Department of Informatics, Technical University of Munich
Scientific Computing in Computer Science
G. Chourdakis, J.-M. Gallard, F. Simonis

Winter term 2019/20

Advanced Programming Tutorial
Project, Sprint 2: Player stats and Fighting.

Sprint description

In this second phase, the player character and enemies are expanded to include RPG elements, that will be used to produce the fighting system (currently against a dummy enemy).

This phase will mostly focus on basic OOP features as well as basic `unique_ptr` manipulations.

User stories

Character stats

As a player, my character has the following attributes:

- a name
- attack points
- defense points
- health points (HP)
- max health point

They are all displayed when chosing the "Show Stats" option in the game.
Hint: Introduce a class with the required fields and a public method to display the stats.

Passing time

In the game loop, when the action menu appears, it tells me how many in-game days have passed. Fighting takes a day and saving/reloading the game keep the day count.

Resting

In the game menu, I can also chose to rest to restore my HP to full HP defined by the max HP stat. Resting takes an in-game day.

Hint: Use a class method to restore the HP (see the technical hints, encapsulation).

Fighting

When I chose to fight, an enemy is created (for this sprint it can always be the same dummy enemy). The fight start by displaying the enemy name (e.g. by saying "A Gobelin appears") The fights takes place in rounds, at each round the damage inflicted is calculated by substracting the attacker's attack to the defender's defense (note that damage cannot be lower than 0) and is substracted to the defender HP. The damage inflicted and received by the player is displayed. Rounds occur as long as both opponents are alive.

Hint: For now an enemy can simply be an object of the same class as the player.

Save folder

When I save the game, my save file is located in a dedicated "saves" folder and when I load a game the save file is picked up from this folder.

Technical Hints

This section describes some problems that will be encountered during this sprint. It also provides some technical hints you are free to take into consideration. You can also come up with your own solution as long as the user stories are fulfilled.

Considerations regarding the player class

When writting a class the following should be considered:

Public vs Private

For each field, should it be public (everything can access and change it) or private (only the class methods can access and change it) with if required public getter and setter functions to set and access the value of it? Public fields are easier to use but restricting the access enforces good practice, for example the name field should not be changed once an object is created, having it being private with no setter function is a good way to enforce it.

Note that a third level of access exists, protected, and may become useful later when inheritance will be introduced but is currently not needed.

Encapsulation

A class is a black box, as such it makes sense to give it responsabilty to handle some behaviors previously handled by the engine. For example restoring a character to full health by resting is something the character class should be doing itself with a public method `void rest()` in the class instead of relying on the engine changing some internal class field. The concept of encapsulation is a very important aspect of OOP to facilitate the growth of a project. In this example if we later introduce another considerations when resting (e.g. also restoring mana for a mage), the engine would not require any change as it would just be calling the `rest` method. The engine itself does not need to know what resting should do to the inner fields of an character, it handles player and enemy characters but not their inner working.

Testing if a character is alive is another valid thing to encapsulate behind an `bool isAlive()` method.

Dealing with pointers

As the player will be an object that need to stay valid as long as the game run, it will need to be allocated on the Heap. Use modern C++11 `std::unique_ptr<T>` and the `std::make_unique<T>` function to create it.

When using an unique pointer, think about ownership: who is responsible for this pointer and for deleting it properly later. So when a method needs to work with the player object, does the method needs to take ownership of the player object by moving the pointer or does it only need to use the object to perform some action with it? If ownership is not required, the object is passed as reference (e.g. `foo(*myUniquePtr)`) and the ownership of the pointer is kept where it was. For example in this project, the engine will have ownership of the player pointer, thus, it needs to take ownership of it if it is created somewhere else (for example when loading if you use a factory or a deserialize function, see below), but on the other hand, the fight method only require to manipulate the player object, not take ownership of the pointer.

For the fight method, an enemy object is needed, it can be created before passing it to the fight method of the engine. Be careful that fighting multiple times should work, and not reuse the dead enemy fron the previous fight, whithout causing a memory leak.

Saving and loading: serialization

The player object should be able to be saved in a file. For this you are faced with the problem of "serialization" (= tranforming an object into a string and vice-versa).

A good way to do it is to have a `serialize` function that take an object and return a string encoding all the relevant informations to be able to recreate a clone of this object. Then `deserialize` function doing the opposite: taking a string and creating a new object with the contained information. Deserializing may require dealing with moving the created `unique_ptr`.

Another way to do serialization is to have the serialize function as a method of the class and the class to have a constructor working with a string as input to build the full object. However this way will still require an outside deserialize function (or factory pattern, will be discussed next sprint) if multiple type of object can be serialized and deserialized as only the correct constructor should be called.

Any format can be chosen to store the data in the string. A simple one that will suffice for now is csv which would look like this "Bob;55;100;25;10;" to store a player named Bob with 55/100HP, 25 atk and 10 def.

Project cleanup (optional)

Moving the source code to an "src/" subfolder can help make the project cleaner.

The engine itself can also be refactored to be class used by the main function. This way the in-game day counter is a field of the class and it has a constructor to initialize a new game and one to automatically restore the game to a save file state.