Object Oriented Programming Project Report - Game

Jack McGinn
10138837
(Dated: May 2020)

For this project, I have made a small game similar in style to a board game like Dungeons and Dragons. The code to complete this game makes use of key object oriented programming structures, such as inheritance and run-time polymorphism. It also makes use of more advanced C++ features like lambda functions, smart pointers,exception handling and move semantics. The user of the code takes control of two characters as they travel from room to room to the top of a tower, taking part in turn based combat encounters and puzzles along the way. All that is occurring to the user is displayed via descriptive text in the output window.

I. INTRODUCTION

The code for this project is saved under the one class per translational unit convention. A translational unit is the basic unit of compilation in C++, and it is constituted of a header (.h) and source file (.cpp). The advantage of organising like this is that translational units are isolated, so that when one translational unit is edited, only this unit needs to be rebuilt, reducing compile time. It is also the most commonly used convention, increasing the readability of the code to a general user.

This code contains 3 class structures. The first starting off with the purely virtual characters class. From this the monsters and players class is derived. Monsters is the base class for all the enemies the user meets during the game. From it the demon, spider, gargoyle and boss class is derived. The players class is the base class for the characters the user can control during the game, and from it the knight and mage class is derived. Figure 1 shows the monsters base class header. Both the players and monsters class require the other base class to be defined creating a circular dependency between the two. Just including one class in the other would create an array of exotic errors from multiple includes of the same header file that even guards can not protect against. This circular dependency is broke by forward declarations.

```
//forward declare dependencies
class players;
//include dependencies
#include "Characters.h"
//Abstract base class derived from characters for all the monsters you can meet in the game
class monsters :public characters
{
public:
    //Overwrite output operator to display status of monster when ouput
    friend std::ostream& operator<<(std::ostream& output, monsters& myobject);
    virtual ~monsters() {}
    virtual void fight(const std::vector<std::shared_ptr<players>>*) = 0;
};
//Overwriting output operator for monsters to display its status
inline std::ostream& operator<<(std::ostream& output, monsters& myobject)
{
    output << myobject.get_name() << " (HP: " << myobject.get_health() << ", effects: ";
    //For loop runs through all the active effects on the player so they can all be output
    for (size_t i{}; i < (myobject.show_effects()).size(); i++) {
        output << myobject.show_effects()[i].first << " ";
    }
    output << ")";
    return output;
}
```

FIG. 1: This shows the code from the monsters class header file. It includes a friend function wrote in-line (not in the source file) that overwrites the output operator.

One of the other classes is the inventory items class. This class is an interface for the items the user can use during the game. From it, the classes of health potion, stamina potion and clear effects potion are derived. There is also a class called inventory. This class contains a polymorphic vector of inventory items as well as member functions, coding how the user playing the game can use their inventory.

The final set of classes are all derived from the abstract base class rooms. All these derived classes are the different types of rooms the user goes through during the game.

II. CODE DESIGN AND IMPLEMENTATION

The file 'MAIN CODE' contains the main function for all the code. Figure 2 shows what this looks like. The "users" vector stores the user controlled characters for the game. They are constructed with the inventory they can use. Both user characters have access to the same inventory hence why the "bag" is passed to them as a pointer so both characters have access to and deplete the same inventory. The vector "users" contains smart pointers called shared pointers to the players objects inside it. A smart pointer is used as they try prevent memory leaks by making resource deallocation automatic. The shared pointer variant of a smart pointer was used because multiple objects need access to the users vector. This is because the user controlled characters go through multiple rooms in the game and each room is an object that needs access to the user controlled characters.

The polymorphic vector tower stores unique pointers for the different room objects in the game. The unique pointer is again another type of smart pointer. The scope of these pointers is just in the main function hence why unique pointers were used as opposed to shared. A for loop then iterates through all the objects in this tower vector. A conditional checks if the game is over. If it is not, then the member function complete room for each object is called. Due to run time polymorphism and all the derived classes overwriting this virtual function from the rooms class, this does a different thing depending on what type of room it is.

The "fight room" class is a turn based combat room, where a group of enemies take turns in fighting the char-

```cpp
int main()
{
    // Set precision of outputs
    std::cout.precision(3);
    //Introductory output message to user
    /* ... */
    //Define class to hold users inventory
    inventory bag;
    //Define a polymorphic vector that stores the characters the user controls, and initialise them wit
    std::vector<std::shared_ptr<players>> users;
    users.push_back(std::make_shared<knight>(knight{ &bag }));
    users.push_back(std::make_shared<mage>(mage{ &bag }));
    //Define a polymorphic vector that stores the rooms the users go through, where each is intialised
    std::vector<std::unique_ptr<rooms>> tower;
    tower.push_back(std::make_unique<fight_room>(fight_room{ &users }));
    tower.push_back(std::make_unique<get_inventory_room>(get_inventory_room{ &bag }));
    tower.push_back(std::make_unique<puzzle_room>(puzzle_room{ (rand() % 5) + 2 }));
    tower.push_back(std::make_unique<boss_fight_room>(boss_fight_room{ &users }));
    //For loop iterates through all the rooms in the tower then completes the actions the user needs to
    for (auto room_iterator = tower.begin(); room_iterator < tower.end(); ++room_iterator) {
        //Conditional checks whether the game is continuing and whether to complete the next room (if t
        if ((*room_iterator)->return_game_not_over()) {
            (*room_iterator)->complete_room();
            //Reset status of next room depending on whether the users characters died in the current r
        }
    }
```

FIG. 2: This shows the code for the main function.

acters the user controls. Figure 3 shows the header files for the "rooms" and derived "fight room" class. The rooms class makes use of a static data member called "game not over". Static data members are not associated with any object,and there is only one instance of them in the whole program with static storage duration. This means that this variable can be accessed and changed by all room objects. So when the users are defeated, the game can be set to be over for all rooms. This is the variable that the conditional in the main function checks the value of to know whether to keep running the game or not. The "output message" function is used to output the status of all the characters currently alive in a visually digestible manner. It outputs the status of the characters through the overloaded output operators that are the friend functions shown in figure 1.

```cpp
class rooms {
protected:
    static bool game_not_over; //Static data member that all rooms can access so that they can all set when the game is over
public:
    virtual ~rooms() {}
    virtual void complete_room() = 0;
    virtual bool return_game_not_over() = 0;
};
class fight_room :public rooms
{
protected:
    std::vector<std::shared_ptr<players>> players_list(); //Polymorphic array of users characters
    std::vector<std::shared_ptr<monsters>> enemy_list(); //Polymorphic array of enemies the user faces
public:
    fight_room();
    fight_room(std::vector<std::shared_ptr<players>>* player);
    ~fight_room();
    //Member function to change the status of the rooms to be game over when the user characters die
    bool return_game_not_over();
    //Member function to complete the users turn in combat
    void users_turn();
    //Member function to complete the enemies turn in combat
    void enemies_turn();
    //Member function to display output message of what the current status and situaiotn of the combat encounter looks like
    void output_message();
    //Member function to complete all the actions required in this room for the user to progress
    void complete_room();
};
```

FIG. 3: This shows the code for the room class header and derived class fight room header.

Figure 4 partially shows what the "users turn" member function looks like. The for each algorithm in the user turn function iterates through each character in the players list vector and then for each one completes the functor in the brackets. A functor is a class that acts like a function. It can be very cumbersome to right a whole separate functor for the loop. Lambda functions allow you to write in-line anonymous functors. The square bracket is what is passed to the lambda function. A special pointer called this is passed to it, meaning the lambda function has pointer access to all the member data of this fight room object.

```cpp
std::for_each(players_list.begin(), players_list.end(), [this](std::shared_ptr<players> player)
{
    //Conditional to check all the enemies have not already been defeated so that the user can take their turn
    if (enemy_list.size() > 0) {
        std::cout << player->get_name() << " TURN: ";
        player->effects();
        //Conditional to check the user character is not efeated so that they can take their turn
        if (player->get_health() <= 0) {
            std::cout << player->get_name() << " is defeated" << std::endl << std::endl;
        }else {
            output_message();
            player->fight_choice(&enemy_list, &players_list);
            //Lambda function to remove any enemies from the room if they have been defeated (less then 0 health)
            enemy_list.erase(std::remove_if(enemy_list.begin(), enemy_list.end(), [](auto const& it) {return it->get_health() <= 0; }))
            //Conditional to check if all the enemies in the room have been defeated so the heroes won
            if (enemy_list.size() == 0) {
                std::cout << "Enemies defeated" << std::endl << std::endl;
            }
        }
    }
});
```

FIG. 4: This shows most of the code for the users turn function which is a member function of the fight room class.

The functor completes the players turn by calling its member functions "effects" and "fight turn". Fight turn takes by reference all the characters in the combat room as its input. This is because for a players turn in combat they can effect any character in the room during combat. Now the users turn is completed, any characters that are defeated are removed from their respective vectors so they no longer appear in the room. Lambda functions inside the standard library function "remove if" remove the characters that are defeated, where the functor of the lambda function checks whether the character has 0 or less health so should be removed. If the players list becomes empty so all the user characters have been defeated, the static member data is set to be false and the game is over and the code is returned to main. Returning to main as opposed to exiting the code is a much better way to end the program as returning to main makes sure any destructor for locally scoped objects are called. The enemy turn member function of the fight room class works in a very similar way to the users turn member function.

Before reference was made to specific player member functions. Figure 5 shows the header file for the mage class. As was implied in the main function, the players have a parametrised constructor that a pointer to an inventory object is passed to. This pointer is originally set to be nullptr so that when the default constructor is called the memory is deallocated. The "fight turn" member function in figure 5 completes a characters turn during combat. It contains a while loop to keep taking user inputs for their turn until they give a valid move. All the available moves the user has is displayed to them through the choice output member function. The user is then asked to input a number to select which move they would like to do. A function called "integer checker" is called which keeps taking a user input till they give an integer that is within a certain range. Conditionals then check and complete the chosen user move. The effects member function is relatively straight forward and uses for loops and conditionals to complete any active effects on the user. Once an effect has 0 turns left a lambda function in the remove if algorithm removes the effect from the effect list so that it is no longer active on the user.

The class "boss fight room" is a derived class of fight room and uses compile time polymorphism. It overrides

```
class mage :public players
{
protected:
    //Key member data for the player
    double health{ 200 };
    double attack{ 40 };
    double stamina{ 300 };
    std::string name{ "MAGE" };
    double resist{ 1 }; //How much a character resits damage
    bool not_stunned{ true };
    //The string is the name of the effect and the int is how many turns it effects the user in combat for
    std::vector<std::pair<std::string, int>> effects_list{};
    //Bag is a pointer to the users currenty inventory. It's a pointer so that all the players the users control have access to the same inventory
    inventory* bag{ nullptr };
public:
    mage();
    mage(inventory*);
    ~mage();
    //Member function outputs all the possible moves for this character and their required input during its turn in combat
    void choice_output();
    //Member function that completes the turn of this character based on user inputs, during its turn of combat
    void fight_choice(const std::vector<std::shared_ptr<monsters>>&, const std::vector<std::shared_ptr<players>>&);
    //Member function to handle all the current active effects on the user and complete what they do
    void  effects();
```

FIG. 5: This shows part of the header file for the mage class, not all its member functions have been shown. The knight class works in the exacts same way just with different values for its member data and moves in the fight turn member function.

the complete room function to add just a boss to the enemy list, and overrides how the enemy turn member function works. Figure 6 shows part of this function. It works in the exact same way as it did in the fight room class, except it has different actions depending on the health of the current enemy it has iterated to. Conditionals check whether their health is below certain thresholds and if it is it appends a demon or boss to a temporary vector. The boss class has a parametrised constructor to create an object with specific health and effects. The boss that is added to the temporary vector is constructed with the exact same health and effects as the one that is currently iterated to, to look like it has duplicated itself. Once the for each loop is over the temporary vector is appended to the enemy list vector and then cleared for the next enemies turn. The temporary vector is needed because you can not just append to a vector whilst you are iterating through it.

```
std::for_each(enemy_list.begin(), enemy_list.end(), [&count, this, &temp_enemy_list](std::shared_ptr<monsters> enemy)
{
    //Conditional to check all the players have not already been defeated so that the user can take their turn
    if (players_list.size()) {
        std::cout << enemy->get_name() << " " << count << ": " << std::endl << std::endl;
        count++;
        enemy->effects();
        //Conditional to check the enemy is not defeated so that they can take their turn
        if (enemy->get_health() <= 0){ ... }
        else if (enemy->get_health() <= summon_at_health && enemy_list.size() < 3 && enemy->get_name().compare("BOSS ENEMY") == 0) {
            //Add them to temp enemy list to after the enemies turn be added to the enemy_list and then clearing the temp_enemy_list for next turn
            temp_enemy_list.push_back(std::make_shared<demon>(demon{}));
            summon_at_health = 0.25 * summon_at_health;
            std::cout << "Summons a demon to assist" << std::endl << std::endl;

        }else if (enemy->get_health() <= double_at_health && enemy_list.size() < 3 && enemy->get_name().compare("BOSS ENEMY") == 0) {
            //Use parameterised constructor to create new boss so it has the same values as the current boss who's turn it is so that appears to be du
            temp_enemy_list.push_back(std::make_shared<boss>(boss{ enemy->get_health(),enemy->show_effects() }));
            double_at_health = 0.5 * double_at_health;
            std::cout << "Duplicates itself" << std::endl << std::endl;
        }else {
            enemy->fight(&players_list);
            //Lambda function to remove any players from the room if they have been defeated (less then 0 health)
            players_list.erase(std::remove_if(players_list.begin(), players_list.end(), [](auto const& it) {return it->get_health() <= 0; }), players_
            //Conditional to check if all the user characters in the room have been defeated so the enemies won
            if (players_list.size() == 0){ ... }
```

FIG. 6: This shows the for each loop and conditionals the code uses complete all the enemy turns in combat in the boss enemy room object.

As can be seen in figure 1, the monsters class has a member function called fight to complete its turn in combat. The classes derived from monster work in much the same way as the classes derived from player. The major differences are that where the players take user inputs to determine their move in combat, the enemies randomly generate numbers to decide which user character they attack and with what.

The code also has a class called "get inventory room". In this room the user has to decide which inventory from a choice of two they would like to take with them for the rest of the game. The users inventory is then move assigned to equal the one they chose. This room is constructed with access via a pointer to the inventory it is changing. It is constructed by pointer to the same one the user characters have access to meaning that it changes the users inventory for all later rooms too.

Figure 7 shows a header file for the inventory class. The sort inventory function contains the remove if lambda function combination previously discussed in this report to remove potions from the contents vector when there is 0 of that type left. Each potion has a member data saying how many of them there are in a particular inventory. So the contents vector in the inventory class does not store for example 3 health potion objects, but one health potion object that has a member data of the value 3. This number can then be changed as the potions are used. The move assignment operator works by copying the data of one inventory class to another. Move semantics use the idea of lvalues and rvalues. An lvalue is something in the code where we can take the address and exists in some permanent memory. An rvalue refers to a temporary object that needs to be captured in an lvalue to remain in the permanent memory. The move assigment operator turns an lvalue in to something like an rvalue so that its data can be moved and the objects previous contents destroyed. The data is moved with swaps functions to equal the inventory object it is copying.

```
class inventory
{
private:
    size_t size{};
    std::vector<inventory_items*> contents{};
public:
    inventory();
    inventory(std::vector<inventory_items*>);
    ~inventory();
    //Member function to remove items from the inventory when the number of items of its type become 0
    void sort_inventory();
    //Member function to output all the items in the inventory in a clear way to the user
    void output_inventory();
    //Member function dealing with how the users use an item from their inventory during combat. The p
    bool use_inventory(players*);
    size_t items_in_inventory();
    inventory(inventory&);
    //Move assignment operator so the code knows how to copy inventorys over, used in one of the rooms
    inventory& operator=(inventory&&) noexcept;
};
```

FIG. 7: This shows the header file code for the inventory class.

The final room class in the game is the "puzzle room". In this room, the user has to figure out the next number in a series of integers, and the series size is randomly chosen. The sequence of integers is stored in a dynamic array whose size is set by an input in to the parametrised constructor. The memory is allocated in a try function shown in figure 8. The code tries to set the array to be of this input size and if this allocation fails, an exception is thrown and it immediately goes to the catch of this error. The try construct is exited and everything in there is reset. In the catch, the game is set to be over by the static room variable and the code returns to main. If the no exceptions are thrown, the rest of the room code can precede as normal. The complete room function outputs this sequence and a conditional checks the user input for the answer.

III. RESULTS

The game appears to the user as text on the output window, where they input numbers to make decisions.

```
puzzle_room::puzzle_room(int amount) {
    //Try allocate memory
    try
    {
        puzzle = new int[amount];
        size = amount;
    }
    catch (std::bad_alloc memFail)
    {
        //If memory allocation failed set the game to be over and return to main so the program ends
        std::cerr << "Memory allocation failure" << std::endl;
        game_not_over = false;
        return;
    }
}
```

FIG. 8: This shows the parametrised constructor for the puzzle room class.

Figure 9 demonstrates what a typical output of the game looks like during the fight room.

```
MAGE TURN:

1. DEMON (HP: 135, effects: )      2. GIANT SPIDER (HP: 85, effects: )      3. GARGOYLE (HP: 160, effects: )



1. KNIGHT (HP: 375 ST:100, effects: )       2. MAGE (HP: 200 ST:300, effects: )


 Press the following numbers to:

   (1.) Fireball                        (2.) Heal
    40 fire damage                       Heal 100 HP
  Costs 30 stamina                     Costs 70 stamina

   (3.) Lighting strike                 (4.) Magic shield
      60 shock damage                Block half of incoming damage
      Cost 60 stamina                    Gain 50 stamina

                    5. Open Inventory

Input: 3

Choose which enemy to attack from their number, Input: 2
Does 60 damage to GIANT SPIDER


DEMON 1:
Does basic attack: Does 40 damage to KNIGHT

GIANT SPIDER 2:
Shock effects makes GIANT SPIDER weak so does less damage on next attack

Does basic attack: Does 35 damage to MAGE

GARGOYLE 3:
Does basic attack: Does 35 damage to KNIGHT

Press Enter to Continue
```

FIG. 9: This shows what a typical output with user inputs looks like in the game during a combat encounter.

III.1. Discussion and Conclusion

To conclude, object oriented techniques in C++ were successfully implemented to create a board game (where the board would be the tower with the rooms and the counters are the user controlled characters). However, this code can be easily expanded. Creating a wider variety of types of players and monsters classes as well as a wider variety of rooms class with different and more interesting puzzles would make for a more entertaining game. The output of the game would also be greatly improved if it included some graphics which would require certain libraries to be downloaded. However, all these things can be more simply added on and the base structures for a game in C++ that uses object oriented techniques has been made.

Word count - 2491 words.