param      : The category of the Dragon (a Category enum) with default value MYSTICAL
  @param      : The Dragon's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Dragon's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Dragon is tame, with default value False
  @param      : The element (an Element enum), with default value NONE if not provided
  @param      : The number of heads (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Dragon can fly, with default value False
  @post       : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
Dragon::Dragon(const std::string &new_name, Category new_category , int new_hitpoint, int new_level, bool new_tame , Element new_element , int new_head, bool new_fly)
    : Creature(new_name, new_category, new_hitpoint, new_level, new_tame), affinity_{new_element}, can_fly_{new_fly} {
if(!setNumberOfHeads(new_head)){
    setNumberOfHeads(1);
};
};

/**
  Getter for the element.
  @return     : The element (a string representation of the Element enum)
*/
```

```
std::string Dragon::getElement()const{
    switch (affinity_){
        case NONE:
            return "NONE";
        case FIRE:
            return "FIRE";
        case WATER:
            return "WATER";
        case EARTH:
            return "EARTH";
        case AIR:
            return "AIR";
    }
};

/**
  Setter for the element.
  @param      : A reference to the element (an Element enum)
  @post       : The element is set to the value of the parameter.
*/
void Dragon::setElement(const Element &new_element){
if(new_element >= NONE && new_element <= AIR){
    affinity_ = new_element;
    };
};


/**
  Getter for the number of heads.
  @return     : The number of heads (an integer)
*/
int Dragon::getNumberOfHeads()const {
    return num_heads_;
};

/**
  Setter for the number of heads.
  @param      : A reference to the number of heads (an integer)
  @pre        : The number of heads is > 0. Do nothing for invalid values.
  @post       : The number of heads is set to the value of the parameter.
  @return     : True if the number of heads is set, false otherwise.
*/
bool Dragon::setNumberOfHeads(const int &new_head){
if(new_head > 0){
    num_heads_ = new_head;
    return true;
}
return false;
};

/**
  Getter for the flight flag.
  @return     : The flight flag (a boolean)
*/
bool Dragon::getFlight()const{
    return can_fly_;
};

/**
  Setter for the flight flag.
  @param      : A reference to the flight flag (a boolean)
  @post       : The flight flag is set to the value of the parameter.
*/
void Dragon::setFlight(const bool &new_fly){
can_fly_ = new_fly;
}
```

Dragon.hpp Download

```
/**
 * @file Dragon.hpp
 * @author Devin Chen
 * @brief Dragon Class
 * @date 2/19/2024
*/
#include "Creature.hpp"
class Dragon : public Creature{

public:
enum Element {NONE, FIRE, WATER, EARTH, AIR};
/**
  Default constructor.
  Default-initializes all private members.
  Default Category: MYSTICAL
  Default element: NONE
  Default number of head(s): 1
  Booleans are default-initialized to False.
*/
Dragon();
/**
  Parameterized constructor.
  @param      : The name of the Dragon (a reference to string)
  @param      : The category of the Dragon (a Category enum) with default value MYSTICAL
  @param      : The Dragon's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Dragon's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Dragon is tame, with default value False
  @param      : The element (an Element enum), with default value NONE if not provided
  @param      : The number of heads (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Dragon can fly, with default value False
  @post       : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
Dragon(const std::string &new_name, Category new_category = MYSTICAL, int new_hitpoint = 1, int new_level = 1, bool new_tame = false, Element new_element = NONE, int new_head = 1, bool new_fly = false);

/**
  Getter for the element.
  @return     : The element (a string representation of the Element enum)
*/
std::string getElement()const;

/**
  Setter for the element.
  @param      : A reference to the element (an Element enum)
  @post       : The element is set to the value of the parameter.
*/
void setElement(const Element &new_element);

/**
  Getter for the number of heads.
  @return     : The number of heads (an integer)
*/
int getNumberOfHeads()const;

/**
  Setter for the number of heads.
  @param      : A reference to the number of heads (an integer)
  @pre        : The number of heads is > 0. Do nothing for invalid values.
  @post       : The number of heads is set to the value of the parameter.
  @return     : True if the number of heads is set, false otherwise.
*/
bool setNumberOfHeads(const int &new_head);

/**
  Getter for the flight flag.
  @return     : The flight flag (a boolean)
*/
bool getFlight()const;

/**
  Setter for the flight flag.
  @param      : A reference to the flight flag (a boolean)
  @post       : The flight flag is set to the value of the parameter.
*/
void setFlight(const bool &new_fly);

private:
Element affinity_;
int num_heads_;
bool can_fly_;
};
```

▾ **Ghoul.cpp** Download

```
/**
 * @file Ghoul.cpp
 * @author Devin Chen
 * @brief Ghoul Class
 * @date 2/19/2024
*/
#include "Ghoul.hpp"
/**
  Default constructor.
  Default-initializes all private members.
  Default Category: UNDEAD
  Default decay: 0
  Default faction: NONE
  Booleans are default-initialized to False.
*/
Ghoul::Ghoul():Creature(),level_of_decay_{0}, faction_{NONE},can_transform_{false}{
    setCategory(UNDEAD);
};

/**
  Parameterized constructor.
  @param      : The name of the Ghoul (a reference to string)
  @param      : The category of the Ghoul (a Category enum) with default value UNDEAD
  @param      : The Ghoul's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Ghoul's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Ghoul is tame, with default value False
  @param      : The level of decay (an integer), with default value 0 if not provided, or if the value provided is negative
  @param      : The faction (a Faction enum), with default value NONE if not provided
  @param      : A flag indicating whether the Ghoul can transform, with default value False
  @post       : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
Ghoul::Ghoul(const std::string &new_name, Category new_type, int new_hitpoint,int new_level, bool new_tame, int new_decay, Faction new_faction, bool new_transform)
:Creature(new_name,new_type, new_hitpoint, new_level, new_tame), faction_{new_faction}, can_transform_{new_transform}{
if (!setDecay(new_decay)){
    level_of_decay_ = 0;
    };
};

/**
  Getter for the level of decay.
  @return     : The level of decay (an integer)
*/

```

```cpp
int Ghoul::getDecay() const{
return level_of_decay_;
}

/**
  Setter for the level of decay.
  @param      : A reference to the level of decay (an integer)
  @pre        : The level of decay must be >= 0 (do nothing otherwise)
  @post       : The level of decay is set to the value of the parameter.
  @return     : true if the level of decay was set, false otherwise
*/
bool Ghoul::setDecay(const int &new_decay){
    if(new_decay >= 0){
        level_of_decay_ = new_decay;
        return true;}
    return false;
};

/**
  Getter for the faction.
  @return     : The faction (a string representation of the Faction enum)
*/
std::string Ghoul::getFaction()const{
switch (faction_){
        case NONE:
            return "NONE";
        case FLESHGORGER:
            return "FLESHGORGER";
        case SHADOWSTALKER:
            return "SHADOWSTALKER";
        case PLAGUEWEAVER:
            return "PLAGUEWEAVER";
    }
};

/**
  Setter for the faction.
  @param      : A reference to the faction (a Faction enum)
  @post       : The faction is set to the value of the parameter.
*/
void Ghoul::setFaction(const Faction& faction) {
    faction_ = faction;
}

/**
  Getter for the transformation.
  @return     : The transformation (a boolean)
*/
bool Ghoul::getTransformation()const{
    return can_transform_;
};

```

```
/**
  Setter for the transformation.
  @param      : A reference to the transformation (a boolean)
  @post       : The transformation is set to the value of the parameter.
*/
void Ghoul::setTransformation(const bool &new_transformation){
    can_transform_ = new_transformation;

};
```

Ghoul.hpp Download

```
/**
 * @file Ghoul.hpp
 * @author Devin Chen
 * @brief Ghoul Class
 * @date 2/19/2024
*/
#include "Creature.hpp"

class Ghoul:public Creature{

public:
enum Faction {NONE, FLESHGORGER, SHADOWSTALKER, PLAGUEWEAVER};
/**
  Default constructor.
  Default-initializes all private members.
  Default Category: UNDEAD
  Default decay: 0
  Default faction: NONE
  Booleans are default-initialized to False.
*/
Ghoul();

/**
  Parameterized constructor.
  @param      : The name of the Ghoul (a reference to string)
  @param      : The category of the Ghoul (a Category enum) with default value UNDEAD
  @param      : The Ghoul's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Ghoul's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Ghoul is tame, with default value False
  @param      : The level of decay (an integer), with default value 0 if not provided, or if the value provided is negative
  @param      : The faction (a Faction enum), with default value NONE if not provided
  @param      : A flag indicating whether the Ghoul can transform, with default value False
  @post      : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
Ghoul(const std::string &new_name, Category new_type = UNDEAD, int new_hitpoint = 1,int new_level = 1, bool new_tame = false, int new_decay = 0, Faction new_faction = NONE, bool new_transform = false);

/**
  Getter for the level of decay.
  @return     : The level of decay (an integer)
*/
int getDecay() const;

/**
```

```cpp
  Setter for the level of decay.
  @param      : A reference to the level of decay (an integer)
  @pre        : The level of decay must be >= 0 (do nothing otherwise)
  @post       : The level of decay is set to the value of the parameter.
  @return     : true if the level of decay was set, false otherwise
*/
bool setDecay(const int &new_decay);

/**
  Getter for the faction.
  @return     : The faction (a string representation of the Faction enum)
*/
std::string getFaction()const;

/**
  Setter for the faction.
  @param      : A reference to the faction (a Faction enum)
  @post       : The faction is set to the value of the parameter.
*/
void setFaction(const Faction &faction);

/**
  Getter for the transformation.
  @return     : The transformation (a boolean)
*/
bool getTransformation()const;

/**
  Setter for the transformation.
  @param      : A reference to the transformation (a boolean)
  @post       : The transformation is set to the value of the parameter.
*/
void setTransformation(const bool &new_transformation);


private:
int level_of_decay_;
Faction faction_;
bool can_transform_;
};
```

Makefile Download

```
CXX = g++
CXXFLAGS = -std=c++17 -g -Wall -O2

PROG ?= main
OBJS = Creature.o test.o Dragon.o Ghoul.o Mindflayer.o

all: $(PROG)

.cpp.o:
	$(CXX) $(CXXFLAGS) -c -o $@ $<

$(PROG): $(OBJS)
	$(CXX) $(CXXFLAGS) -o $@ $(OBJS)

clean:
	rm -rf $(EXEC) *.o *.out main

rebuild: clean all

```

Mindflayer.cpp   Download

```cpp
/**
 * @file Mindflayer.cpp
 * @author Devin Chen
 * @brief Mindflayer Class
 * @date 2/19/2024
*/
#include "Mindflayer.hpp"

/**
  Default constructor.
  Default-initializes all private members.
  Default Category: ALIEN
  Default summoning: False
*/
Mindflayer::Mindflayer():Creature(){
    setCategory(ALIEN);
    setSummoning(false);
};

/**
  Parameterized constructor.
  @param      : A reference to the name of the Mindflayer (a string)
  @param      : The category of the Mindflayer (a Category enum) with default value ALIEN
  @param      : The Mindflayer's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Mindflayer's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Mindflayer is tame, with default value False
  @param      : The projectiles (a vector of Projectile structs), with default value an empty vector if not provided
  @param      : A flag indicating whether the Mindflayer can summon, with default value False
  @param      : The affinities (a vector of Variant enums), with default value an empty vector if not provided
  @post       : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
Mindflayer::Mindflayer(const std::string &new_name, Category new_type, int new_hitpoint,int new_level, bool new_tame,
std::vector<Projectile> new_projectile, bool new_summon, std::vector<Variant> new_variant)
:Creature(new_name,new_type, new_hitpoint, new_level, new_tame) {
    setProjectiles(new_projectile);
    setSummoning(new_summon);
    setAffinities(new_variant);
};

/**
  Getter for the projectiles.
  @return     : The projectiles (a vector of Projectile structs)
*/
```

```cpp
std::vector<Mindflayer::Projectile> Mindflayer::getProjectiles()const{
    return projectiles_;
}

/**
  Setter for the projectiles.
  @param      : A reference to the projectiles (a vector of Projectile structs)
  @post       : The projectiles are set to the value of the parameter. There should not be any duplicate projectiles in Mindflayer's projectiles vector.
        : For example, if the vector in the given parameter contains the following Projectiles: {{PSIONIC, 2}, {TELEPATHIC, 1}, {PSIONIC, 1}, {ILLUSIONARY, 3}},
        : the projectiles vector should be set to contain the following Projectiles: {{PSIONIC, 3}, {TELEPATHIC, 1}, {ILLUSIONARY, 3}}.
        : If the quantity of a projectile is 0 or negative, it should not be added to the projectiles vector.
        : Note the order of the projectiles in the vector.
*/
    void Mindflayer::setProjectiles(const std::vector<Projectile> &new_projectile){
projectiles_.clear(); // Clear the vector to prevent duplicates
    for (int i = 0; i < new_projectile.size(); i++) {
        const auto& projectile = new_projectile[i];
        if (projectile.quantity_ > 0) {
            bool found = false;
            for (int j = 0; j < projectiles_.size(); j++) {
                auto& p = projectiles_[j];
                if (p.type_ == projectile.type_) {
                    p.quantity_ += projectile.quantity_;
                    found = true;
                    break;
                }
            }
            if (!found) {
                projectiles_.push_back(projectile);
            }
        }
    }
}

/**
  Getter for the summoning.
  @return     : The summoning (a boolean)
*/
    bool Mindflayer::getSummoning() const{
        return summoning_;
    }

/**
  Setter for the summoning.
  @param      : A reference to the summoning (a boolean)
  @post       : The summoning is set to the value of the parameter.
*/
    void Mindflayer::setSummoning(const bool &new_summon){
        summoning_ = new_summon;
    }

/**
  Getter for the affinities.
  @return     : The affinities (a vector of Variant enums)
*/
    std::vector<Mindflayer::Variant> Mindflayer::getAffinities()const{
        return affinities_;
    }

/**
  Setter for the affinities.
  @param      : A reference to the affinities (a vector of Variant enums)
  @post       : The affinities are set to the value of the parameter.
        : There should not be any duplicate affinities in Mindflayer's affinities vector.
        : For example, if the vector in the given parameter contains the following affinities: {PSIONIC, TELEPATHIC, PSIONIC, ILLUSIONARY},
        : the affinities vector should be set to contain the following affinities: {PSIONIC, TELEPATHIC, ILLUSIONARY}.
        : Note the order of the affinities in the vector.
*/
void Mindflayer::setAffinities(const std::vector<Variant>& new_variant) {
    for (int i = 0; i < new_variant.size(); i++) {
        Variant temp = new_variant[i];
        bool found = false;
        for (int j = 0; j < affinities_.size(); j++) {
            if (temp == affinities_[j]) {
                found = true;
                break;
            }
        }
        if (!found) {
            affinities_.push_back(temp);
        }
    }
}
/**
  @param       : A reference to the Variant
  @return      : The string representation of the variant
*/
    std::string Mindflayer::variantToString(const Variant &new_variant){
        switch(new_variant) {
            case PSIONIC:
                return "PSIONIC";
            case TELEPATHIC:
                return "TELEPATHIC";
            case ILLUSIONARY:
                return "ILLUSIONARY";
            default:
            return "NONE";
        }
    }
```

Mindflayer.hpp Download

```cpp
/**
 * @file Mindflayer.hpp
 * @author Devin Chen
 * @brief Mindflayer Class
 * @date 2/19/2024
*/
#include "Creature.hpp"
#include <vector>

class Mindflayer: public Creature{
public:
    enum Variant {PSIONIC, TELEPATHIC, ILLUSIONARY};

    struct Projectile
    {
        Variant type_;
        int quantity_;
    };

/**
  Default constructor.
  Default-initializes all private members.
  Default Category: ALIEN
  Default summoning: False
*/
    Mindflayer();

/**
  Parameterized constructor.
  @param      : A reference to the name of the Mindflayer (a string)
  @param      : The category of the Mindflayer (a Category enum) with default value ALIEN
  @param      : The Mindflayer's hitpoints (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : The Mindflayer's level (an integer), with default value 1 if not provided, or if the value provided is 0 or negative
  @param      : A flag indicating whether the Mindflayer is tame, with default value False
  @param      : The projectiles (a vector of Projectile structs), with default value an empty vector if not provided
  @param      : A flag indicating whether the Mindflayer can summon, with default value False
  @param      : The affinities (a vector of Variant enums), with default value an empty vector if not provided
  @post      : The private members are set to the values of the corresponding parameters.
  Hint: Notice the default arguments in the parameterized constructor.
*/
    Mindflayer(const std::string &new_name, Category new_type = ALIEN, int new_hitpoint = 1,int new_level =1 , bool new_tame = false,
std::vector<Projectile> new_projectile = {}, bool new_summon = false, std::vector<Variant> new_variant = {});

```

```
/**
  Getter for the projectiles.
  @return     : The projectiles (a vector of Projectile structs)
*/
    std::vector<Projectile> getProjectiles()const;

/**
  Setter for the projectiles.
  @param      : A reference to the projectiles (a vector of Projectile structs)
  @post       : The projectiles are set to the value of the parameter. There should not be any duplicate projectiles in Mindflayer's projectiles vector.
        : For example, if the vector in the given parameter contains the following Projectiles: {{PSIONIC, 2}, {TELEPATHIC, 1}, {PSIONIC, 1}, {ILLUSIONARY, 3}},
        : the projectiles vector should be set to contain the following Projectiles: {{PSIONIC, 3}, {TELEPATHIC, 1}, {ILLUSIONARY, 3}}.
        : If the quantity of a projectile is 0 or negative, it should not be added to the projectiles vector.
        : Note the order of the projectiles in the vector.
*/
    void setProjectiles(const std::vector<Projectile> &new_projectile);

/**
  Getter for the summoning.
  @return     : The summoning (a boolean)
*/
    bool getSummoning() const;

/**
  Setter for the summoning.
  @param      : A reference to the summoning (a boolean)
  @post       : The summoning is set to the value of the parameter.
*/
    void setSummoning(const bool &new_summon);

/**
  Getter for the affinities.
  @return     : The affinities (a vector of Variant enums)
*/
    std::vector<Variant> getAffinities()const;

/**
  Setter for the affinities.
  @param      : A reference to the affinities (a vector of Variant enums)
  @post       : The affinities are set to the value of the parameter.
        : There should not be any duplicate affinities in Mindflayer's affinities vector.
        : For example, if the vector in the given parameter contains the following affinities: {PSIONIC, TELEPATHIC, PSIONIC, ILLUSIONARY},
        : the affinities vector should be set to contain the following affinities: {PSIONIC, TELEPATHIC, ILLUSIONARY}.
        : Note the order of the affinities in the vector.
*/
    void setAffinities(const std::vector<Variant> &new_variant);

```

```cpp
/**
  @param       : A reference to the Variant
  @return      : The string representation of the variant
*/
    std::string variantToString(const Variant &new_variant);

    private:
    std::vector<Projectile> projectiles_;
    std::vector<Variant> affinities_;
    bool summoning_;
};
```

README.md Download

```
[![Review Assignment Due Date](https://classroom.github.com/assets/deadline-readme-button-24ddc0f5d75046c5622901739e7c5dd533143b0c8e959d652212380cedb1ea36.svg)](https://classroom.github.com/a/eSTK6Nbh)
# Project2

The project specification can be found on Blackboard

```

test.cpp   Download

```cpp
#include "Creature.hpp"
#include "Dragon.hpp"
#include "Ghoul.hpp"
#include "Mindflayer.hpp"
void newDisplay(const Creature &crea){
    std::cout << "NAME: "<< crea.getName() << "\n";
    std::cout << "CATEGORY: " << crea.getCategory() << "\n";
    std::cout << "HP: " << crea.getHitpoints() << "\n";
    std::cout << "LVL: " << crea.getLevel()<< "\n";
    std::cout << "TAME: " << (crea.isTame()? "TRUE" : "FALSE") << "\n";
    std::cout << std::endl;
    }


void dragonDisplay(const Dragon &Drag){
    newDisplay(Drag);
    std::cout<< "ELEMENT: " << Drag.getElement() << "\n";
    std::cout<< "HEADS: " << Drag.getNumberOfHeads() << "\n";
    std::cout<< "FLIGHT: " << (Drag.getFlight() ? "TRUE" : "FALSE") << "\n";
    std::cout << std::endl;
}

void ghoulDisplay(const Ghoul &g1){
    newDisplay(g1);
    std::cout<< "DECAY: " << g1.getDecay()<< "\n";
    std::cout<< "FACTION: " << g1.getFaction() << "\n";
    std::cout<< "TRANSFORM: " << (g1.getTransformation() ? "TRUE" : "FALSE") << "\n";
    std::cout << std::endl;
}
void MindflayerDisplay(Mindflayer &m2){
    newDisplay(m2);
        for(int i =0; i < m2.getProjectiles().size(); i++){
       auto temp = m2.getProjectiles()[i];
        std::cout << m2.variantToString(temp.type_)<< " : " << temp.quantity_ << "\n";
    }
        std::cout << "SUMMONING: " << (m2.getSummoning() ? "TRUE" : "FALSE")<<"\n";
    if(m2.getAffinities().size()){
        std::cout << "AFFINITIES: \n";

    }
    for (int i = 0; i < m2.getAffinities().size(); i ++){
        auto temp = m2.getAffinities();
        std::cout << m2.variantToString(temp[i])<< "\n";
    }
std::cout << std::endl;
}

int main(){
    Dragon Drag;
    dragonDisplay(Drag);


    Dragon d2("Smog");
    dragonDisplay(d2);

    Dragon d3("Burny", Creature::Category::UNDEAD, 100, 10, true, Dragon::Element::FIRE, 1, true);
    dragonDisplay(d3);

    d3.setElement(Dragon::Element::WATER);
    d3.setNumberOfHeads(2);
    d3.setNumberOfHeads(0);
    d3.setFlight(false);
    dragonDisplay(d3);

    Ghoul g1;
    ghoulDisplay(g1);


    Ghoul g2("Homph");
    ghoulDisplay(g2);

    Ghoul g3("CHOMPER", Creature::Category::ALIEN, 100, 10, true, 3, Ghoul::Faction::FLESHGORGER, true);
    ghoulDisplay(g3);

    g3.setDecay(2);
    g3.setDecay(-20);
    g3.setFaction(Ghoul::Faction::SHADOWSTALKER);
    g3.setTransformation(false);
    ghoulDisplay(g3);


    Mindflayer m1;
    MindflayerDisplay(m1);

std::vector<Mindflayer::Projectile> proj = {{Mindflayer::Variant::PSIONIC, 2}, {Mindflayer::Variant::TELEPATHIC, 1}, {Mindflayer::Variant::PSIONIC, 1}, {Mindflayer::Variant::ILLUSIONARY, 3}};
    std::vector<Mindflayer::Variant> att = {Mindflayer::Variant::PSIONIC, Mindflayer::Variant::TELEPATHIC, Mindflayer::Variant::PSIONIC, Mindflayer::Variant::ILLUSIONARY};
    Mindflayer m2 ("BIGBRAIN", Creature::MYSTICAL, 100, 10, true, proj,
    true,att);
    MindflayerDisplay(m2);

}

```