



# CS 225 Project

## Dungeon Crawl

JAXSON MITCHELL

ZOE MORAWEK

THOMAS MEYERS

# Overview

# Overview: What is the project?

- ▶ The goal of the project is to create a text-based campaign where
  - ▶ Four brave adventurers struggle to find the ever elusive salt-shaker!
- ▶ The adventure was based off Dungeons and Dragons
  - ▶ Adventurers have multiple classes (Knight, Wizard, etc.)
  - ▶ There are multiple different types of enemies.



# Overview: The rubric

- ▶ The code must utilize the following within our project.

## C++ Program and Style Guide (80%)

Your finished project should demonstrate the following concepts we learned in class:

- thoughtful selection of variable names, function names, and class names.
- division of code into multiple header and source code files
- if statements
- while, do-while, and for loops
- functions
- class definitions
- constructors and destructors
- operator overloading
- object composition
- inheritance
- exceptions
- file I/O
- a UML diagram
- demonstration of a C++ command/function/concept that we have not covered in class yet
- good programming practices

# Overview: Repository and Style Guide

- ▶ The code lives in the repository given by
  - ▶ <https://github.com/JaxsonMitchell/CS-225-Project>
- ▶ We have a style guide in our project's README
  - ▶ Camel casing for functions and variables.
  - ▶ Pascal Casing for class names
  - ▶ Comments
    - ▶ File headers
    - ▶ Breaks between sections.
    - ▶ Extraneous comments used for later reference
  - ▶ There are more rules for the style in the README

## CS-225-Project

### Overview

This repository is a simplified text-based DND campaign for our Computer Science II course (CS 225) created by Zoe Marowek, Thomas Meyers, and Jaxson Mitchell. To run the code, you run from main which will call the campaign. Our game utilizes object-oriented programming that is organized through vectors. Each object has its own unique class that all inherits from a common parent class.

### Style Guide

Here are a few necessary guides that are followed in the project.

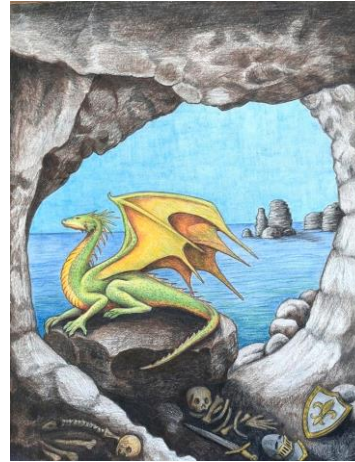
- Function and variable names will be camel-cased.
- Classes adhere to Pascal casing.
- There is a file header at the top of every file giving a basic description of the file as a whole as well as the name of the file.
- Comments are used as breaks between implementations within code to compartmentalize the code better allowing for better organization of the code.
- Comments may add extra clutter to code, so the code itself should explain what is happening through good naming style. \*\* Keep comments sparse \*\* Allow function names and variable names be clear enough to explain what is happening.
- Header files (.h) don't contain function implementation and will be in a separate C++ (.cpp) source file. They only contain class and function headers.

 README.md	Update README.md
 StartGame.cpp	Update StartGame.cpp
 UMLClass.pdf	add UML again...
 battle.cpp	Add if statements to fight function and gameOver function
 battle.h	Update battle.h
 battlers.cpp	Update Dragon's lair battle (battle 4
 battlers.h	Update battlers.h



# Overview: Our Game

- ▶ There are four battles the adventurers must fight through, each of which with different enemies.
  - ▶ Gnome York
  - ▶ The Circus
  - ▶ The Swamp
  - ▶ Dragon's Lair

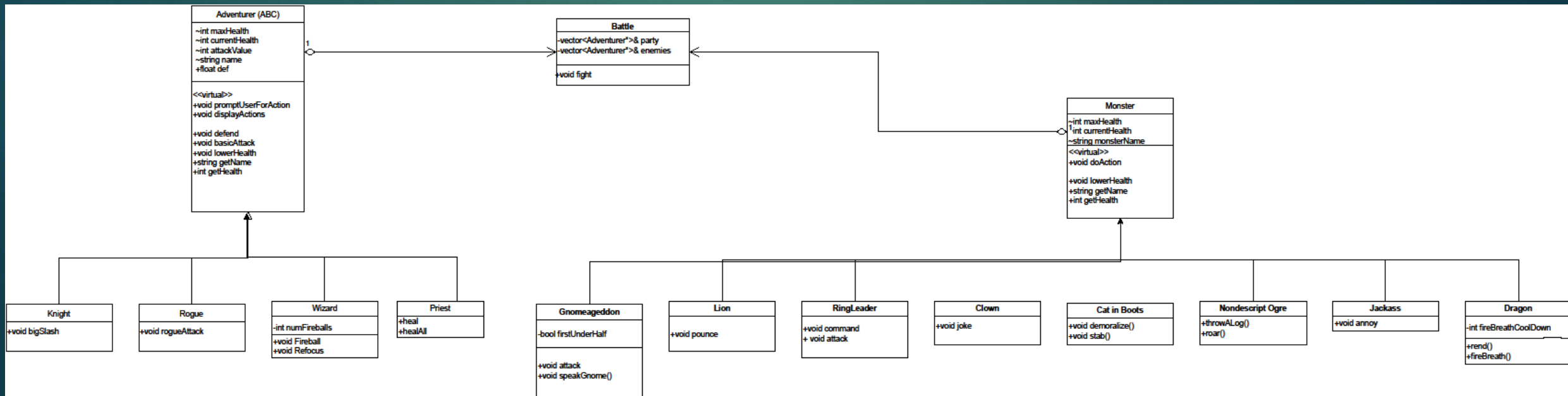




# Implementation and Diagrams

# Class Diagram

- ▶ The UML diagram can be classified into three major sections:
  - ▶ Adventurers
  - ▶ Monsters
  - ▶ Battle





# Implementation: Adventurer Classes

- ▶ These Adventurers contain the following classes:
  - ▶ Abstract Base Class (Adventurer)
  - ▶ Knight Class
  - ▶ Wizard Class
  - ▶ Rogue Class
  - ▶ Priest Class
- ▶ Operator Overloading the insertion operator

```
.....
Adventure Class (Parent)
.....
class Adventurer {
protected:
    int maxHealth;
    int attackValue;
    int currentHealth;
    string name;

public:
    float def = 1;

    // Pure Virtual Functions
    virtual void promptUserForAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) = 0;
    virtual void displayActions() = 0;

    // Basic Functions
    void defend();
    void basicAttack(Monster* M);
    void lowerHealth(int x);

    // Getter Functions
    string getName();
    int getHealth();

    // Default constructors to remove issues
    Adventurer() = default;
    virtual ~Adventurer() = default;

    // Overloading the << operator for ostream
    friend ostream& operator<<(ostream& os, const Adventurer&);
};
```

```
ostream& operator<<(ostream& os, const Adventurer& adventurer) {
    os << adventurer.name << " has " << adventurer.currentHealth << " health left!";
    return os;
}
```

```
.....
Knight Class (Child)
.....
class Knight : public Adventurer {
public:
    void promptUserForAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void displayActions() override;

    void bigSlash(Monster*);
    Knight();
};

.....
Rogue Class (Child)
.....
class Rogue : public Adventurer {
public:
    Rogue();
    void rogueAttack(Monster*);

    void promptUserForAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void displayActions() override;
};

.....
Wizard Class (Child)
.....
class Wizard : public Adventurer {
private:
    int numFireballs;
public:
    void promptUserForAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void fireball(vector<Monster*>& monsters);
    void refocus();
    void displayActions() override;
    Wizard();
};

.....
Priest Class (Child)
.....
class Priest : public Adventurer {
public:
    void promptUserForAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void heal(vector<Adventurer*>& adventurers);
    void healAll(vector<Adventurer*>& adventurers);
    void displayActions() override;
    Priest();
};
```

# Implementation: Monster Classes

```

/*****
Monster Class (Parent)
*****/
class Monster {
protected:
    int maxHealth;
    int currentHealth;
    string monsterName;

public:
    // Pure Virtual Function
    virtual void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) = 0;

    // Basic functions
    void lowerHealth(int healthDropped);

    // Getter functions
    string getName();
    int getHealth();

    // Default constructors to remove issues
    Monster() = default;
    virtual ~Monster() = default;

    friend ostream& operator<<(ostream& os, const Monster& monster);
};

```

```

ostream& operator<<(ostream& os, const Monster& monster) {
    os << monster.monsterName << " has " << monster.currentHealth << " health Left!";
    return os;
}

```

```

/*****
Gnome Class (Child)
*****/
class Gnomeageddon : public Monster {
protected:
    bool firstUnderHalf = true;

public:
    Gnomeageddon();
    ~Gnomeageddon();

    void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void attack(vector<Adventurer*>& adventurers);
    void speakGnome(bool);
};

/*****
Lion Class (Child)
*****/
class Lion : public Monster {
public:
    Lion();
    ~Lion();

    void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void pounce(vector<Adventurer*>& adventurers);
};

/*****
Ringleader Class (Child)
*****/
class Ringleader : public Monster {
public:
    Ringleader();
    ~Ringleader();

    void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void command(vector<Monster*>& monsters, vector<Adventurer*>& adventurers);
    void attack(vector<Adventurer*>& adventurers);
};

/*****
Clown Class (Child)
*****/
class Clown : public Monster {
public:
    Clown();
    ~Clown();

    void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void joke(vector<Adventurer*>& adventurers, vector<Monster*>& monsters);
};

/*****
Cat in Boots Class (Child)
*****/
class CatInBoots : public Monster {
public:
    CatInBoots();
    ~CatInBoots();

    void doAction(vector<Adventurer*>& adventurers, vector<Monster*>& monsters) override;
    void demoralize(vector<Adventurer*>& adventurers);
    void stab(vector<Adventurer*>& adventurers);
};

```

- ▶ There are many monster classes, but it has a similar structure to adventurers
  - ▶ Abstract Base Class (Monster)
  - ▶ Gnomeageddon Class
  - ▶ Lion Class
  - ▶ Ringleader Class
  - ▶ Clown Class
  - ▶ NondescriptOgre Class
  - ▶ CatInBoots Class
  - ▶ Jackass Class
  - ▶ Dragon Class
- ▶ Operator Overloading insertion Operator

# Implementation: Battle Class

- ▶ The Battle class allows for an object oriented approach of having adventurers and monsters battle.
  - ▶ Object Composition is used to store the party and enemies
  - ▶ References are used to ensure when a member gets knocked out, they aren't revived for later fights.
  - ▶ Pointers are used to allow for polymorphism
    - ▶ Abstract Functions: doAction, promptUserForAction

```
class Battle {
private:
    vector<Adventurer*> party;
    vector<Monster*> enemies;
public:
    Battle(vector<Adventurer*> party, vector<Monster*> enemies);
    void fight();
};
```

```

// Battle class
// Battle class
Battle::Battle(vector<Adventurer*> party, vector<Monster*> enemies) : party(party), enemies(enemies) {}

void Battle::fight() {
    sleep(1);
    cout << endl << endl;
    print("\n[e@31m"); // Red font
    cout << "BATTLE BEGIN! \n\n" << endl;
    print("\n[e@3m");
    sleep(1);

    do {
        resetDefenseMultipliers(party);

        // Print monster position numbers
        print("\n[e@32m"); // Green font
        printoutAllMembers(party);
        print("\n[e@3m");
        sleep(1);

        cout << endl;
        print("\n[e@31m"); // Red font
        printoutAllMembers(enemies);
        print("\n[e@3m");
        sleep(1);

        for (auto& adventurer : party) {
            if (enemies.size() != 0) {
                adventurer->displayActions();
                adventurer->promptUserForAction(party, enemies);
                removeIfNoHealth(party);
                removeIfNoHealth(enemies);
            }
        }

        print("\n[e@36m"); // Red font
        for (auto& monster : enemies) {
            if (party.size() == 0) {
                print("\n[e@31m"); // Red font
                cout << "Party is dead" << endl;
                gameOver();
            }
            monster->doAction(party, enemies);
            cout << endl;
            removeIfNoHealth(party);
            removeIfNoHealth(enemies);
            sleep(1);
        }
        print("\n[e@3m");
    } while (neitherVectorsEmpty(party, enemies));

    // If your party all get's knocked out.
    if (party.size() == 0) {
        gameOver();
    }
}
```

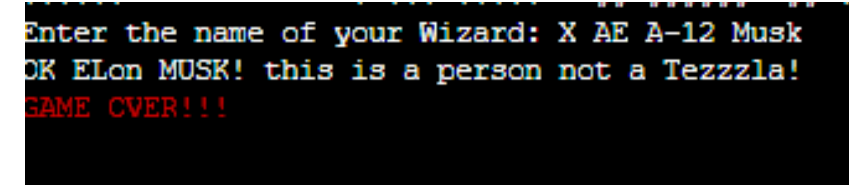
# New C++ Concept

- ▶ Our new function allows for speech of user and monsters to be typed out at specific rates.
- ▶ Used within the function `displayStringLikeText`
  - ▶ Inputs include text, and the rate of text outputted.
  - ▶ The function utilizes threading
- ▶ A thread reads the script and has an order of functions to complete
  - ▶ `Sleep_for` within the code forces the thread to stop for a moment simulating a specified text speed.

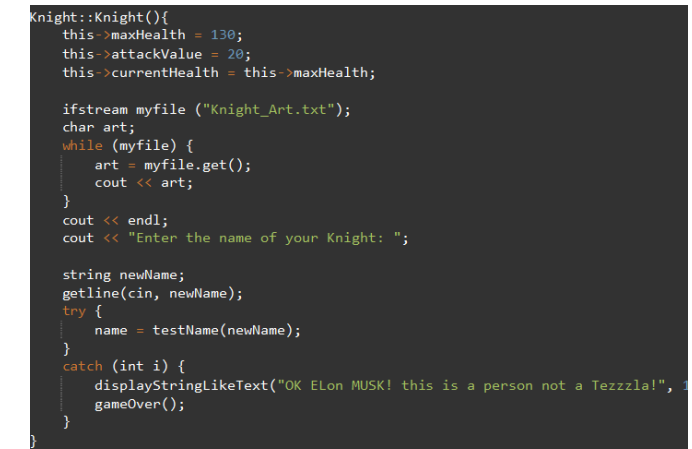
```
void displayStringLikeText(string text, float rateOfMessage) {  
    for (int i = 0; i < text.length(); i++) {  
        cout << text[i] << flush;  
        this_thread::sleep_for(chrono::milliseconds(static_cast<int>(1000 / rateOfMessage)));  
    }  
    cout << endl;  
}
```

# Exceptions

- ▶ Exception handling was utilized in the child classes of the adventurer.
  - ▶ It forces the adventurer's name to not have any numbers within it.
  - ▶ Why should we have numbers, what are we... Elon Musk??
- ▶ We can see below what happens if you name your dear adventurers with a number



```
Enter the name of your Wizard: X AE A-12 Musk
OK ELon MUSK! this is a person not a Tezzzla!
GAME OVER!!!
```



```
Knight::Knight(){
    this->maxHealth = 130;
    this->attackValue = 20;
    this->currentHealth = this->maxHealth;

    ifstream myfile ("Knight_Art.txt");
    char art;
    while (myfile) {
        art = myfile.get();
        cout << art;
    }
    cout << endl;
    cout << "Enter the name of your Knight: ";

    string newName;
    getline(cin, newName);
    try {
        name = testName(newName);
    }
    catch (int i) {
        displayStringLikeText("OK ELon MUSK! this is a person not a Tezzzla!", 1);
        gameOver();
    }
}
```

# File I/O

- ▶ Within the Constructors of the child classes of the Adventurer class, we also have file I/O
  - ▶ Art of the adventurer is displayed as you enter your name
  - ▶ Reading in ASCII art of the adventurers and displaying it to cout

```
ht::Knight(){
    this->maxHealth = 130;
    this->attackValue = 20;
    this->currentHealth = this->maxHealth;

    ifstream myfile ("Knight_Art.txt");
    char art;
    while (myfile) {
        art = myfile.get();
        cout << art;
    }
    cout << endl;
    cout << "Enter the name of your Knight: ";

    string newName;
    getline(cin, newName);
    try {
        name = testName(newName);
    }
    catch (int i) {
        displayStringLikeText("OK ELon MUSK! this is a person not a Tezzzla!", 1
        gameOver();
    }
}
```



# Demo Time?

- ▶ While it's nice to talk about the code, I feel like it's more fun to see it in practice.
- ▶ Let's play



DEMO TIME