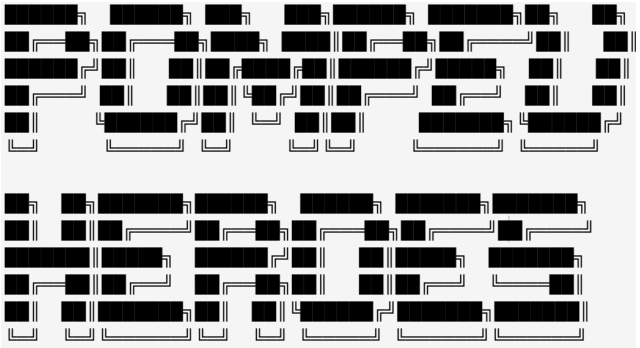
Lab Project - RPG Game:



DATA STRUCTURES AND ALGORITHMS II

(Course 2023/24, 29th May 2024)

Alan Le Roux Osorio - u233515

Oriol Roca Martín - u232892

Eduardo Tobia Rubert - u233480

Pau Martínez Carrión - u232318

CONTENTS

1. INTRODUCTION

This project, titled Pompeu Heroes, is a turn-based RPG video game that immerses players in the unique and challenging environment of Pompeu Fabra University. In this game, players step into the shoes of a new university student navigating the trials and tribulations of the first trimester of their first year of engineering undergraduate studies. The primary goal is to successfully pass the final exam of the first trimester, but first they have to earn the necessary credits to be able to face the final exam.

Players will need to strategically manage their time and resources by attending various activities such as theory classes, practice sessions, and study periods in the library (that is, by passing through different scenarios). Each of these activities helps the player gain knowledge and skills essential for passing the different challenges they will encounter. The game presents the player with different enemies that metaphorically mimic real-life university experiences, and the way of passing them is by winning turn-based fights using different attacks and obtained skills, adding layers of strategy and depth to the gameplay.

The concept for Pompeu Heroes was inspired by our own experiences during our first year at this university. We wanted to capture the excitement, stress, and sense of accomplishment that comes with dealing with the first year of university life. By transforming these experiences into a game, we aimed to create something both fresh and entertaining, providing players with a comic, sometimes nostalgic, and engaging adventure.

In this report we will explain how we have deployed our programming knowledge to carry out this project, what concepts and ideas we have used to make the game as original as possible, and how we have been able to solve the problems that we have gradually encountered while writing the code for this video game.

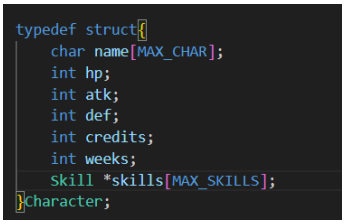
2. PROJECT OBJECTIVES

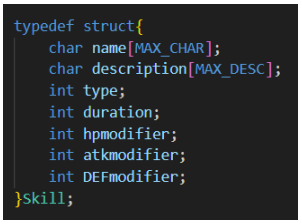
2.1. MANDATORY OBJECTIVES MET

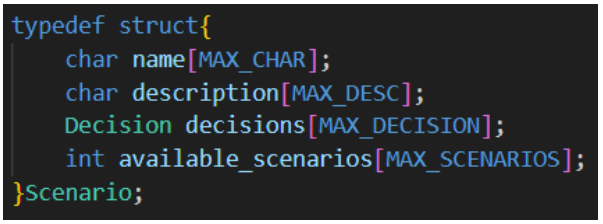
Objective 1: Entities and structs definitions

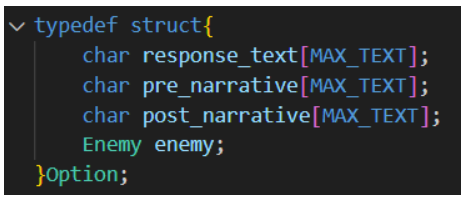
To start implementing and developing the game, we first need to begin with the most basic step, which is declaring the structures of the crucial elements of the game (characters, enemies, scenarios, etc.). Thus, the implementation of character structures is crucial for defining both enemy and team entities in our game. These structures encapsulate all the necessary attributes and behaviors for the characters within the game.

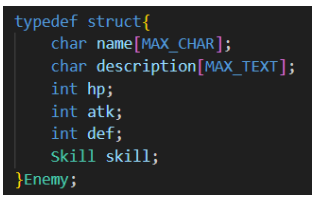
In the figure below, screenshots of all the structs we have used in our code:

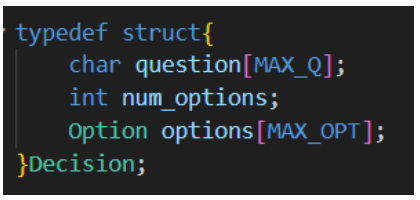












The playable character (player) is implemented by a Character structure. This structure includes the following variables: name, health points (hp), attack points (atk), defense points (def), credits, weeks, and an array of skills. The name variable holds the character's name, which will be input at the beginning of the game by the player, bringing a more personalized experience to the game. The hp, atk, and def variables represent the character's health, attack power, and defense capabilities, respectively, which are crucial for determining the character's performance in battles. The credits variable represents the in-game currency or points the character possesses, and the week's variable tracks the progression of time within the game. Additionally, the skills array stores pointers to the character's skills, allowing each character to have a unique set of abilities that can influence combat and other interactions within the game.

The Skill structure defines the abilities characters can use, including fields for name, description, type, duration, and modifiers for hp, atk, and def. The name and description explain the skill's purpose, while the type indicates if it is a temporary modifier or a direct attack. The duration specifies how long the modifier lasts, and the modifiers adjust the character's stats, enabling dynamic changes in combat.

As well as the character, the enemy has its own structure that defines the name, description, hp, atk, def, and the skill of the enemy. The name and description provide details about the enemy, while hp, atk, and def determine its combat capabilities. The skill field gives each enemy a unique ability, adding variety and challenge to battles.

The decision structure is used to define choices that players can make in the game, influencing the narrative and encounters. It includes a question, the number of options, and an array of options. The question presents the decision to the player, prompting them to make a choice. The num_options indicates how many choices are available, and the options array stores the possible responses, each represented by an Option structure.

Then, the option structure represents a single choice in a decision, including response text, pre-narrative text, post-narrative text, and an enemy. The response_text captures the player's choice, while the pre_narrative and post_narrative fields enhance storytelling by setting the scene before and after the encounter. The enemy field defines the adversary faced if this option is chosen, linking decisions to the combat system.

Finally, the scenario structure defines name, description, an array of decisions, and a list of available scenarios. The name and description provide context for the scenario. The decisions array stores possible player choices, each leading to different outcomes and encounters. The available_scenarios field lists scenarios accessible from the current one, allowing the game to progress based on the player's decisions and ensuring a branching narrative.

All the structs written are located in a header file named “structures.h”. Overall, the time complexities are either constant or linear, bounded by the predefined maximum sizes (“MAX_SKILLS”, “MAX_DECISION”, “MAX_OPT”).

Furthermore, the initial time it took us to create the structures was around 30 minutes, but as we have done so many changes in the structs during all the projects we could conclude that the total time could be around 3 hours more or less.

Objective 2: Scenarios and narrative

The main narrative of the game is: Once you have passed the "Selectivitat" exam, the chosen player will begin their university adventure at Pompeu Fabra University. The game will involve surviving the first trimester of university by attending theoretical classes, practical classes, or simply studying in the library to gain enough knowledge to face the final exam. The game is divided into the four aforementioned scenarios (theoretical class, practical class, library, and final exam), and each decision the player makes within these scenarios will have repercussions and influence. Once the remaining weeks are over, the player will proceed to the final scenario (final exam). The time taken to figure out the whole plot of the game and the scenarios used was around 3 hours in total.

Objective 3: Interactive menu (UI)

We made a very basic menu that has only three options. In the first option, you can load a tutorial that explains and teaches you the basics of the fight flow game with a demonstration fight.
Then, the second option is the “new game” option, where you start your adventure at Pompeu Heroes, inside this option the player customize their character and then the normal operation of the game begins.
Finally, the third option is to exit the game, and in case the player enters an invalid option, the code prints a message that says "Invalid choice. Please try again".

```c
void print_menu() {
    printf("1. Tutorial\n");
    printf("2. New Game\n");
    printf("3. Exit\n");
}

int main() {
    int choice;

    while (1) {
        print_menu();
        scanf("%d", &choice);

        switch (choice) {



            case 1:
                tutorial();
                break;
            case 2:
                new_game();
                break;
            case 3:
                printf("Exiting the game.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

The time taken to create the menu was not too much, however the contents inside the menu took around 8 hours to develop, and the time complexity of the print_menu() function is O(1), which means it operates in constant time regardless of the input size. This is because the function simply prints a fixed number of lines, independent of any input parameters.

Objective 4: Skill System:

```c
typedef struct{
    char name[MAX_CHAR];
    char description[MAX_DESC];
    int type;
    int duration;
    int hpmodifier;
    int atkmodifier;
    int DEFmodifier;
}Skill;
```

Skills can have multiple modifiers from the player or enemy stats. These are limited by duration (number of uses), and in the activation, by handling negative number errors.

```c
void createSkills(Skill *skills){
    //first skill
    strcpy(skills[0].name,"Laptop shield");
    strcpy(skills[0].description," You bought a laptop that gives you some knwoledge and this
    skills[0].duration = 1;
    skills[0].type = SINGLE_ROUND_MODIFIER;
    skills[0].hpmodifier = 15;
    skills[0].atkmodifier = 0;
    skills[0].DEFmodifier= 0;
    // second skill
    strcpy(skills[1].name,"Mr. Taixas particular classes");
    strcpy(skills[1].description,"You had some particular classes from a good teacher and he g
    skills[1].duration = 4;
    skills[1].type = MULTIPLE_ROUND_MODIFIER;
    skills[1].hpmodifier = 0;
    skills[1].atkmodifier = 10;
    skills[1].DEFmodifier= 0;
    // third skill
    strcpy(skills[2].name,"Sharp Pen");
    strcpy(skills[2].description,"This pen can be used once to stab your opponent, giving a hu
    skills[2].duration = 1;
    skills[2].type = SINGLE_ROUND_MODIFIER;
    skills[2].hpmodifier = 0;
    skills[2].atkmodifier = 10;
    skills[2].DEFmodifier= 0;
```

Initializing the skills that the player will choose.

```c
player->def += chosen_skill->DEFmodifier;
player->hp += chosen_skill->hpmodifier;
player->atk += chosen_skill->atkmodifier;
```

The structure definition is at structures.h file, the implementation at combat.c, as they are activated during the fight, and the initialisation of all the skills, at the main.c file, when starting the game.

There are a total of 16 skills. The first 8 are only accessible for the player, and the next 8 are done only for enemies. At the beginning of the game, the player chooses its skills, and during the game, enemies are randomly selected.

Objective 5: Queue-based turns:

```c
typedef struct Node
{
    void *fighter; // since it can be either a Charater* or an Enemy* we use void
    struct Node *next;
} Node;
// define the structure of a queue
typedef struct Queue
{
    Node *first; // points to the first of the queue
    Node *last;  // points to the last of the queue
} Queue;
```

Definition of structures needed.

```c
Node *newNode(void *fighter)
{
    Node *temp = (Node *)malloc(sizeof(Node));
    temp->fighter = fighter;
    temp->next = NULL;
    return temp;
}
```

Adding nodes function.

```c
void enqueue(Queue *queue, void *fighter)
{
    // create a new node
    Node *node = newNode(fighter);

    // If the queue is empty, then the new node is both first and last
    if (queue->last == NULL)
    {
        queue->first = node;
        queue->last = node;
        return;
    }

    // Add the new node at the end of the queue and update last
    queue->last->next = node;
    queue->last = node;
}
```

Enqueuing function.

```c
Queue *createQueue()
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->first = NULL;
    queue->last = NULL;
    return queue;
}
```

Initializing a queue function.

The utility and explanation of queue-based turns is in the next section, as it involves the fight flow section.

Objective 6: Fight flow:

Design the flow of turn-based combat. On the player's turn, prompt the available skills' menu. On CPU's turn, randomly choose an enemy's skill. (6h)
Basically, with everything commented before on the character variables (attack, defense, health), the different enemy and player kills, the queue, we can now explain the main flow of the fight…

The start of the fight is random, with basic assignments on which turn enqueues first.

```
if (rand() % 2 == 0)
{ // 50% chance
    enqueue(queue, player);
}
else
{
    enqueue(queue, enemy);
}
```

Then, the turn starts by dequeuing, as a simple queue.

```
if (dequeue(queue) == player) // ---------------------- TURN OF THE PLAYER
{
```

The turn is done, and the player will have to input which skill to use. The stats of both, player and enemy, are updated, and to finish the turn, it enqueues the opposite fighter.

```
enqueue(queue, enemy);
```

This will repeat in a loop, until someone dies. At each iteration of the loop it checks who died, and returns the result to the main file. All the fight flow occurs at the combat.c file.

Objective 7: Scenario Graph:

For the scenario graph, we tackled it with the following method:

Instead of doing new structures for nodes, starting nodes, edges… we added this information as supplementary data in the scenario structure.

```
typedef struct{
    char name[MAX_CHAR];
    char description[MAX_DESC];
    Decision decisions[MAX_DECISION];
    int available_scenarios[MAX_SCENARIOS];
}Scenario;
```

This part of the code is in the structures' header file called “structures.h”.

The integer array, “available_scenarios”, stores 1 if there is an edge to the index of another scenario, or 0 if there is no edge. These numbers are inserted when initializing the scenarios and once stored, they are not modified again.

```
scenario->available_scenarios[0] = 0;
scenario->available_scenarios[1] = 1;
scenario->available_scenarios[2] = 1;
scenario->available_scenarios[3] = 0;
```

This part of the code is in the main.c file.

The size of the array is always 4 (the number of scenarios). In this image, we see the initialisation of scenario 1: index 0 is for scenario 1, so obviously, it is 0 as it can not go back to its own scenario. Index 1 and 2 represent scenarios 2 and 3, respectively. As it stores a 1, it means that there is an edge. Finally, scenario 4 is the final exam (final boss), so it can only be entered when all scenarios have been played.

This method works well as it has an asymptotic complexity of O(n), when searching for the scenarios that can be entered. Bearing in mind that we would only have 4 scenarios, an asymptotic complexity of O(n) is pretty optimal. Having said that, if we were to increase the number of scenarios in the game up to a really large amount, we would have to look for some other algorithm that can handle the search in a logarithmic manner, maybe.

```
int choose_scenario(Scenario scenarios[MAX_SCENARIOS], int actual_position){
    char option;
    printf("Choose the scenario you want to play: \n");

    for(int i = 0 ; i < MAX_SCENARIOS ; i++){
        if(scenarios[actual_position].available_scenarios[i] == 1 ){
            printf("%d. %s \n", i+1, scenarios[i].name );
        }
    }

    while (true){
        switch(actual_position){
            case 0:
                scanf("%s",&option);
                switch (option) {
                    case '2':
                        return 1;
                        break;
                    case '3':
                        return 2;
                        break;
                    default:
                        printf("Invalid choice! Please enter a number between 1 and 3.\n");
```

This part of the code is in the main.c file.

The function in charge of asking the next scenario to the user and checking if it is available is choose_scenario. It takes as input all the scenarios, and the index of the current one. All the scenario graphs took around 6h considering the day was done and all the times it had problems and had to be adapted.

Objective 8: Start and End Nodes:

The first scenario, the starting node, of our game is the “Theory class” scenario, where the player has a basic introduction of how a normal class at university is, and depending on the decisions made by the player it will face a different enemy. Once the first scenario is completed, the player can choose which scenario to do next, meaning that there is no linear order of scenarios throughout the game. However, once the week's counter (the player's remaining weeks) reaches zero, the final scenario, the final exam, appears regardless of which scenario the player is currently in. This can be considered the final node of our game. Determining this took us about 2 hours of collaborative work.

Objective 9: Scenario Logic:

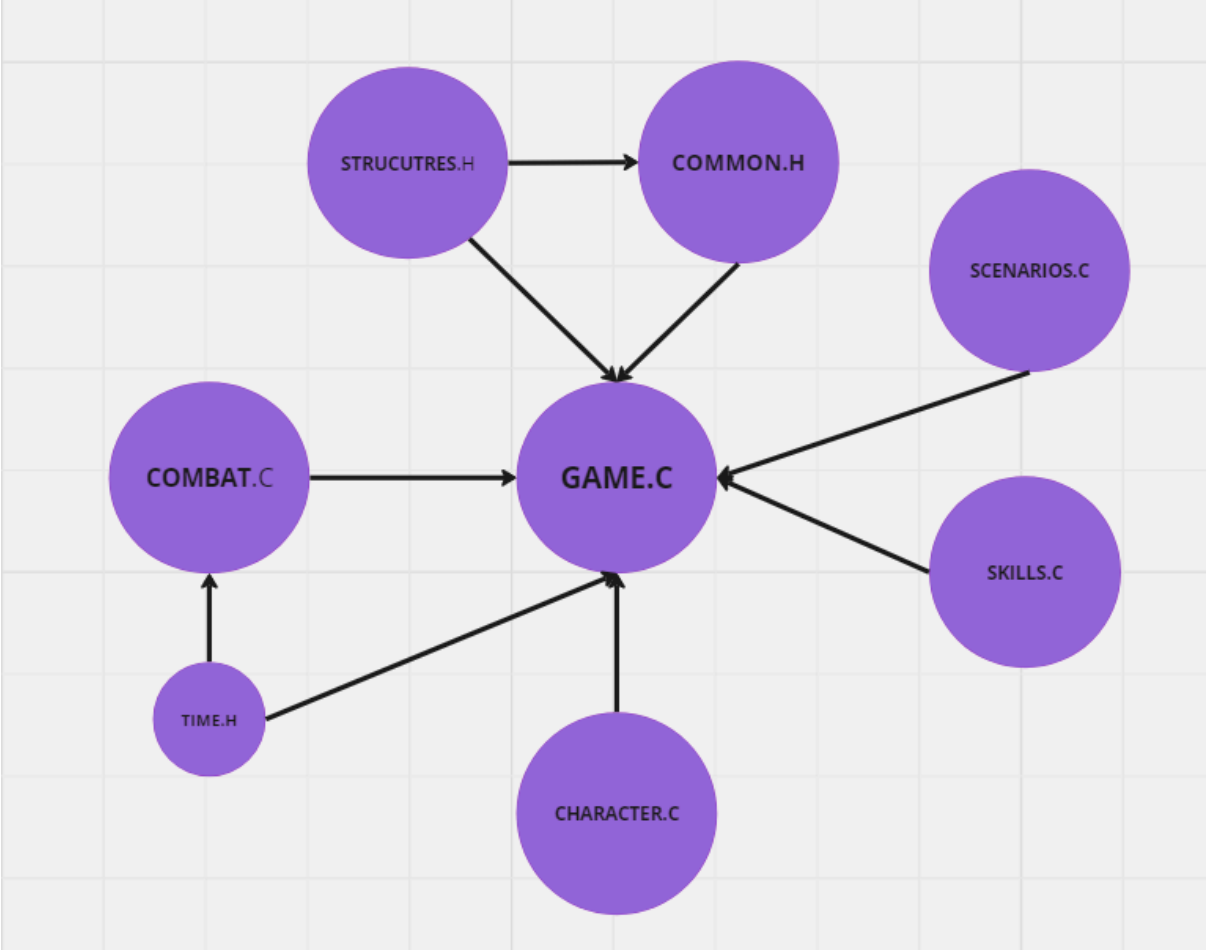
In terms of affection for the story from past decisions, we decided to make the scenarios as constants. The main advantage is that there is a unique story, and the user will see all the story before arriving at the final boss and determining if it wins or loses the whole game. The only variables that can be affected between scenarios are the skills. If the user has won all the past fights, it will have some advantage for future fights, as it has more skills farmed.

In conclusion, the only variability in the game between scenarios is the difficulty, but the story remains constant. All the scenario logic is in the main.c file, and took a few hours, bearing in mind that the scenario graphs were already done. Also, this number is not taking into account the variability in skills when finishing the game.

---

3. SOLUTION

3.1 System Architecture



Game.c is the main file. It gathers data and secondary functions from every other file.
Skills, scenarios and character provide the data and the necessary functions to initialize the information, each one giving the data that their name indicates.

Time and combat files are in charge of the fight_flow system, and are used when entering a fight. These two files will essentially return a true or false where they were called in the game.c file. Simplifying, they will determine the game.c if the fight was won or lost.

Structures provide the structures to all files, and common provides the macros, also to all the other files. Furthermore, it gets all the included libraries that are used, such as stdio, stdbool…

In general terms, action happens in game.c (the game_flow), and only goes to file combat for the fight_flow. The other files of course execute functions, but are more a complementary way to organize, and not as vital as the other two files.

3.2. Error Handling

The “Pompeu Heroes” game mixes together characters, skills, enemies, options and scenarios to make an interesting story which can be interacted with. This is done by establishing core constructions, including “Character”, “Skill”, “Enemy”, “Option”, “Decision” and “Scenario” in order to manage this diverse set of elements.

During the development of this project, the error handling was handled by a number of methods. First, there is the array bounds checking. Arrays such as “skills”, “scenarios”, “options”, and “available_scenarios” are created with fixed sizes having constants like “MAX_SKILLS” and “MAX_SCENARIOS”. This ensures that the indices of an array remain safe from overflowing beyond their capacities, as well as stops any segmentation faults caused by out of bound errors.

Moreover, string handling is managed carefully. In order to prevent common mistakes when handling strings, structures are supplied with strings by using string functions such as “strcpy”, and sufficient space is reserved for the destination arrays so that there are no buffer overflow errors. Particularly when dealing with “Skill” structures in the “Character” structure’s “skills” array, pointer management proves crucial. To steer clear of issues with incorrect or missing “Skill” object references, the code checks whether the pointers actually store references to valid “Skill” instances.

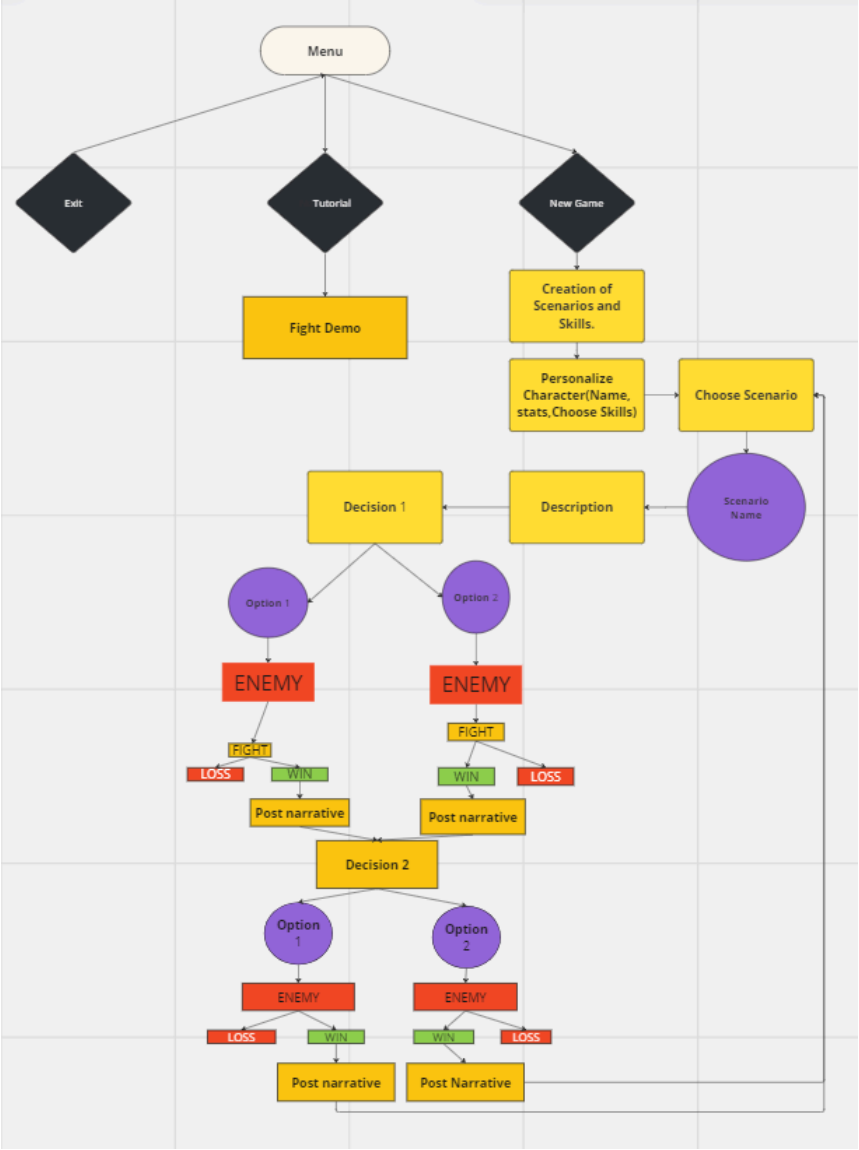
Further, to prevent errors, variables and structures are initialized to default values. Arrays and structures zero-initialize them, which puts them into a known state and lessens the risk of unexpected behavior due to uninitialized memory. This ensures that all elements of the game are predictable when starting. Additionally, logging and print statements are used all over the code to give real-time insight into the flow and the state of the program. That is helping in debugging and showing exactly where errors occur.

In general, these strategies of checking array bounds, handling strings carefully, handling pointers correctly, and initialization thoroughly provide a good backbone for the RPG game and reduce the prevalence of usual programming errors.

3.3 Data model design

This diagram below illustrates the gameplay structure of the “Pompeu Heroes” video game, detailing the menu options, decision points, and enemy encounters. The game begins with a menu offering three choices: Exit, Tutorial, or New Game. Selecting Exit terminates the game, while the Tutorial option provides a demonstration of fighting mechanics labeled "Fight Demo." Choosing a New Game leads to the creation of scenarios and skills, followed by personalizing the character's name, stats, and chosen skills. The player then selects a scenario to play, which is accompanied by a description and scenario name.

The gameplay involves a series of decision points and battles. The first decision presents two options, each leading to an enemy encounter. Each encounter can result in either a win or a loss, leading to respective post-narrative scenarios. After the initial encounters, a second decision point offers two more options, each leading to another fight with an enemy. The outcomes of these fights, whether wins or losses, lead to their respective post-narrative scenarios.



3.4 Dataset description and processing

For this project there is no large dataset in the traditional sense of dataset. Yet we have some different types of data to store and manage. In our game the data has been directly stored directly in the different .c files, or it is inputted by the player:.

1. scenarios.c has functions where a structure Skill pointer is imputed, and these functions fill all the information, taking into account the description of the scenario, decisions, options and enemies in each option. There is a structured Skill pointer array with all addresses of the scenario variables. This way, it is sent only the addresses to the functions that will use or print the scenario information.

2. skills.c, just as scenarios.c, has the information of each skill in a function, and when called, it fills the information into a variable through the address. This will be activated in two files: main.c when choosing the skills of the player, and combat when activating and giving skills to the enemies.

3. Dynamic data: The rest of the data is dynamic. It is, written by the same player, for instance, the character's name, or calculated and modified during the game, for instance, the hp statistic.

4. ETHICAL CONSIDERATIONS

We really hope that this section is read by the professors who decided to assign this project for the Data Structures and Algorithms 2 course so they can reflect on the average student who arrives in the first year most likely has not programmed before. This problem has been happening since the first term but became significant with the subjects EDA 1 and 2. You arrive at the theory class, and they teach you one thing, then you go to the practices or seminars, and they have nothing to do with it. Members of this group have even tried to reach out to the professors via email to explain that we are having issues with the coding that we can't solve, but then we couldn't get any meaningful help from them*. (*Except for Sortirios Papadiamntis, who at least did one meeting to try to help us. But it should not be expected that PhD candidates, as university PDI which are trying to advance their research, have the most important teaching load if their subject part is supposed to be just to guide us through lab exercises.)

All the knowledge we have put into this project has come from websites, AI assistants and even YouTube videos that are sources outside the university, and some questions to friends that work in the coding world.

It is really frustrating for us to see how we put in hours and hours and still cannot achieve what they ask of us because we have not been taught. And, honestly, we are really proud to say that all this code is 100% made by us (and only using LLMs for asking questions about syntax and errors, but not for generating code).

As a final note on this line, it seems bad organization from those who design the subject schedule to give us an important deadline–which is already pushing us to finish the project without having enough time to resolve many of the (commented) issues–the night before an exam from that same subject. As some of the group members are also PDI themselves, this is something to take into account; students' health and care should be a priority.

We must add that by no means is our aim to disrespect anyone. We want to highlight the challenges that we faced, and want to encourage the faculty to better consider how to approach these subjects next year, bearing in mind that most students are starting from scratch.

In adhering to ethical standards, we have maintained transparency in our use of LLMs and have properly acknowledged their role in the development process. This approach not only upholds academic integrity but also demonstrates the responsible and innovative use of advanced technologies in project development.

Overall, the creation of "Pompeu Heroes" has been a testament to our dedication to producing original work while embracing modern tools to enhance our capabilities. The collaboration between our team and the LLMs has resulted in a richer, more polished final product, showcasing both our technical skills and creative vision. This culture of "the more hardship the better" is something toxic from the past.

All the members of the group have been equally engaged in the development of the project and its final outputs. *Pau is not present in the video due to being ill that day and unable to participate, but this should not be understood as any lack of contribution on his part.

4. REFERENCES

Some reference we have used during the development of our game:

→DevChatter. (2018, 9 august). RPG Combat Programming Exercise (part 1) in .NET Core 2.1 using C# - Episode 67 [Vídeo]. YouTube.
https://www.youtube.com/watch?v=d-ym8G_InSA

→ *C and C on a rpg battle system - GameCreators Forum*. (s. f.).
https://forum.thegamecreators.com/thread/56691

→ Build software better, together. (s. f.). GitHub.
https://github.com/topics/rpg-game?l=c%2B%2B&o=asc&s=forks

→ C++ - Creating Basic Combat Function for Text RPG. (s. f.). Stack Overflow.
https://stackoverflow.com/questions/52417539/c-creating-basic-combat-function-for-text-rpg

→ Lecture Notes UPF EDA 2
https://docs.google.com/presentation/d/17io-jWx70E5xrgdJmSELv5nGYm_jQWlb/edit#slide=id.p1

We have used some usual coding sources as stackoverflow, github, and GeeksforGeeks Besides, we have used various AI assistants such as ChatGPT and Perplexity.AI to correct syntax errors in the code and orthographical errors in the report (not to generate the code).

Furthermore, we sought advice from professionals in this field, asking them how they would handle parts of the code that we did not have sufficient knowledge to do properly.

And, as commented before, we also obtained help from professor Sotirios Papadiamantis.