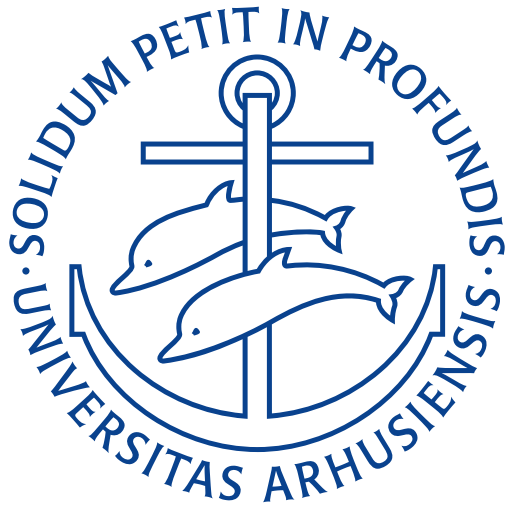
AVANCEREDE PROGRAMMERINGSKONCEPTER

ITTAPK

EXAM REPORT



GROUP 2

Mads E. Jørgensen, AU541836

Name, AU id

Chris W. Olesen, AU453533

Name, AU id

Richard J. Key, AU510495

Name, AU id

Søren Hansen

AU-Vejleder

Aarhus Universitet Ingeniørhøjskolen,
Inge Lehmanns Gade 10
DK-8000 Aarhus C

02/01-2019

Contents

INTRODUCTION | 1

1.1 | ASSIGNMENT EXPLANATION

This report describes the exam project in ITTAPK-01.
The subject for this report is to make a role playing game. The player controls a hero though paths and arenas by choosing, which direction to go. When entering an arena, an enemy will appear, whom will need to be defeated.

The main goal for the assignment is to try and implement and combine as many different C++17 ideoms, patterns and features from the semester and get a better understanding of these. This report will present some of these features and along with the source code give an better understanding of the program.

The game is a one player role playing game with a Hero who have some set stats: healt(HP), attack(ATT) and defence(DEF) and can wear and collect items to enhance either ATT og DEF stats. The HP is set from start and will drop as opponents are fought. When zero is reached the game is over.
Starting the game the player is first asked for a hero name and then off on an adventure! The game asks the player for which path to follow, if the path leads to an arena a battle will take place. If the battle is won a reward in form of an item is dropped from the opponent. At the beginning of each turn the player is asked if he/she wants to change to a new item.

ARCHITECTURE | 2

2.1 | CLASS DIAGRAM

The class diagram is shown on figure 2.2 on the following page it give an overview of the fragments and their individual responsibility as well as the type of connections between them.

2.2 | SEQUENCE DIAGRAM

The sequence diagram is illustrating the users flow through the program. The start and end is excluded for simplicity. But main sets up the game and starts, as mentioned, by first asking the user for a name, then creates a hero and passes it to path of destiny and runs game logic. If a combat results in death the game ends.

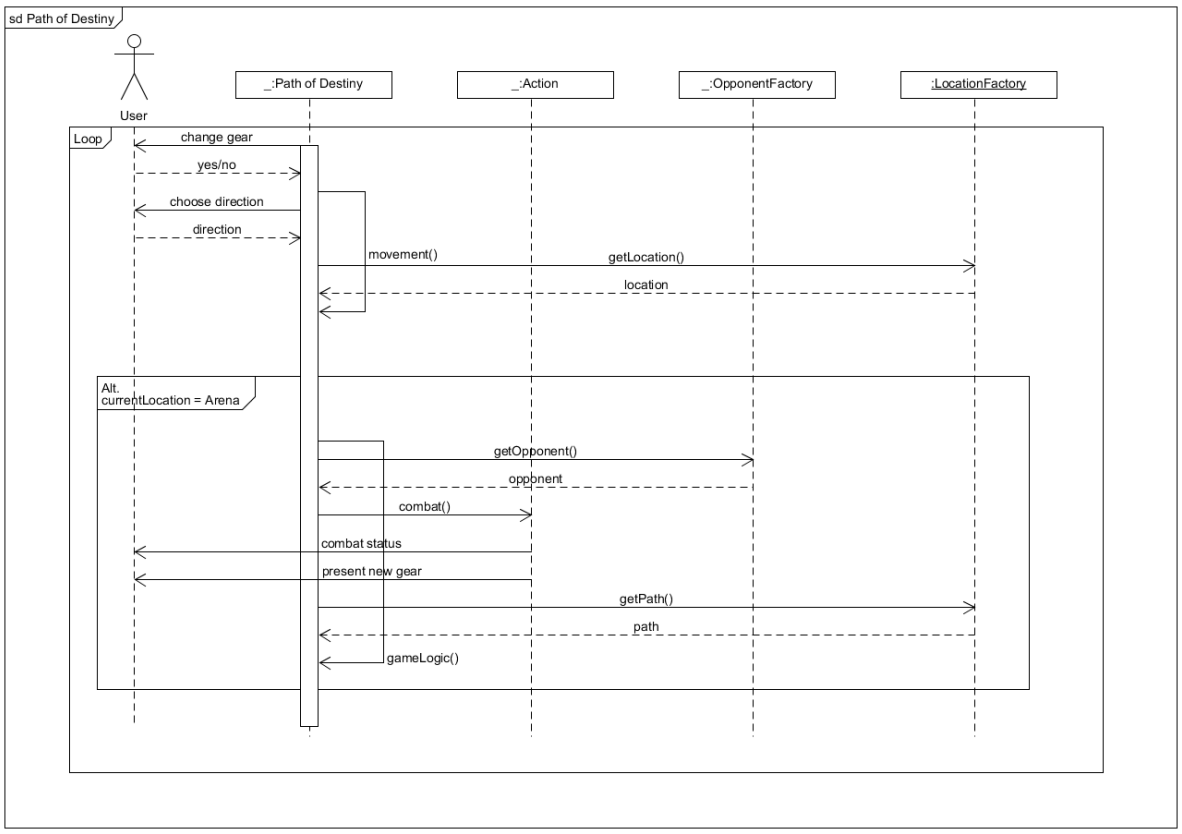


Figure 2.1: Sequence diagram for Path Of Destiny

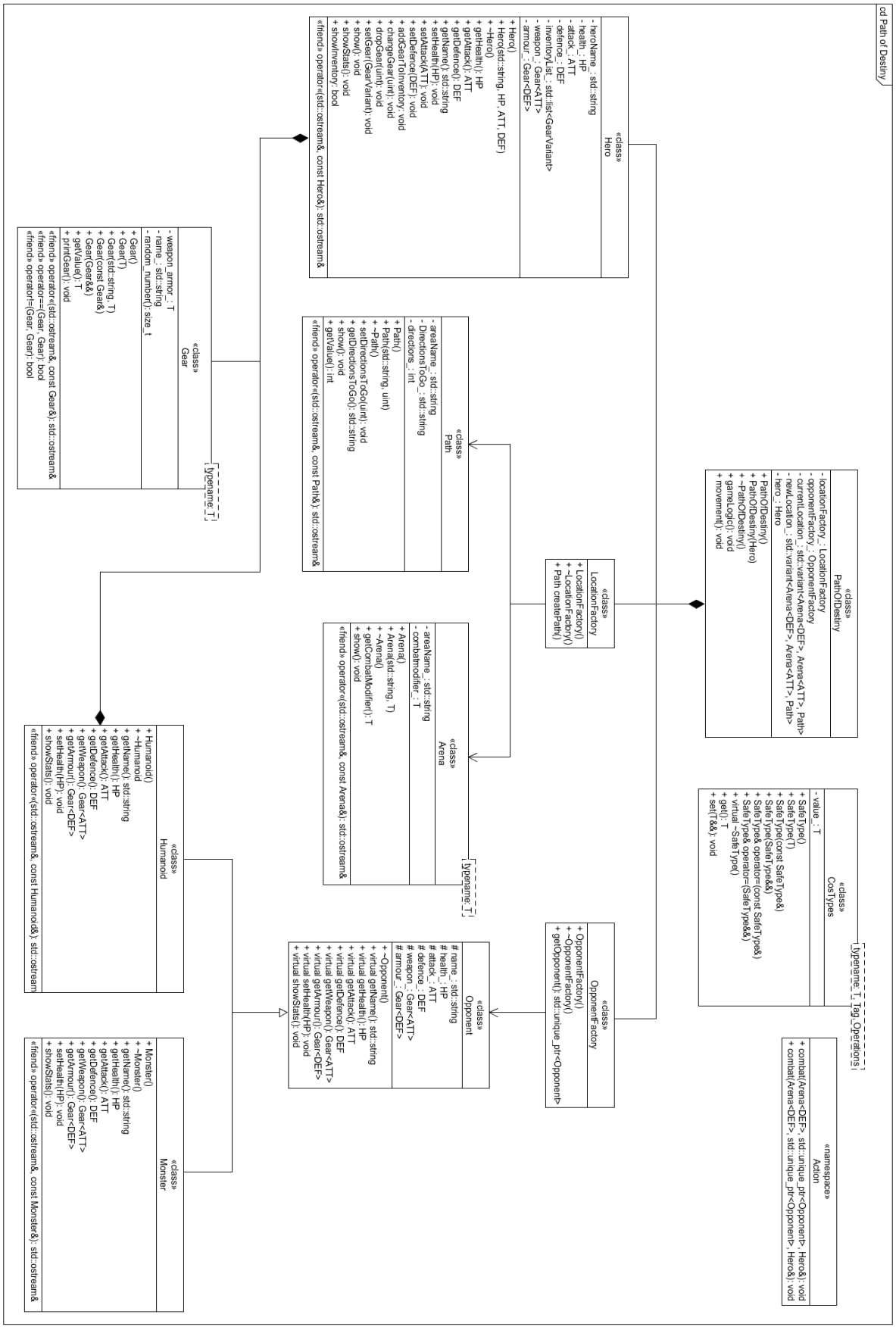


Figure 2.2: Class diagram for Path Of Destiny

DESIGN AND IMPLEMENTATION | 3

3.1 | COSTYPES

CosTypes.HPP is a class calles SafeType for making costume types. Three types are created inside the file; HP (Health Points), ATT (Attack) and DEF (Defence) for describing our character's stats and for defining legal arithmetic.

```
1 template<typename T, typename Tag, template<typename> typename... Operations>
2 class SafeType : public Operations<SafeType<T, Tag, Operations...>>...
3 {
4 // Implementation ...
5
6 using HP = SafeType<int16_t, struct HP_tag, Addition, Subtraction, OutStream>;
7 //using MUL = SafeType<int16_t, struct HP_tag, Multiplication, OutStream>;
```

Listing 3.1: SafeType prototype, one instantiation of SafeType

It contains several interesting features the first of which being the Curiously recurring template pattern, which inherits SafeType from itself. We use this pattern because we know how the class is going to look at compile time and inheriting from it ensures that the type will only be defines once.
The behaviour is decided the compile when creating the type by a variadic input taking structs with different behaviours as seen line 2 and used in 6 and 7 in Listing 3.1. These structs includes the overloaded arithmec operations corresponding to the name (see implementation in listings 3.2 line 1-7). The reason for using a variadic is bacause it makes the generation of SafeTypes generic, so the same constructor can create types with very different behaviours.

```
1  template<typename ST>
2  struct Addition {
3    ST& operator += (const ST& other) {
4      static_cast<ST&>(*this).set(static_cast<ST&>(*this).get() + other.get());
5      return static_cast<ST&>(*this);
6    }
7  };
8
9  // Only subtract ATT from HP (NOT the other way around!)
10 auto operator-(HP first, const ATT& second)
11 {
12   first.set(first.get() - second.get());;
13   return std::move(first);
14 }
```

Listing 3.2: Operations struct, one implementation of a cross type arithmetic

In listings 3.2 line 10-15 is an example of the allowed cross-type arithmetic's implemented in SafeType. Only a handful of these are implemented/allowed but more should have been implemented to avoid using trick and temporary variables in the game logic. This is due to the amount of code it takes to implement since every type needs to handle every outcome of one operator.

Another feature in the SafeType class to highlight is the friendfunctions for the comparison operators. It's implemented with a enable_if<> the reason for this is to include SFINAE (Substitution fail is not an error) See listings 3.3. The way it works is enable_if only allows the compiler to evaluate the succeeding expression if the condition is true. Here it is used along with the typetrait is_same<> to determine if the input type - right hand side (rhs) is the same type as the type your testing. if they are the same type is_same returns = true, which is then inverted (!) to false.
The enable_if will then have a false as an integer and compare it to zero, and since this is true the friendfunction is allowed to be evaluated.

```
template<class Rhs, std::enable_if_t<!std::is_same<Rhs, SafeType>{},int> =0 >
  friend bool operator==(Rhs const& ptr, SafeType) {
    return !*ptr;
  }
}
```

Listing 3.3: (==) comparison as friend in SafeType

3.2 | GEAR

Gear is a template class which is used to create Gear objects for the Hero and Humanoid. The accepted types for Gear are the custom types ATT or DEF, which defines if the gear is a weapon or amour. Gear validates the type input by using a static assertion, to verify that the types provided is the same as either of the custom types mentioned above. The type trait, is same, is used to check that the type provided matches the types that are allowed by Gear. By doing this, the class is less prone to exceptions. Gear has a predefined array containing different Gear names which are chosen based on the type used for the class.

Gear implements rule of five, which includes the move assignment. This overload is implemented using std::swap as seen in listing 3.5

Using static assertion in listing 3.4:

```
static_assert(std::is_same<T, ATT>::value || std::is_same<T, DEF>::value, "GEAR: invalid type");
```

Listing 3.4: Static assertion in Gear.hpp

Using swap in move assignment 3.5:

```
template <typename T>
Gear<T>& Gear<T>::operator=(Gear&& other) noexcept
{
        std::swap(name_, other.name_);
        std::swap(weapon_amour_, other.weapon_amour_);
        return *this;
}
```

Listing 3.5: Move assignment in Gear.hpp

3.3 LOCATIONS

The locations are split in two different types Path and Arena.
The Path is a simple class, where the Arena is a template class based on the safetypes DEF and ATT described in section 3.1 on page 6 depending on whether the arena is affecting the hero attack power or defence power. The effect on the hero is random and generated when the Arena. The locations are generated in a location factory and it is random whether it is an Arena or a Path that is generated.

3.4 CHARACTERS

In Path of Destiny, there are two types of different characters. The Hero which is the character controlled by the User and an Opponent which the Hero fights.

3.4.1 HERO

Hero is a class that defines the object the user controls when playing the game. A hero has a name, health, attack and defence stats as well as the possibility to carry weapons and armour. The weapons and armour is of type Gear and is placed inside a container. A variant is used to be able to place both weapons and armour inside the same container. When changing or dropping Gear, the iterator for the list is retrieved which is then used in the advance function to increment a given number of elements. Get is then called to retrieve the object that was being searched for from the container. Because the object is a variant object, a visitor is used to get the type of the object.

Getting the iterator and using advance in listing 3.6:

```
1 // make iterator
2 auto it = inventoryList_.begin();
3 // set iterator to chosen item
4 advance(it, i-1);
```

Listing 3.6: Using the iterator in Hero.hpp where i is a given number

3.4.2 OPPONENT

Opponent uses the factory pattern to instantiate the opponent objects. The Opponent class is a pure virtual class that is used as an interface for the Monster and Humanoid classes. The Opponent class has most of the same functionality as Hero, except an inventory and the possibility to change gear. Both the Monster class and the Humanoid class inherits from the Opponent interface and is using dynamic polymorphism. The only difference of the implementation of the Monster and Humanoid class is that the Humanoid has a weapon and armour which is created with random values in the constructor. Like the Hero, the Monster and Humanoid uses the custom types, HP, ATT and DEF.

The OpponentFactory class is where the Opponents are instantiated. The getOpponent() method in the factory class creates a random number between 0 and 1 and from this either a Monster or Humanoid is created. The Opponents are created by using a smartpointer, which is of type unique. The object shall be copied and is only avaiable a short amount of time, which makes the Unique pointer a good choice.

Because the unique pointer is in danger of failing when allocation space for the object, a try/catch block is placed around the creation of the unique pointer which catches the bad allocation exception if one should occur.

Creating an Opponent with a try/catch block in listing 3.7:

```
try{
    return std::make_unique<Monster>();
}
catch(std::bad_alloc& ba){
    std::cerr << "Bad allocation caught in OpponentFactory. " << ba.what() << '\n';
}
```

Listing 3.7: Exception safety in OpponentFactory.hpp

3.5 Path of Destiny

The PathOfDestiny.hpp file is the game class, and contains the game logic and the major functions, like move and combat these functions and some of the advanced programming concepts used will be explained.

3.5.1 Combat

The combat function is an overloaded function, depending in the arena type, implemented in a separate namespace called "action" implemented in action.hpp. The combat function clashes the hero and the enemy by subtracting a defending parts defence from the attacking parts attack then reducing the health of the defending part if damage goes thought. The enemy attacking the hero is made as a future, since it the calculation do not depend on the hero attacking the enemy. In this case it might be too extreme since it is a simple calculation, this can be seen in listing 3.8. Since futures can throw exceptions is a try/catch block implemented in case an exception is thrown.

```
try{
    std::future<DEF> f2 = std::async(std::launch::async, [&hero, &arena, &enemy]{
        return hero.getDefence() + arena.getCombatModifier() - enemy->getAttack(); });
    // hero attacks enemy.
    dmg = enemy->getDefence() - hero.getAttack();
    if ( dmg < (DEF) 0) {
        enemy->setHealth((enemy->getHealth() + dmg));
    }
    f2.wait(); // If not done wait
    dmg = f2.get();
```

```
11      if ( dmg < (DEF) 0) {
12          hero.setHealth((hero.getHealth() + dmg));
13      }
14  }
15  catch(const std::future_error& e){
16      std::cerr << "DEF combat future making \n"<< e.what() << '\n';
17  }
```

Listing 3.8: action.hpp: Use of future to calculate enemy attacking hero, with exception safety.

3.5.2 MOVEMENT

The movement functions handles the change of location and the possibility to change the carried gear. This is done by using the implemented functions for the hero and interacting with the player. After changing the gear the current path is presented, and the player is ask to choose which direction to go afterwards a location is generated.
The namespace "action" could also have contained the movement function, since this also is an action the hero performs.
Since the current location is a variant of the types *Arena<DEF>, Arena<ATT>, Path* has a visitor been used to access the variant and get to possible directions to go. The visitor is shown i listing 3.9.

```
1  // the variant to visit
2  using var_t = std::variant<Arena<DEF>,Arena<ATT>,Path>;
3  ...
4  // type-matching visitor: to return posible ways.
5  ways = std::visit([](auto&& arg) -> int {
6      using T = std::decay_t<decltype(arg)>;
7      if constexpr (std::is_same_v<T, Path>){
8          arg.show();
9          return arg.getValue();
10     }
11     else {
12         //Assert if not Path
13         assert(true);
14         std::cout << "ERROR PathOfDestiny::movement - Not Path in currentlocation!\n";
15         return 0;
16     }
17 }, currentLocation_);
```

Listing 3.9: PathOfDestiny.hpp: Use of visitor to access the variant.

In listing 3.9 line 7 in the std::visit() a lambda expression without captures and taking the variant as parameter and returning an integer from the variant, and in the body an decay of the variant type is defined and in a const expression compared using the type trait std::is_same. In this case where a path is expected in order to present the location to the player and return the member indicating the possible ways to move forward.

3.5.3 GameLogic

The function gameLogic is containing the logic that determinate in which order functions are called. It determinates which location type is the current by using a visitor similar to the visitor in the movement function, and if it is a arena. It creates an opponent and calls the combat function after a combat the current location need to be updated to a path since the location type is the only one with movement elements then calling movement. a partly implementation is seen in listing 3.10.

```
1  auto combatDEF = std::bind(static_cast<void(*)(Arena<DEF>, std::unique_ptr<Opponent>,
2  Hero&)>(action::combat),std::placeholders::_1, std::placeholders::_2, hero_);
3  // can catch [&] [this]
4  auto combatATT = [&hero = this->hero_](Arena<ATT> arena, std::unique_ptr<Opponent> enemy)
5  {action::combat(arena, std::move(enemy), hero);};
6  ...
7  combatDone = std::visit([&combatDEF, &combatATT, &enemyFactory = this->opponentFactory_,
8  &hero = this->hero_](auto&& arg) -> bool {
9      using T = std::decay_t<decltype(arg)>;
10     if constexpr (std::is_same_v<T, Arena<DEF>>){
11         std::cout << arg << '\n';
12         //combatDEF(arg,std::move(enemyFactory.getOpponent())); // using the std::bind function
13         action::combat(arg,std::move(enemyFactory.getOpponent()),hero); // normal function call
14         return true;
15     }
16 ...
```

Listing 3.10: PathOfDestiny.hpp: use of std::bind and placeholders and call to combat for Arena<DEF>.

The combat function is bound to always take the hero_ object as hero, using std::bind and std::placeholders as see in listing 3.10 line 1. In the bind a static_cast is used to determinate which overload of combat the bind is for, then the placeholders and bind parameters is defined, here the hero_. In line 4 a bind for the combat function using Arena<ATT> only using a lambda expression. This lambda captures the this pointer hero_, since the hero_ is needed in the combat function, then the parametres Arena<ATT> and an unique pointer to an opponent, in the lamdba body a call to the combat function using the captured hero and the parametres recieved. The use of std::move for passing the unique pointer is intended, since the opponent always needs to be destoyed when leaving the combat function. However these binds do not work as intended as the combat prototype decleares a reference to the hero_ is wanted, but these binds makes a copy of the hero_ and passes the same hero with every combat call, and not an updated hero with damage taken and added inventory. That is why line 12 in listing 3.10 is a comment and line 13 exists.

FUTURE WORK | 4

Since implementation of software is an iterative process and the first use of new concepts are expected to need some more iterations in order to have every new concept working as intended. The developers of this project could with extended time have ability to make some of the less elegant solutions more elegant and implemented some of the flaws mentioned earlier. Further could some of the ideas listed below been implemented as well:

- Add a monster defeated/ location passed counter as a score marker.
- Add directions in an arenas removing the need to generate a new path object after a combat.
- allow user to drop items when in changing gear, to reduce inventory size
- allow player to flee/try to flee combat.
- resting heal some small amount of health. getting a path and not a arena.
- custom types as literals, increasing health attack and defence values.

5 Conclusion

The goal for this project was to develop a project using advanced programming concepts in C++. The developers for this project has successfully implemented a game using many of these concepts. Even though there are a few flaws, the final outcome satisfies the requirements set by the developers. A lot of knowledge regarding C++ has been acquired during the process of developing this game.