

## 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 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
1	.DS_Store	
2	.vscode	
3	*.log	
4	*.o	
5	main	

```
1  /*
2  CSCI235 Spring 2024
3  Project 1 - Creature Class
4  Georgina Woo
5  Nov 13 2023
6  Creature.cpp defines the constructors and private and public function implementation of the
   Creature class
7  */
8
9  #include "Creature.hpp"
10
11 /**
12   Default constructor.
13   Default-initializes all private members.
14   Default creature name: "NAMELESS".
15   Booleans are default-initialized to False.
16   Default enum value: UNKNOWN
17   Default Hitpoints and Level: 1.
18 */
19 Creature::Creature(): name_{"NAMELESS"}, category_{UNKNOWN}, hitpoints_{1}, level_{1},
   tame_{false}
20 {
21
22 }
23
24 /**
25   Parameterized constructor.
26   @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.
27   @param    : The category of the creature (a Category enum) with default value UNKNOWN
28   @param    : The creature's hitpoints (an integer) , with default value 1 if not provided, or if the
   value provided is 0 or negative
29   @param    : The creature's level (an integer), with default value 1 if not provided, or if the value
   provided is 0 or negative
30   @param    : A flag indicating whether the creature is tame, with default value False
31   @post     : The private members are set to the values of the corresponding parameters.
32   Hint: Notice the default arguments in the parameterized constructor.
33 */
34 Creature::Creature(const std::string& name, Category category, int hitpoints, int level, bool tame):
   category_{category}
35 {
36     if(!SetName(name))
37     {
38         name_ = "NAMELESS";
39     }
40
41     if(!SetHitpoints(hitpoints))
42     {
43         hitpoints_ = 1;
```

```

44     }
45     if(!setLevel(level))
46     {
47         level_ = 1;
48     }
49     tame_ = tame;
50
51 }
52
53 /**
54  @param : the name of the Creature, a reference to string
55  @post  : sets the Creature's name to the value of the parameter in UPPERCASE.
56           (convert any lowercase character to uppercase)
57           Only alphabetical characters are allowed.
58           : If the input contains non-alphabetic characters, do nothing.
59  @return : true if the name was set, false otherwise
60 */
61 bool Creature::setName(const std::string& name)
62 {
63     if (name.length() == 0)
64     {
65         return false;
66     }
67     else
68     {
69         std::string nameUpper = name;
70         for (int i = 0; i < name.length(); i++)
71         {
72             if (!isalpha(name[i]))
73             {
74                 return false;
75             }
76             else
77             {
78                 nameUpper[i] = toupper(name[i]);
79             }
80         }
81         name_ = nameUpper;
82         return true;
83     }
84 }
85
86 /**
87  @return : the name of the Creature
88 */
89 std::string Creature::getName() const
90 {
91     return name_;
92 }
93
94
95 /**

```

```

96     @param : the category of the Creature (an enum)
97     @post  : sets the Creature's category to the value of the parameter
98     */
99     void Creature::setCategory(const Category& category)
100    {
101        category_ = category;
102    }
103
104
105    /**
106        @return : the category of the Creature (in string form)
107    */
108    std::string Creature::getCategory() const
109    {
110        switch(category_)
111        {
112            case UNDEAD:
113                return "UNDEAD";
114            case MYSTICAL:
115                return "MYSTICAL";
116            case ALIEN:
117                return "ALIEN";
118            default:
119                return "UNKNOWN";
120        }
121    }
122
123    /**
124        @param : an integer that represents the creature's hitpoints
125        @pre   : hitpoints > 0 : Creatures cannot have 0 or negative hitpoints (do nothing for invalid
input)
126        @post  : sets the hitpoints private member to the value of the parameter
127        @return : true if the hitpoints were set, false otherwise
128    */
129    bool Creature::setHitpoints(const int& hitpoints)
130    {
131        if (hitpoints > 0)
132        {
133            hitpoints_ = hitpoints;
134            return true;
135        }
136        else
137        {
138            return false;
139        }
140    }
141
142
143    /**
144        @return : the value stored in hitpoints_
145    */
146    int Creature::getHitpoints() const

```

```

147 {
148     return hitpoints_;
149 }
150
151 /**
152     @param : an integer level
153     @pre   : level > 0 : Characters cannot have 0 or negative levels (do nothing for invalid input)
154     @post  : sets the level private member to the value of the parameter
155     @return : true if the level was set, false otherwise
156 */
157 bool Creature::setLevel(const int& level)
158 {
159     if (level > 0)
160     {
161         level_ = level;
162         return true;
163     }
164     else
165     {
166         return false;
167     }
168 }
169
170
171 /**
172     @return : the value stored in level_
173 */
174 int Creature::getLevel() const
175 {
176     return level_;
177 }
178
179
180 /**
181     @param : a boolean value
182     @post  : sets the tame flag to the value of the parameter
183 */
184 void Creature::setTame(const bool& tame)
185 {
186     tame_ = tame;
187 }
188
189
190 /**
191     @return true if the creature is tame, false otherwise
192     Note: this is an accessor function and must follow the same convention as all accessor functions
193     even if it is not called getTame
194 */
195 bool Creature::isTame() const
196 {
197     return tame_;
198 }

```



```
198
199 /**
200     @post    : displays Creature data in the form:
201     "[NAME]\n
202     Category: [CATEGORY]\n
203     Level: [LEVEL]\n
204     Hitpoints: [Hitpoints]\n
205     Tame: [TRUE/FALSE]"
206 */
207 void Creature::display() const
208 {
209     std::cout << name_ << std::endl;
210     std::cout << "Category: " << getCategory() << std::endl;
211     std::cout << "Level: " << level_ << std::endl;
212     std::cout << "Hitpoints: " << hitpoints_ << std::endl;
213     std::cout << "Tame: " << (tame_ ? "TRUE" : "FALSE") << std::endl;
214 }
```

```
1  /*
2  CSCI235 Spring 2024
3  Project 1 - Creature Class
4  Georgina Woo
5  Nov 13 2023
6  Creature.hpp declares the Creature class along with its private and public members
7  */
8  #ifndef CREATURE_HPP_
9  #define CREATURE_HPP_
10 #include <iostream>
11 #include <string>
12 #include <cctype>
13
14
15 class Creature
16 {
17     public:
18         enum Category {UNKNOWN, UNDEAD, MYSTICAL, ALIEN};
19         /**
20          Default constructor.
21          Default-initializes all private members.
22          Default creature name: "NAMELESS".
23          Booleans are default-initialized to False.
24          Default enum value: UNKNOWN
25          Default Hitpoints and Level: 1.
26         */
27         Creature();
28
29         /**
30          Parameterized constructor.
31          @param    : The name of the creature (a string)
32          @param    : The category of the creature (a Category enum) with default value UNKNOWN
33          @param    : The creature's hitpoints (an integer), with default value 1 if not provided, or if
the value provided is 0 or negative
34          @param    : The creature's level (an integer), with default value 1 if not provided, or if the
value provided is 0 or negative
35          @param    : A flag indicating whether the creature is tame, with default value False
36          @post     : The private members are set to the values of the corresponding parameters.
37          Hint: Notice the default arguments in the parameterized constructor.
38         */
39         Creature(const std::string& name, Category category = UNKNOWN, int hitpoints = 1, int level =
1, bool tame = false);
40
41         /**
42          @param : the name of the Creature, a string
43          @post  : sets the Creature's name to the value of the parameter in UPPERCASE (convert any
lowercase character to upppercase
44                  Only alphabetical characters are allowed.
45                  : If the input contains non-alphabetic characters, do nothing.
```

```

46     @return : true if the name was set, false otherwise
47     */
48     bool setName(const std::string& name);
49
50     /**
51     @return : the name of the Creature
52     */
53     std::string getName() const;
54
55
56     /**
57     @param : the category of the Creature (an enum)
58     @post  : sets the Creature's category to the value of the parameter
59     */
60     void setCategory(const Category& category);
61
62
63     /**
64     @return : the race of the Creature (in string form)
65     */
66     std::string getCategory() const;
67
68     /**
69     @param : an integer that represents the creature's hitpoints
70     @pre   : hitpoints > 0 : Creatures cannot have 0 or negative hitpoints (do nothing for invalid
input)
71     @post  : sets the hitpoints private member to the value of the parameter
72     @return : true if the hitpoints were set, false otherwise
73     */
74     bool setHitpoints(const int& hitpoints);
75
76
77     /**
78     @return : the value stored in hitpoints_
79     */
80     int getHitpoints() const;
81
82     /**
83     @param : an integer level
84     @pre   : level > 0 : Creatures cannot have 0 or negative levels (do nothing for invalid input)
85     @post  : sets the level private member to the value of the parameter
86     @return : true if the level was set, false otherwise
87     */
88     bool setLevel(const int& level);
89
90
91     /**
92     @return : the value stored in level_
93     */
94     int getLevel() const;
95
96

```

```

97     /**
98         @param : a boolean value
99         @post  : sets the tame flag to the value of the parameter
100    */
101    void setTame(const bool& tame);
102
103
104    /**
105        @return true if the Creature is tame, false otherwise
106        Note: this is an accessor function and must follow the same convention as all accessor
functions even if it is not called getTame
107    */
108    bool isTame() const;
109
110    /**
111        @post   : displays Creature data in the form:
112        "[NAME]\n
113        Category: [CATEGORY]\n
114        Level: [LEVEL]\n
115        Hitpoints: [Hitpoints]\n
116        Tame: [TRUE/FALSE]"
117    */
118    void display() const;
119
120    private:
121        // The name of the creature (a string in UPPERCASE)
122        std::string name_;
123        // The category of the creature (an enum)
124        Category category_;
125        // The creature's hitpoints (a non-zero, non-negative integer)
126        int hitpoints_;
127        // The creature's level (a non-zero, non-negative integer)
128        int level_;
129        // A flag indicating whether the creature is tame
130        bool tame_;
131
132 };
133
134 #endif

```

```
1  /**
2   * @file Dragon.cpp
3   * @author Devin Chen
4   * @brief Dragon Class
5   * @date 2/19/2024
6   */
7  #include "Dragon.hpp"
8
9  /**
10   * Default constructor.
11   * Default-initializes all private members.
12   * Default Category: MYSTICAL
13   * Default element: NONE
14   * Default number of head(s): 1
15   * Booleans are default-initialized to False.
16   */
17  Dragon::Dragon(): Creature(), affinity_{NONE}, num_heads_{1}, can_fly_{false} {
18      setCategory(MYSTICAL);
19  };
20
21  /**
22   * Parameterized constructor.
23   * @param    : The name of the Dragon (a reference to string)
24   * @param    : The category of the Dragon (a Category enum) with default value MYSTICAL
25   * @param    : The Dragon's hitpoints (an integer), with default value 1 if not provided, or if the
26   *              value provided is 0 or negative
27   * @param    : The Dragon's level (an integer), with default value 1 if not provided, or if the value
28   *              provided is 0 or negative
29   * @param    : A flag indicating whether the Dragon is tame, with default value False
30   * @param    : The element (an Element enum), with default value NONE if not provided
31   * @param    : The number of heads (an integer), with default value 1 if not provided, or if the value
32   *              provided is 0 or negative
33   * @param    : A flag indicating whether the Dragon can fly, with default value False
34   * @post     : The private members are set to the values of the corresponding parameters.
35   * Hint: Notice the default arguments in the parameterized constructor.
36   */
37  Dragon::Dragon(const std::string &new_name, Category new_category , int new_hitpoint, int
38   new_level, bool new_tame , Element new_element , int new_head, bool new_fly)
39   : Creature(new_name, new_category, new_hitpoint, new_level, new_tame), affinity_{new_element},
40   can_fly_{new_fly} {
41   if(!setNumberOfHeads(new_head)){
42       setNumberOfHeads(1);
43   }
44   };
45
46  /**
47   * Getter for the element.
48   * @return    : The element (a string representation of the Element enum)
49   */
```

```

45 std::string Dragon::getElement()const{
46     switch (affinity_){
47         case NONE:
48             return "NONE";
49         case FIRE:
50             return "FIRE";
51         case WATER:
52             return "WATER";
53         case EARTH:
54             return "EARTH";
55         case AIR:
56             return "AIR";
57     }
58 };
59
60 /**
61  Setter for the element.
62  @param    : A reference to the element (an Element enum)
63  @post     : The element is set to the value of the parameter.
64  */
65 void Dragon::setElement(const Element &new_element){
66     if(new_element >= NONE && new_element <= AIR){
67         affinity_ = new_element;
68     };
69 };
70
71
72 /**
73  Getter for the number of heads.
74  @return   : The number of heads (an integer)
75  */
76 int Dragon::getNumberOfHeads()const {
77     return num_heads_;
78 };
79
80 /**
81  Setter for the number of heads.
82  @param    : A reference to the number of heads (an integer)
83  @pre      : The number of heads is > 0. Do nothing for invalid values.
84  @post     : The number of heads is set to the value of the parameter.
85  @return   : True if the number of heads is set, false otherwise.
86  */
87 bool Dragon::setNumberOfHeads(const int &new_head){
88     if(new_head > 0){
89         num_heads_ = new_head;
90         return true;
91     }
92     return false;
93 };
94
95 /**
96  Getter for the flight flag.

```

```
97     @return      : The flight flag (a boolean)
98     */
99     bool Dragon::getFlight()const{
100         return can_fly_;
101     };
102
103     /**
104     Setter for the flight flag.
105     @param      : A reference to the flight flag (a boolean)
106     @post       : The flight flag is set to the value of the parameter.
107     */
108     void Dragon::setFlight(const bool &new_fly){
109         can_fly_ = new_fly;
110     }
```

```
1  /**
2   * @file Dragon.hpp
3   * @author Devin Chen
4   * @brief Dragon Class
5   * @date 2/19/2024
6   */
7  #include "Creature.hpp"
8  class Dragon : public Creature{
9
10 public:
11     enum Element {NONE, FIRE, WATER, EARTH, AIR};
12     /**
13      * Default constructor.
14      * Default-initializes all private members.
15      * Default Category: MYSTICAL
16      * Default element: NONE
17      * Default number of head(s): 1
18      * Booleans are default-initialized to False.
19      */
20     Dragon();
21     /**
22      * Parameterized constructor.
23      * @param    : The name of the Dragon (a reference to string)
24      * @param    : The category of the Dragon (a Category enum) with default value MYSTICAL
25      * @param    : The Dragon's hitpoints (an integer), with default value 1 if not provided, or if the
26      * value provided is 0 or negative
27      * @param    : The Dragon's level (an integer), with default value 1 if not provided, or if the value
28      * provided is 0 or negative
29      * @param    : A flag indicating whether the Dragon is tame, with default value False
30      * @param    : The element (an Element enum), with default value NONE if not provided
31      * @param    : The number of heads (an integer), with default value 1 if not provided, or if the value
32      * provided is 0 or negative
33      * @param    : A flag indicating whether the Dragon can fly, with default value False
34      * @post     : The private members are set to the values of the corresponding parameters.
35      * Hint: Notice the default arguments in the parameterized constructor.
36      */
37     Dragon(const std::string &new_name, Category new_category = MYSTICAL, int new_hitpoint = 1, int
38     new_level = 1, bool new_tame = false, Element new_element = NONE, int new_head = 1, bool new_fly
39     = false);
40
41     /**
42      * Getter for the element.
43      * @return    : The element (a string representation of the Element enum)
44      */
45     std::string getElement()const;
46
47     /**
48      * Setter for the element.
49      * @param    : A reference to the element (an Element enum)
```



```
45  @post      : The element is set to the value of the parameter.
46  */
47  void setElement(const Element &new_element);
48
49  /**
50   * Getter for the number of heads.
51   * @return    : The number of heads (an integer)
52   */
53  int getNumberOfHeads()const;
54
55  /**
56   * Setter for the number of heads.
57   * @param     : A reference to the number of heads (an integer)
58   * @pre       : The number of heads is > 0. Do nothing for invalid values.
59   * @post      : The number of heads is set to the value of the parameter.
60   * @return    : True if the number of heads is set, false otherwise.
61   */
62  bool setNumberOfHeads(const int &new_head);
63
64  /**
65   * Getter for the flight flag.
66   * @return    : The flight flag (a boolean)
67   */
68  bool getFlight()const;
69
70  /**
71   * Setter for the flight flag.
72   * @param     : A reference to the flight flag (a boolean)
73   * @post      : The flight flag is set to the value of the parameter.
74   */
75  void setFlight(const bool &new_fly);
76
77  private:
78  Element affinity_;
79  int num_heads_;
80  bool can_fly_;
81  };
```

```
1  /**
2   * @file Ghoul.cpp
3   * @author Devin Chen
4   * @brief Ghoul Class
5   * @date 2/19/2024
6   */
7  #include "Ghoul.hpp"
8  /**
9   Default constructor.
10  Default-initializes all private members.
11  Default Category: UNDEAD
12  Default decay: 0
13  Default faction: NONE
14  Booleans are default-initialized to False.
15  */
16  Ghoul::Ghoul():Creature(),level_of_decay_{0}, faction_{NONE},can_transform_{false}{
17      setCategory(UNDEAD);
18  };
19
20  /**
21   Parameterized constructor.
22   @param    : The name of the Ghoul (a reference to string)
23   @param    : The category of the Ghoul (a Category enum) with default value UNDEAD
24   @param    : The Ghoul's hitpoints (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
25   @param    : The Ghoul's level (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
26   @param    : A flag indicating whether the Ghoul is tame, with default value False
27   @param    : The level of decay (an integer), with default value 0 if not provided, or if the value
provided is negative
28   @param    : The faction (a Faction enum), with default value NONE if not provided
29   @param    : A flag indicating whether the Ghoul can transform, with default value False
30   @post     : The private members are set to the values of the corresponding parameters.
31   Hint: Notice the default arguments in the parameterized constructor.
32  */
33  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)
34  :Creature(new_name,new_type, new_hitpoint, new_level, new_tame), faction_{new_faction},
can_transform_{new_transform}{
35  if (!setDecay(new_decay)){
36      level_of_decay_ = 0;
37  };
38  };
39
40  /**
41   Getter for the level of decay.
42   @return    : The level of decay (an integer)
43  */
44
```

```

45 int Ghoul::getDecay() const{
46     return level_of_decay_;
47 }
48
49 /**
50  Setter for the level of decay.
51  @param    : A reference to the level of decay (an integer)
52  @pre      : The level of decay must be >= 0 (do nothing otherwise)
53  @post     : The level of decay is set to the value of the parameter.
54  @return   : true if the level of decay was set, false otherwise
55  */
56 bool Ghoul::setDecay(const int &new_decay){
57     if(new_decay >= 0){
58         level_of_decay_ = new_decay;
59         return true;}
60     return false;
61 };
62
63 /**
64  Getter for the faction.
65  @return   : The faction (a string representation of the Faction enum)
66  */
67 std::string Ghoul::getFaction()const{
68     switch (faction_){
69         case NONE:
70             return "NONE";
71         case FLESHGORGER:
72             return "FLESHGORGER";
73         case SHADOWSTALKER:
74             return "SHADOWSTALKER";
75         case PLAGUEWEAVER:
76             return "PLAGUEWEAVER";
77     }
78 };
79
80 /**
81  Setter for the faction.
82  @param    : A reference to the faction (a Faction enum)
83  @post     : The faction is set to the value of the parameter.
84  */
85 void Ghoul::setFaction(const Faction& faction) {
86     faction_ = faction;
87 }
88
89 /**
90  Getter for the transformation.
91  @return   : The transformation (a boolean)
92  */
93 bool Ghoul::getTransformation()const{
94     return can_transform_;
95 };
96


```

```
97  /**
98   Setter for the transformation.
99   @param    : A reference to the transformation (a boolean)
100  @post     : The transformation is set to the value of the parameter.
101  */
102  void Ghoul::setTransformation(const bool &new_transformation){
103      can_transform_ = new_transformation;
104
105  };
```

```
1  /**
2   * @file Ghoul.hpp
3   * @author Devin Chen
4   * @brief Ghoul Class
5   * @date 2/19/2024
6   */
7  #include "Creature.hpp"
8
9  class Ghoul:public Creature{
10
11  public:
12  enum Faction {NONE, FLESHGORGER, SHADOWSTALKER, PLAGUEWEAVER};
13  /**
14   Default constructor.
15   Default-initializes all private members.
16   Default Category: UNDEAD
17   Default decay: 0
18   Default faction: NONE
19   Booleans are default-initialized to False.
20  */
21  Ghoul();
22
23  /**
24   Parameterized constructor.
25   @param   : The name of the Ghoul (a reference to string)
26   @param   : The category of the Ghoul (a Category enum) with default value UNDEAD
27   @param   : The Ghoul's hitpoints (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
28   @param   : The Ghoul's level (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
29   @param   : A flag indicating whether the Ghoul is tame, with default value False
30   @param   : The level of decay (an integer), with default value 0 if not provided, or if the value
provided is negative
31   @param   : The faction (a Faction enum), with default value NONE if not provided
32   @param   : A flag indicating whether the Ghoul can transform, with default value False
33   @post    : The private members are set to the values of the corresponding parameters.
34   Hint: Notice the default arguments in the parameterized constructor.
35  */
36  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);
37
38  /**
39   Getter for the level of decay.
40   @return  : The level of decay (an integer)
41  */
42  int getDecay() const;
43
44  /**
```

```
45  Setter for the level of decay.
46  @param    : A reference to the level of decay (an integer)
47  @pre      : The level of decay must be >= 0 (do nothing otherwise)
48  @post     : The level of decay is set to the value of the parameter.
49  @return    : true if the level of decay was set, false otherwise
50  */
51  bool setDecay(const int &new_decay);
52
53  /**
54   Getter for the faction.
55   @return   : The faction (a string representation of the Faction enum)
56   */
57  std::string getFaction()const;
58
59  /**
60   Setter for the faction.
61   @param    : A reference to the faction (a Faction enum)
62   @post     : The faction is set to the value of the parameter.
63   */
64  void setFaction(const Faction &faction);
65
66  /**
67   Getter for the transformation.
68   @return   : The transformation (a boolean)
69   */
70  bool getTransformation()const;
71
72  /**
73   Setter for the transformation.
74   @param    : A reference to the transformation (a boolean)
75   @post     : The transformation is set to the value of the parameter.
76   */
77  void setTransformation(const bool &new_transformation);
78
79
80  private:
81  int level_of_decay_;
82  Faction faction_;
83  bool can_transform_;
84  };
```

## ▼ Makefile

 Download

```
1 CXX = g++
2 CXXFLAGS = -std=c++17 -g -Wall -O2
3
4 PROG ?= main
5 OBJS = Creature.o test.o Dragon.o Ghoul.o Mindflayer.o
6
7 all: $(PROG)
8
9 .cpp.o:
10     $(CXX) $(CXXFLAGS) -c -o $@ $<
11
12 $(PROG): $(OBJS)
13     $(CXX) $(CXXFLAGS) -o $@ $(OBJS)
14
15 clean:
16     rm -rf $(EXEC) *.o *.out main
17
18 rebuild: clean all
19
```

```
1  /**
2   * @file Mindflayer.cpp
3   * @author Devin Chen
4   * @brief Mindflayer Class
5   * @date 2/19/2024
6   */
7  #include "Mindflayer.hpp"
8
9  /**
10   Default constructor.
11   Default-initializes all private members.
12   Default Category: ALIEN
13   Default summoning: False
14   */
15  Mindflayer::Mindflayer():Creature(){
16      setCategory(ALIEN);
17      setSummoning(false);
18  };
19
20  /**
21   Parameterized constructor.
22   @param    : A reference to the name of the Mindflayer (a string)
23   @param    : The category of the Mindflayer (a Category enum) with default value ALIEN
24   @param    : The Mindflayer's hitpoints (an integer), with default value 1 if not provided, or if the
value provided is 0 or negative
25   @param    : The Mindflayer's level (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
26   @param    : A flag indicating whether the Mindflayer is tame, with default value False
27   @param    : The projectiles (a vector of Projectile structs), with default value an empty vector if
not provided
28   @param    : A flag indicating whether the Mindflayer can summon, with default value False
29   @param    : The affinities (a vector of Variant enums), with default value an empty vector if not
provided
30   @post     : The private members are set to the values of the corresponding parameters.
31   Hint: Notice the default arguments in the parameterized constructor.
32   */
33  Mindflayer::Mindflayer(const std::string &new_name, Category new_type, int new_hitpoint, int
new_level, bool new_tame,
34  std::vector<Projectile> new_projectile, bool new_summon, std::vector<Variant> new_variant)
35  :Creature(new_name, new_type, new_hitpoint, new_level, new_tame) {
36      setProjectiles(new_projectile);
37      setSummoning(new_summon);
38      setAffinities(new_variant);
39  };
40
41  /**
42   Getter for the projectiles.
43   @return    : The projectiles (a vector of Projectile structs)
44   */
```



```

45  std::vector<Mindflayer::Projectile> Mindflayer::getProjectiles()const{
46      return projectiles_;
47  }
48
49  /**
50   Setter for the projectiles.
51   @param    : A reference to the projectiles (a vector of Projectile structs)
52   @post     : The projectiles are set to the value of the parameter. There should not be any duplicate
projectiles in Mindflayer's projectiles vector.
53       : For example, if the vector in the given parameter contains the following Projectiles:
{{PSIONIC, 2}, {TELEPATHIC, 1}, {PSIONIC, 1}, {ILLUSIONARY, 3}},
54       : the projectiles vector should be set to contain the following Projectiles: {{PSIONIC, 3},
{TELEPATHIC, 1}, {ILLUSIONARY, 3}}.
55       : If the quantity of a projectile is 0 or negative, it should not be added to the projectiles vector.
56       : Note the order of the projectiles in the vector.
57  */
58  void Mindflayer::setProjectiles(const std::vector<Projectile> &new_projectile){
59      projectiles_.clear(); // Clear the vector to prevent duplicates
60      for (int i = 0; i < new_projectile.size(); i++) {
61          const auto& projectile = new_projectile[i];
62          if (projectile.quantity_ > 0) {
63              bool found = false;
64              for (int j = 0; j < projectiles_.size(); j++) {
65                  auto& p = projectiles_[j];
66                  if (p.type_ == projectile.type_) {
67                      p.quantity_ += projectile.quantity_;
68                      found = true;
69                      break;
70                  }
71              }
72              if (!found) {
73                  projectiles_.push_back(projectile);
74              }
75          }
76      }
77  }
78
79  /**
80   Getter for the summoning.
81   @return    : The summoning (a boolean)
82  */
83  bool Mindflayer::getSummoning() const{
84      return summoning_;
85  }
86
87  /**
88   Setter for the summoning.
89   @param    : A reference to the summoning (a boolean)
90   @post     : The summoning is set to the value of the parameter.
91  */
92  void Mindflayer::setSummoning(const bool &new_summon){
93      summoning_ = new_summon;

```

```

94     }
95
96 /**
97  * Getter for the affinities.
98  * @return    : The affinities (a vector of Variant enums)
99  */
100 std::vector<Mindflayer::Variant> Mindflayer::getAffinities()const{
101     return affinities_;
102 }
103
104 /**
105  * Setter for the affinities.
106  * @param    : A reference to the affinities (a vector of Variant enums)
107  * @post     : The affinities are set to the value of the parameter.
108  *           : There should not be any duplicate affinities in Mindflayer's affinities vector.
109  *           : For example, if the vector in the given parameter contains the following affinities: {PSIONIC,
110  *           TELEPATHIC, PSIONIC, ILLUSIONARY},
111  *           : the affinities vector should be set to contain the following affinities: {PSIONIC, TELEPATHIC,
112  *           ILLUSIONARY}.
113  *           : Note the order of the affinities in the vector.
114  */
115 void Mindflayer::setAffinities(const std::vector<Variant>& new_variant) {
116     for (int i = 0; i < new_variant.size(); i++) {
117         Variant temp = new_variant[i];
118         bool found = false;
119         for (int j = 0; j < affinities_.size(); j++) {
120             if (temp == affinities_[j]) {
121                 found = true;
122                 break;
123             }
124         }
125         if (!found) {
126             affinities_.push_back(temp);
127         }
128     }
129 }
130
131 /**
132  * @param    : A reference to the Variant
133  * @return    : The string representation of the variant
134  */
135 std::string Mindflayer::variantToString(const Variant &new_variant){
136     switch(new_variant) {
137         case PSIONIC:
138             return "PSIONIC";
139         case TELEPATHIC:
140             return "TELEPATHIC";
141         case ILLUSIONARY:
142             return "ILLUSIONARY";
143         default:
144             return "NONE";
145     }
146 }

```



```
1  /**
2   * @file Mindflayer.hpp
3   * @author Devin Chen
4   * @brief Mindflayer Class
5   * @date 2/19/2024
6   */
7  #include "Creature.hpp"
8  #include <vector>
9
10 class Mindflayer: public Creature{
11 public:
12     enum Variant {PSIONIC, TELEPATHIC, ILLUSIONARY};
13
14     struct Projectile
15     {
16         Variant type_;
17         int quantity_;
18     };
19
20 /**
21  Default constructor.
22  Default-initializes all private members.
23  Default Category: ALIEN
24  Default summoning: False
25  */
26     Mindflayer();
27
28 /**
29  Parameterized constructor.
30  @param    : A reference to the name of the Mindflayer (a string)
31  @param    : The category of the Mindflayer (a Category enum) with default value ALIEN
32  @param    : The Mindflayer's hitpoints (an integer), with default value 1 if not provided, or if the
value provided is 0 or negative
33  @param    : The Mindflayer's level (an integer), with default value 1 if not provided, or if the value
provided is 0 or negative
34  @param    : A flag indicating whether the Mindflayer is tame, with default value False
35  @param    : The projectiles (a vector of Projectile structs), with default value an empty vector if
not provided
36  @param    : A flag indicating whether the Mindflayer can summon, with default value False
37  @param    : The affinities (a vector of Variant enums), with default value an empty vector if not
provided
38  @post     : The private members are set to the values of the corresponding parameters.
39  Hint: Notice the default arguments in the parameterized constructor.
40  */
41     Mindflayer(const std::string &new_name, Category new_type = ALIEN, int new_hitpoint = 1, int
new_level = 1, bool new_tame = false,
42     std::vector<Projectile> new_projectile = {}, bool new_summon = false, std::vector<Variant>
new_variant = {});
43
```

```

44 /**
45  * Getter for the projectiles.
46  * @return : The projectiles (a vector of Projectile structs)
47  */
48 std::vector<Projectile> getProjectiles()const;
49
50 /**
51  * Setter for the projectiles.
52  * @param : A reference to the projectiles (a vector of Projectile structs)
53  * @post : The projectiles are set to the value of the parameter. There should not be any duplicate
54  * projectiles in Mindflayer's projectiles vector.
55  * : For example, if the vector in the given parameter contains the following Projectiles:
56  * {{PSIONIC, 2}, {TELEPATHIC, 1}, {PSIONIC, 1}, {ILLUSIONARY, 3}},
57  * : the projectiles vector should be set to contain the following Projectiles: {{PSIONIC, 3},
58  * {TELEPATHIC, 1}, {ILLUSIONARY, 3}}.
59  * : If the quantity of a projectile is 0 or negative, it should not be added to the projectiles vector.
60  * : Note the order of the projectiles in the vector.
61  */
62 void setProjectiles(const std::vector<Projectile> &new_projectile);
63
64 /**
65  * Getter for the summoning.
66  * @return : The summoning (a boolean)
67  */
68 bool getSummoning() const;
69
70 /**
71  * Setter for the summoning.
72  * @param : A reference to the summoning (a boolean)
73  * @post : The summoning is set to the value of the parameter.
74  */
75 void setSummoning(const bool &new_summon);
76
77 /**
78  * Getter for the affinities.
79  * @return : The affinities (a vector of Variant enums)
80  */
81 std::vector<Variant> getAffinities()const;
82
83 /**
84  * Setter for the affinities.
85  * @param : A reference to the affinities (a vector of Variant enums)
86  * @post : The affinities are set to the value of the parameter.
87  * : There should not be any duplicate affinities in Mindflayer's affinities vector.
88  * : For example, if the vector in the given parameter contains the following affinities: {PSIONIC,
89  * TELEPATHIC, PSIONIC, ILLUSIONARY},
90  * : the affinities vector should be set to contain the following affinities: {PSIONIC, TELEPATHIC,
91  * ILLUSIONARY}.
92  * : Note the order of the affinities in the vector.
93  */
94 void setAffinities(const std::vector<Variant> &new_variant);
95


```

```

91  /**
92   @param      : A reference to the Variant
93   @return     : The string representation of the variant
94   */
95   std::string variantToString(const Variant &new_variant);
96
97   private:
98   std::vector<Projectile> projectiles_;
99   std::vector<Variant> affinities_;
100   bool summoning_;
101 };

```

## ▼ README.md

 Download

```

1  [[Review Assignment Due Date](https://classroom.github.com/assets/deadline-readme-button-
2  24ddc0f5d75046c5622901739e7c5dd533143b0c8e959d652212380cedb1ea36.svg)]
3  (https://classroom.github.com/a/eSTK6Nbh)
4  # Project2
5  The project specification can be found on Blackboard

```

```
1  #include "Creature.hpp"
2  #include "Dragon.hpp"
3  #include "Ghoul.hpp"
4  #include "Mindflayer.hpp"
5  void newDisplay(const Creature &crea){
6      std::cout << "NAME: " << crea.getName() << "\n";
7      std::cout << "CATEGORY: " << crea.getCategory() << "\n";
8      std::cout << "HP: " << crea.getHitpoints() << "\n";
9      std::cout << "LVL: " << crea.getLevel() << "\n";
10     std::cout << "TAME: " << (crea.isTame()? "TRUE" : "FALSE") << "\n";
11     std::cout << std::endl;
12 }
13
14
15 void dragonDisplay(const Dragon &Drag){
16     newDisplay(Drag);
17     std::cout << "ELEMENT: " << Drag.getElement() << "\n";
18     std::cout << "HEADS: " << Drag.getNumberOfHeads() << "\n";
19     std::cout << "FLIGHT: " << (Drag.getFlight() ? "TRUE" : "FALSE") << "\n";
20     std::cout << std::endl;
21 }
22
23 void ghoulDisplay(const Ghoul &g1){
24     newDisplay(g1);
25     std::cout << "DECAY: " << g1.getDecay() << "\n";
26     std::cout << "FACTION: " << g1.getFaction() << "\n";
27     std::cout << "TRANSFORM: " << (g1.getTransformation() ? "TRUE" : "FALSE") << "\n";
28     std::cout << std::endl;
29 }
30 void MindflayerDisplay(Mindflayer &m2){
31     newDisplay(m2);
32     for(int i = 0; i < m2.getProjectiles().size(); i++){
33         auto temp = m2.getProjectiles()[i];
34         std::cout << m2.variantToString(temp.type_) << " : " << temp.quantity_ << "\n";
35     }
36     std::cout << "SUMMONING: " << (m2.getSummoning() ? "TRUE" : "FALSE") << "\n";
37     if(m2.getAffinities().size()){
38         std::cout << "AFFINITIES: \n";
39
40     }
41     for (int i = 0; i < m2.getAffinities().size(); i++){
42         auto temp = m2.getAffinities();
43         std::cout << m2.variantToString(temp[i]) << "\n";
44     }
45     std::cout << std::endl;
46 }
47
48 int main(){
49     Dragon Drag;
```

```
50 dragonDisplay(Drag);
51
52
53 Dragon d2("Smog");
54 dragonDisplay(d2);
55
56 Dragon d3("Burny", Creature::Category::UNDEAD, 100, 10, true, Dragon::Element::FIRE, 1, true);
57 dragonDisplay(d3);
58
59 d3.setElement(Dragon::Element::WATER);
60 d3.setNumberOfHeads(2);
61 d3.setNumberOfHeads(0);
62 d3.setFlight(false);
63 dragonDisplay(d3);
64
65 Ghoul g1;
66 ghoulDisplay(g1);
67
68
69 Ghoul g2("Homph");
70 ghoulDisplay(g2);
71
72 Ghoul g3("CHOMPER", Creature::Category::ALIEN, 100, 10, true, 3, Ghoul::Faction::FLESHGORGER,
true);
73 ghoulDisplay(g3);
74
75 g3.setDecay(2);
76 g3.setDecay(-20);
77 g3.setFaction(Ghoul::Faction::SHADOWSTALKER);
78 g3.setTransformation(false);
79 ghoulDisplay(g3);
80
81
82 Mindflayer m1;
83 MindflayerDisplay(m1);
84
85 std::vector<Mindflayer::Projectile> proj = {{Mindflayer::Variant::PSIONIC, 2},
{Mindflayer::Variant::TELEPATHIC, 1}, {Mindflayer::Variant::PSIONIC, 1},
{Mindflayer::Variant::ILLUSIONARY, 3}};
86 std::vector<Mindflayer::Variant> att = {Mindflayer::Variant::PSIONIC,
Mindflayer::Variant::TELEPATHIC, Mindflayer::Variant::PSIONIC, Mindflayer::Variant::ILLUSIONARY};
87 Mindflayer m2 ("BIGBRAIN", Creature::MYSTICAL, 100, 10, true, proj,
true,att);
88 MindflayerDisplay(m2);
89
90
91 }
92
```