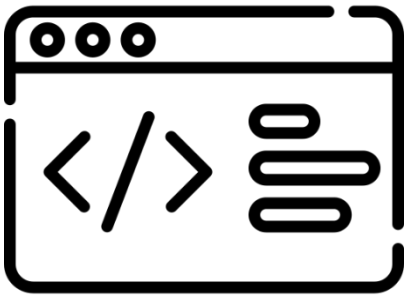
University of Nottingham
UK | CHINA | MALAYSIA

Faculty of Science and Engineering

Department of Electrical and Electronic Engineering

Report 2 – Adventure Game



ACADEMIC YEAR 2023/2024

Course Code: EEEE1032/1042 UNMC

Student's Name : MARCUS YEW MIN JIE

Student ID : 20612926

Course : Please tick (√)

Electrical and Electronic Engineering (√)

Mechatronic Engineering ( )

TABLE OF CONTENTS

ABSTRACT

This report provides a documentation of how the adventure game program was made possible by using C language. The game utilizes descriptions of areas that the player is in, and is can be played on virtually any computer since the game's main platform is the command prompt. The program has 13 accessible rooms, where some can be freely accessed throughout the entire run of the game, while others can only be entered when certain conditions are met. Monsters will be present in specific rooms as well. Some monster battles are not mandatory to defeat in order to progress the story, while some are. Besides that, there are also artifacts and items in the game that will give the player the advantage in battle against certain monsters. A score and message will be generated at the end of the game to notify the player whether he or she won the game. The functionality and logic behind each block of code will be discussed in further parts of the code to give clarity as to how they work and the method used to achieve the outcome desired.

INTRODUCTION

Games have evolved a long way since they were first released. In this current era, games have been separated into many subdivisions such as first-person shooters, sandbox, role playing game (RPG) and many more. The adventure game program made falls under the and RPG and sandbox category. RPG is a genre where the player takes on the role of a fictional character, and complete the quests assigned by the game by exploring the world around them. It is worth noting however, that the adventure game program utilizes the command prompt as a platform for the program, which is severely outdated and thus the functionalities of the program may be limited and considered outdated compared to other games in this day and age. They say that a picture speaks a thousand words, however due to the limited capabilities of the command prompt and C Language not typically being a language for game development, a good description of the world around the player has to be given to paint a good image for the realm of imagination to work its job.

This report is structured in a way that each section can be accessed from the table of contents, and the reader can jump back to the table of contents from any header title. This is so that the reader is able to conveniently access necessary parts of the reports either by category or specific functions in the report.

GENERAL DESIGN CHOICES

A project format was used for this assignment which consist of 3 main files, a main file(main.c), a function declaration file(CW2.c) and a function definition file(lab2.h). This provides better visual clarity to readers as each file serve their own purpose and provide valuable insight on parts of the code. In addition, a struct is used to package variables that are commonly used in functions as a substitute to global variables as it provides better modularity to each part of the code, encapsulates data and enhances code readability. Structs also offer better code maintenance as a struct is more localized than changing global variables scattered all over said code. With that said, the declaration of the members of the struct will be done in the lab2.h file while the initialization of the members will be main.c file.

1.0 THE HEADER FILE

The header file serves as a file to input function definitions of made to allow the program to execute commands faster and more importantly to declare members of the struct used in this program which is called adventure game. Members declared can be of all data types. The data types used for variables in this program consist of integers, constant strings, strings, array of characters and integer arrays. Members of the struct can be called from anywhere in the project, allowing for easy alterations to the values of the member of the struct. Comments were also added to the code that concisely explains what each member of the struct is tasked with, and what each function do.

2.0 THE MAIN FILE

The main file is responsible for initializing the member of the structs with values, all the code were commented neatly to provide visual clarity as to what each of them mean and also separated into sections. Besides hosting the members of the struct, it is also used as a core component to obtain user input for commands, this is achieved by using a `while` loop with the `fgets` function. `Fgets` reads a line of character from a current stream up until the end of string character is encountered or the size of the string is reached, whichever comes first. By declaring the stream as stdin, it means that the data read is from the user input, similar to the scanf function. After the user input, the string is stored in the command struct, which is then read by the fgets function. This while loop will be continuously active, unless the specified

function returns 0, which will be discussed in the next section of the report. The code block below shows the code for obtaining the commands through user input.

```
while (fgets(ag.command,1000,stdin)){
      printf("+-------------------------------------------------------------+\n");
      repeatCommand(&ag);
      if (processCommand(&ag)==0) break;
      printf("+-------------------------------------------------------------+\n");
```

Code Block 1: Loop for obtaining Commands

3.0 THE CW2.C FILE

This file is home to all the functions that are in the program. All the functions in this file are coded where modularity rules are followed and ensuring a structured and organized design. Each function within this file serves a distinct purpose, promoting code readability, ease of maintenance, and the reusability of components. This section of the report will discuss the working principles of each code and their purpose in the software.

3.1 MOVEMENT FUNCTION

This section will explain the movement functions in the program. There are also some members of the adventure game struct that are related to movement in the game and the rooms of the game which will be explained here. ag.totalRoom is an constant integer that represents the total amount of rooms in the game. ag.rooms is an array that contains the name of all the rooms in the game. ag.moveToArray is a 2D array that contains the value of the room that the player will be in if the player decides to go in that direction. -1 means that the player is not allowed to travel in said direction, -2 means that that area is locked and any value greater 0 means that player is allowed to go in that direction. It essentially updates the ag.room value in the adventure game struct. ag.roomDesc is an array that contains the description of all the rooms in the game.

3.1.1 PROCESS MOVEMENT FUNCTION

To start, the movement commands of the adventure game consists of north, south, east and west or n, s, e, w. If the command typed matches the commands that were predetermined, the function would run for the individual if statements that match the criteria. This can be achieved through several if statements using the strcmp function, which accepts two strings as inputs

and returns 0 if they match. If they do, it will then check if the direction the player is moving to is valid. It is worth noting that the index of the ag.moveToArray member is accessed by declaring the index as [ag->room][direction] as the movetoArray is a 13x4 array, where by ag->room can be incremented up to a value of 13. Therefore, ag.moveToArray has 13 rooms worth of movement checking, and checks all 4 direction for each room. The direction index is replaced by values 0 to 3 where the sequence of direction is north to south to east to west. A message that prints the location of the room and the description is executed if the player successfully moves to a valid specified room and will print an error message if the player is not allowed to move there. Similar logic are applied for the other 3 directions by changing the second index of the ag.moveToArray to 1,2 and 3.

For locked doors, it was previously mentioned that the value in the ag.moveToArray would be -2. This situation can be handled by implementing an additional if statement that checks if the value encountered is -2. The only rooms that are locked are room 10 and room 13 where the player would have to south and east respectively. Therefore, this if statement will only be necessary to be implemented in these two movement sections. Code block 1.2 shows the movement logic involving locked doors and regular rooms. Just two of the movement functions will be shown in this section as the other two use similar logic.

```
if(strcmp(ag->command,"north")==0||strcmp(ag->command,"n")==0){ //accepts north or n as input
        if (ag->moveToArray[ag->room][0]!=-1) { //checks if room exists
            ag->room=ag->moveToArray[ag->room][0]; //updates room
            printf("You are now in %s\n",ag->rooms[ag->room]);
            printf("%s",ag->roomDesc[ag->room]);

        }
        else printf("You can't go north from here!\n"); //error message
        return(1);
    }
    if(strcmp(ag->command,"south")==0||strcmp(ag->command,"s")==0){  //accepts south or s as
        if (ag->moveToArray[ag->room][1]>-1) {
            ag->room=ag->moveToArray[ag->room][1];
            printf("You are now in %s\n",ag->rooms[ag->room]);
            printf("%s",ag->roomDesc[ag->room]);
        }
        else if (ag->moveToArray[ag->room][1]==-2){ //checks if room is locked
            printf("This room is locked...\n");
            printf("It seems like you need a Bronze Key to open it.\n");
        }
        else printf("You can't go south from here!\n");
        return(1);
        }
```

Code Block 1.2: Movement Logic

Additionally, Figure 1.1 to 1.7 shows several outputs that are generated when the player move to valid rooms and non-valid rooms.

```
n
You are now in The Living Room
You see dusty furnitures draped in tattered sheets, the once vibrant living space which has surrendered to the passage o
f time.
north
You are now in The Graveyard
A haunting silence lingers, the air heavy with the weight of history.
```

Figure 1.1: Travelling North to an Unlocked Room

```
s
You are now in The Living Room
You see dusty furnitures draped in tattered sheets, the once vibrant living space which has surrendered to the passage o
f time.
south
You are now in Secret Armoury
A secret room that hosts many dangerous weapons, you feel a force of nature locked behind the doors in front.
```

Figure 1.2: Travelling South to an Unlocked Room

```
e
You are now in The Medical Bay
You feel a sense of serenity washing over you, promising relief and recovery.
east
You are now in The Forbidden Prison
Stone walls echo with distant whispers, and the air is heavy with an ominous stillness.
```

Figure 1.3: Travelling East to an Unlocked Room

```
w
You are now in The Medical Bay
You feel a sense of serenity washing over you, promising relief and recovery.
west
You are now in The Graveyard
A haunting silence lingers, the air heavy with the weight of history.
```

Figure 1.4: Travelling West to an Unlocked Room

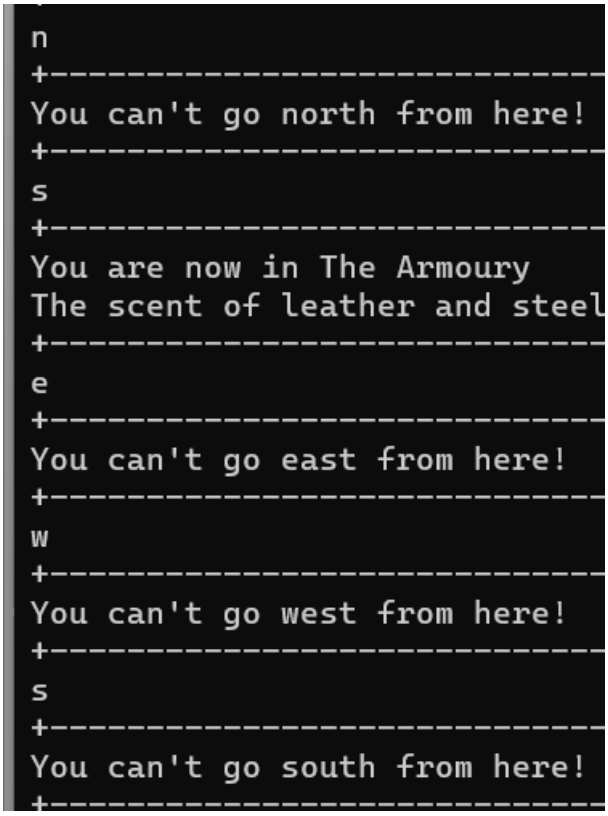


Figure 1.5: Travelling to an invalid Room

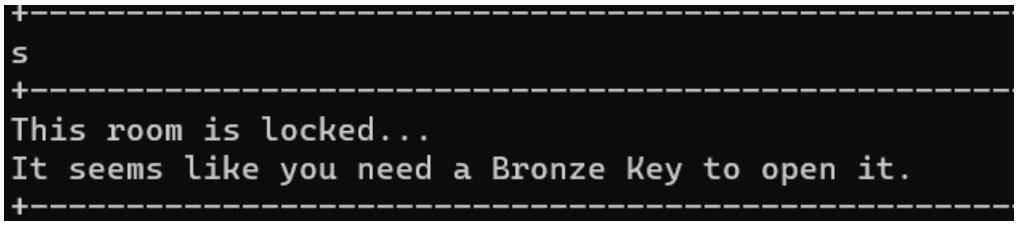


Figure 1.6: Travelling to a Locked Room 10

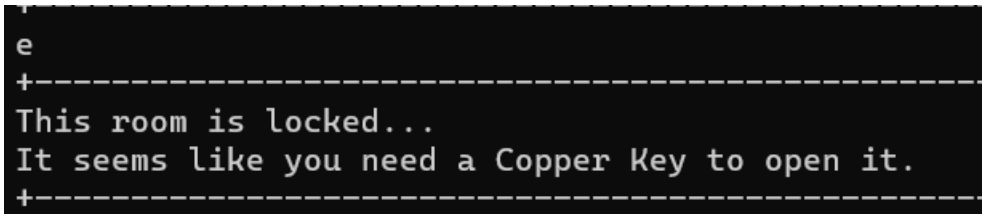


Figure 1.7: Travelling to a Locked Room 13

3.1.2 UNLOCK DOOR FUNCTION

To unlock the door for room 10, a bronze key would be required while room 13 requires a copper key. The function checks if the player is in the room connected to the locked door and if they have the required key by checking if the key is picked up by the player. If these conditions are met, the value in the ag.moveToArray is updated from -2 to a valid room number to allow the player to move to said unlocked room. The key is also removed from the players inventory and from the game upon use. If the player tries to unlock a door in a room where there are no locked doors present, an error message will be printed by using an else statement.

Additionally, the monster in room 5 has to be slain before the player is allowed to unlock the door even if they have the specified key. This situation can be handled by simply implementing an if statement to check if the monster is alive in room 5. If it is, the program will exit the function through the return function and the door will not be unlocked. The code snippet for unlocking the door is shown in Code Block 1.3. Figure 1.8 to 2.2 shows all outcomes of this function.

```
/*UNLOCK DOOR*/
void unlockDoor(adventureGame *ag){
    if (((ag->room)==4) && (ag->itemLoc[7]==-2)) {
        if(ag->monsterStatusArray[4]>0){
            printf("The Werewolf blocks the door!\n");
            return;
        }
        ag->moveToArray[ag->room][1]=9;
        printf("You used the Bronze Key.\n");
        printf("Ka-chink!\n");
        --ag->inventory;
        ag->itemLoc[7]=-1;
        return;

    }
    if (((ag->room)==11) && (ag->itemLoc[6]==-2)) {
        ag->moveToArray[ag->room][2]=12;
        printf("You insert the Copper Key.\n");
        printf("*click*\n");
        --ag->inventory;
        ag->itemLoc[6]=-1;
    }
    else printf("There are no doors to unlock here...\n");
}
```

Code Block 1.3: Unlock Door Function

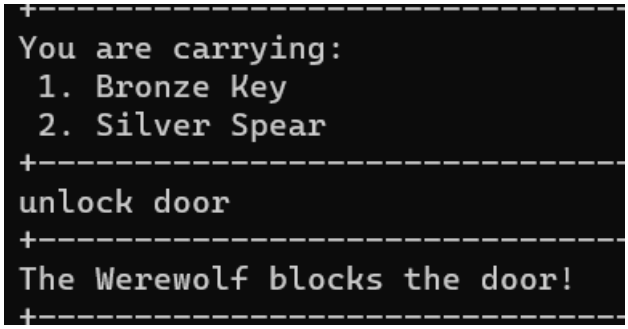


Figure 1.8: Monster Exists When Unlocking Door

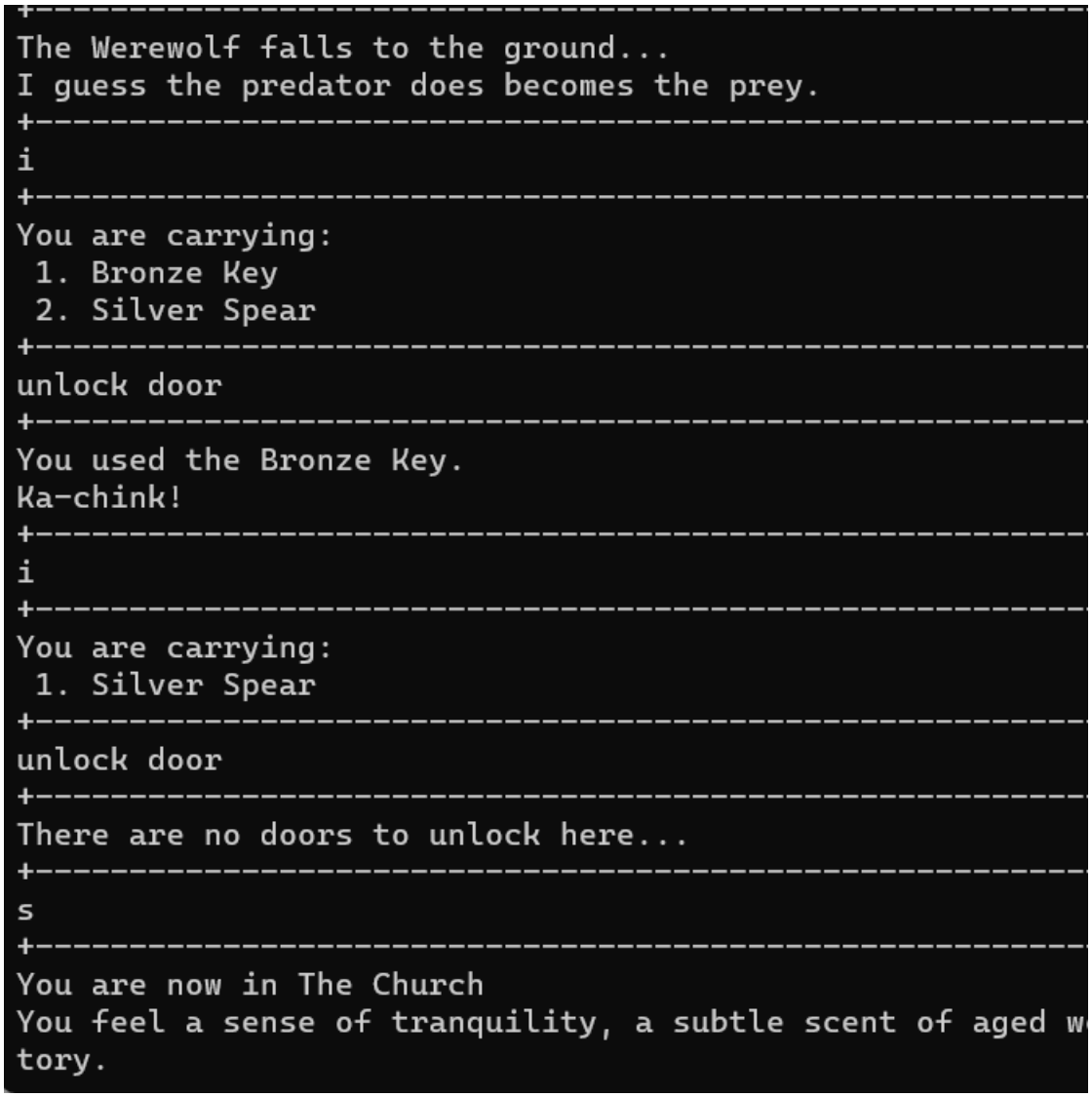


Figure 1.9: Monster Dies and Unlocking Door and Error Message

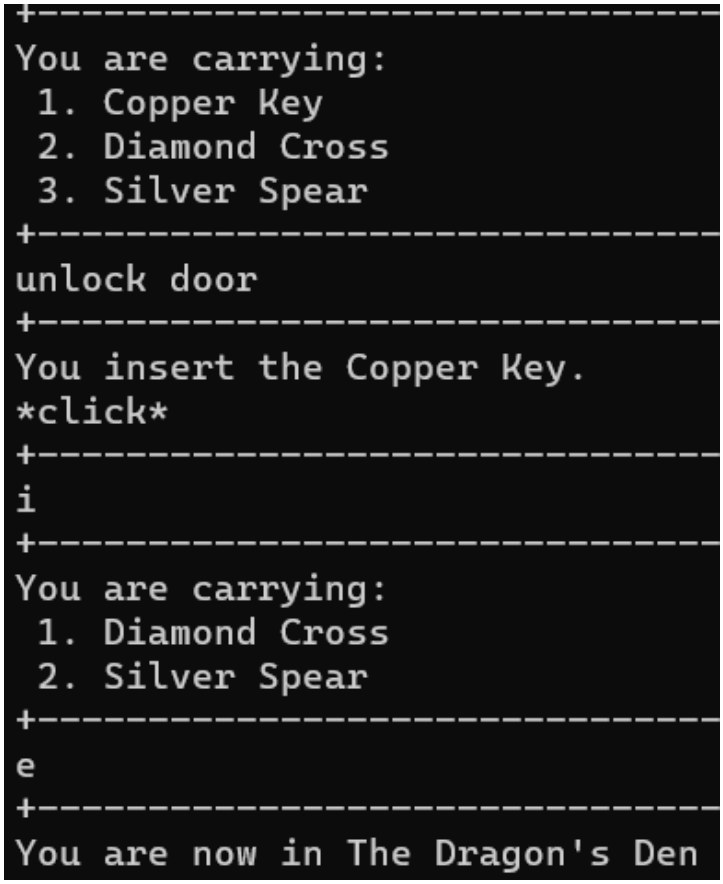


Figure 2.0: Unlocking the Door

3.2 ITEM FUNCTIONS

The members of the struct related to functions regarding items are as follows. ag.inventory is an integer that is updated based on the number of items the player is carrying. ag.itemLoc is a integer array of 13 elements that has values representing the location of all the items in the game which are assigned to the room number of each item that resides there. A value of -1 represents that the room does not have an item or the item is removed from the game, a value of -2 represents the item is in the players inventory and a value greater than 0 represents the item is in the room it is placed at. ag.itemArray is an array of constant characters with 13 elements that has the name of all the items in the game. ag. itemDesc is also an array of constant characters with 13 elements that stores the description for the items in the game when seen.

3.2.1 PROCESS GET FUNCTION

To get an item in the game, the player would have to type “get (item name)” as the command. With this information, the function first compares the item name typed, with the item name in the itemArray struct member. This can be achieved by declaring a variable(itemName in this case) as a pointer to a character which is the fifth letter in the “get (item name)” command, which is index 4 of the command. A for loop iterates through all elements in the item array as the strcasecmp function will compare the item name typed with the item array. If the item matches with an item in the array, it will move on to the next stage of checking, which is whether the item location matches the room the player is in. If it does not match with an item, it will print an error message accordingly.

If the room and the item location match, the player would be able to pick up the item where the inventory value is updated and the item location is changed to -2 to signify that it is in the players possession. The function is then returned and does not continue down the code. If this function returns false, an error message will be printed to notify the player that they have not picked up the item.

The conditions above have also been encapsulated inside an if statement that checks if the inventory of the player exceeds the maximum value, which is 3 items. If it does, the player is not allowed to pick up any additional items and an error message is printed.

Additionally, if monsters are present in certain rooms, they will stop the player from picking up the items. To achieve this outcome, an if statement is implemented that checks the location

of the player and the status of the monster in said room. If these conditions are met, a message that urges the player to defeat the monster is printed and the function is returned, effectively breaking the if statement. Code block 1.4 shows the code snippet of the process get function. Figure 2.1 to 2.5 shows all outcomes of this function.

```c
/*ITEM FUNCTION*/
void processGet(adventureGame *ag) {
    char *itemName = &ag->command[4];
    for (int i = 0; i < ag->totalRoom; i++) {
        if (ag->inventory<3){
        if (strcasecmp(itemName, ag->itemArray[i]) == 0) {
            if((ag->room==6 || ag->room == 9 || ag->room == 12)&& ag->monsterStatusArray[ag->room]>0){
                printf("It look's like it won't give the item up without a fight!\n");
                return;
            }
            if (ag->itemLoc[i]== ag->room) {
                printf("You picked up the %s!\n", ag->itemArray[i]);
                ag->itemLoc[i] = -2;
                ag->inventory++;
                return;
            }
            else {
            printf("Maybe you should continue to look around...\n");
            return;
            }
        }
        }
        else{
            printf("Your inventory is full!\n");
            return;
        }
    }
    printf("You can't find that item...\n");
    return;
}
```

Code Block 1.4: Item Get Function

```
+------------------------
get food
+------------------------
You picked up the Food!
+------------------------
```

Figure 2.1: Picking Item in the Correct Room and Name

```
+---------------------------
get itemthatdoesntexist
+---------------------------
You can't find that item...
+---------------------------
```

Figure 2.2: Picking Item with the Wrong Name

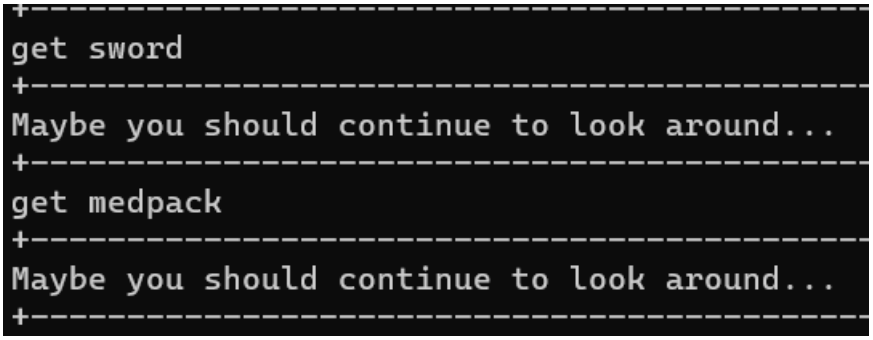


Figure 2.3: Picking Item with the Correct Name but Incorrect Room

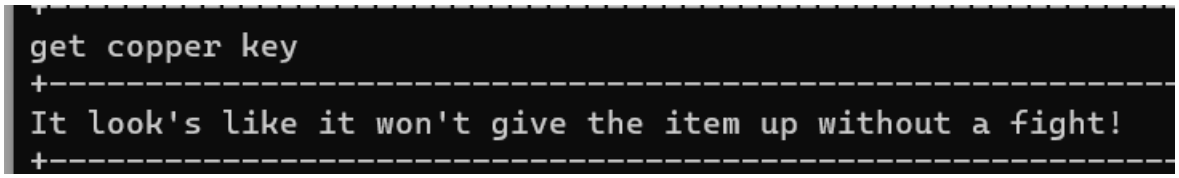


Figure 2.4: Picking Item with Monster Present

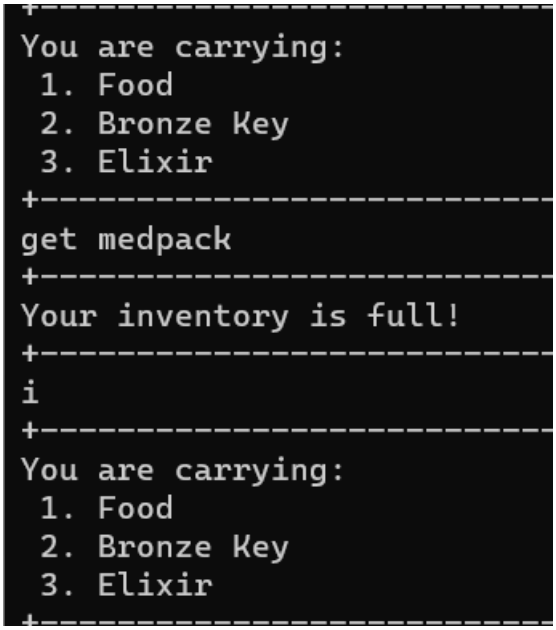


Figure 2.5: Picking Item when Inventory is Full

3.2.2 PROCESS DROP FUNCTION

This function deals with the player dropping items and removing them from the inventory. It could be said that the logic applied in this function is the inverse of what was done in the process get function. Similarly, the command the player should type when needed to drop an item will be “drop (item name)”. A pointer to a character called itemName is initialized as the fifth index of the command typed if drop is recognized. The function would then check if the player’s inventory is empty and if it is, an message stating the player has nothing to drop is printed and the function is returned early.

If the player does indeed have items in his or her inventory, a for loop is implemented to check if the item typed is picked up by the player. In other words, the function checks if the value of the item location is -2. It then enters another nested if statement to confirm that the item name typed matches with the item name in the item array. If both of these conditions return true, a message stating that the item has been dropped will be printed, and the location of the item dropped will be updated to the value of the room that the player is currently residing in. The value of inventory which represents the total number of items in the player’s possession is also reduced by 1 and the function is returned.

Additionally, if none of the conditions above are met, it means that the player has typed an invalid item name that does not exist or either typed wrongly. An error message will be printed stating that the item is not in the player’s inventory is printed to prevent the player from dropping anything, and the function is returned. The Code Block 1.5 shows the code explained in this section.

```c
/*ITEM FUNCTION*/
void processDrop(adventureGame *ag) {
    char *itemName = &ag->command[5];

    if (ag->inventory == 0) {
        printf("You have nothing in your inventory...\n");
        return;
    }

    for (int i = 0; i < ag->totalRoom; i++) {
        if (ag->itemLoc[i] == -2) {
            if (strcasecmp(itemName, ag->itemArray[i]) == 0) {
                printf("You dropped the %s!\n", ag->itemArray[i]);
                ag->itemLoc[i] = ag->room;
                --ag->inventory;
                return;
            }
        }
    }
    printf("You don't have %s in your inventory!\n",itemName);
    return;
```

Code Block 1.5: Drop Item Function

Figure 2.6 to 2.8 shows the possible outcomes for situations covered above.

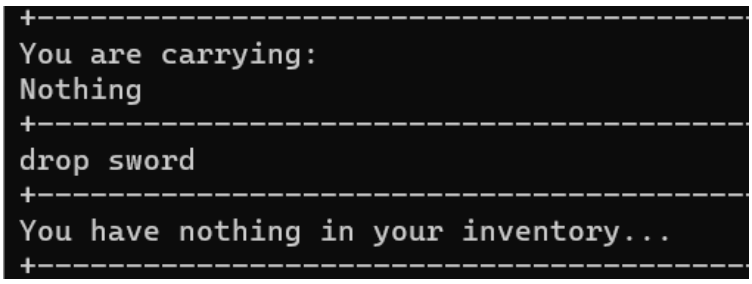


Figure 2.6: Dropping Item while Inventory is Empty

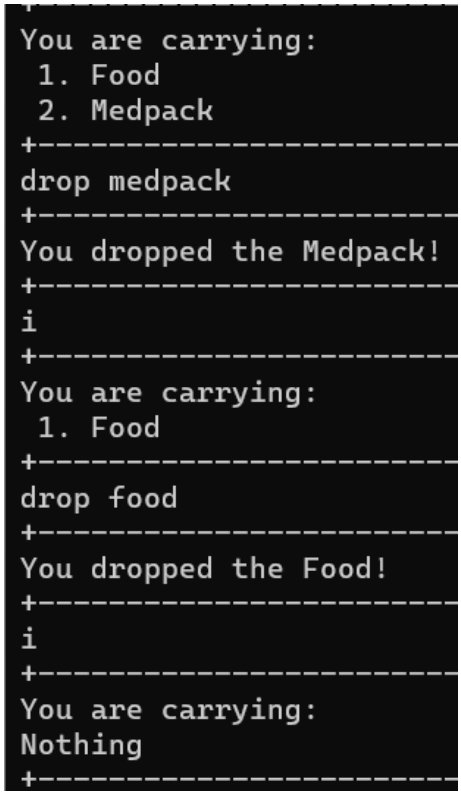


Figure 2.7: Dropping Valid Items

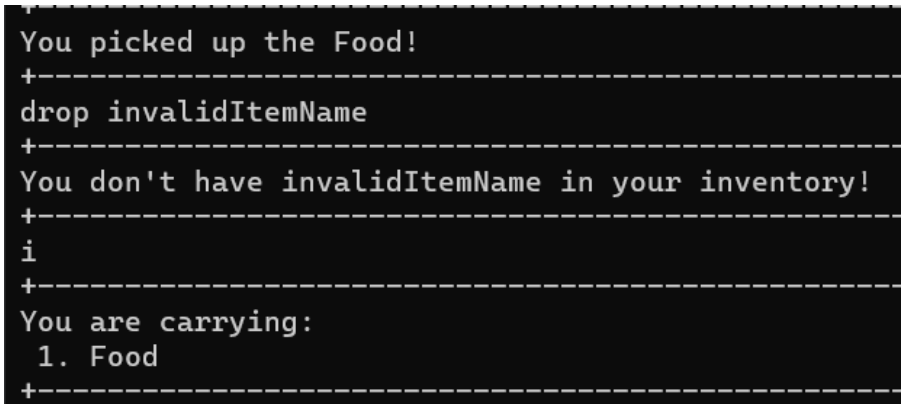


Figure 2.8: Dropping Invalid Items

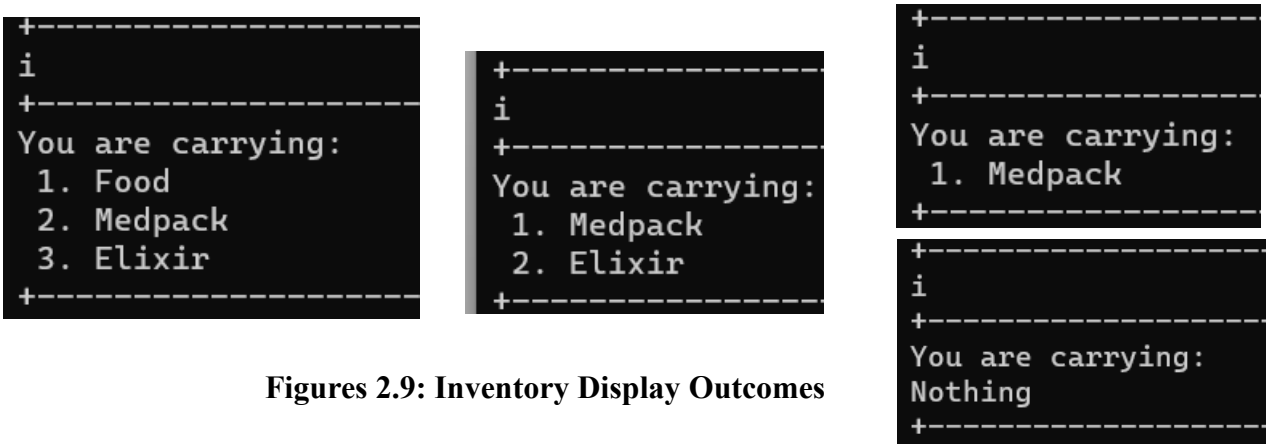
3.2.3 PRINT INVENTORY FUNCTION

This function displays what the player currently is carrying in his or her inventory. It does this by checking all the values of the item location in the game by using a for loop to iterate through all 12 indexes of the item location array. If the item is in the players inventory(value of -2), it will display the item with the item corresponding to the matched index. It is also worth mentioning that a variable will be incremented by 1 if the loops returns true for an item. If the loop fails to match any item location with the value of -2, the function will simply display that the player is carrying nothing. This can be achieved as through a simple if statement where the condition is if the variable is equal to 0, as it will not be incremented if the player is not carrying anything. Code Block 1.6 shows the code explained in this section.

```c
/*ITEM FUNCTION*/
void printInventory(adventureGame *ag){
    int i=0,j=0;
    printf("You are carrying: \n");
    for (i=0,j=0;i<ag->totalRoom;i++){
        if (ag->itemLoc[i]==-2){
            printf("%2d. %s\n",++j,ag->itemArray[i]);
        }
    }
    if (j==0){
        printf("Nothing\n");
        return;
    }
}
```

Code Block 1.6: Inventory Display Function

Figures 2.9 show examples for the outcome of this function.



Figures 2.9: Inventory Display Outcomes

3.3 DAMAGE FUNCTIONS

The members of the struct related to damage dealing are as follows, ag.monsterArray is an array of constant characters which has 13 elements, which stores the name of all the monsters in the game, indexed according to the room they reside in. ag.monsterStatusArray is an array of integers with 13 elements, which has a purpose of recording the health or status of the monster in each room. ag.monsterDesc is an array of constant characters that stores the description of the monster when encountered. ag.monsterDeath is an array of constant characters that stores strings for the message printed when the monster dies in combat for the first time. ag->damage is an integer that can be updated based on the weapon that is available to the player which always deals the highest possible damage at the current moment. Finally, ag.monsterDead is an array of constant characters with 13 elements which has the messages that will be printed when the player attempts to kill the monster which is already dead.

3.3.1 ATTACK DAMAGE

As mentioned in before, the player is able to pick up various equipment scattered throughout all 13 rooms of the game, which will give the player an edge in battle. In this adventure game program, the player with no attack items in his or her inventory would deal 1 damage to everything. The sword deals 4 damage to all enemies except for the Dracula and his bride which deals only 2 damage. The silver spear deals 3 damage to all enemies except for the werewolf in which case it deals 6. The Diamond Cross deals 2 damage to all enemies except for the Dracula and his bride in which case it deals 6.

As long as the player has a weapon in his or her inventory, the program should be able to dish out the maximum amount of damage that is possible at the given moment by automatically choosing the highest damage weapon, even with multiple weapons on hand. To do this, a for loop is implemented to iterate all indexes of the item location and check if any items are in the players possession. It would then go on to check if the items taken are any attack items by comparing the item with the item name specified(sword, silver spear or diamond cross). Therefore, 3 separate if statements were deployed for 3 possible damage values in normal circumstances. It is important to note that this logic would be flawed if the player was carrying more than 1 item, in which case there will be both cases the item location and item checker returns true, which will mess up the damage dealt. To combat this, the damage of the player is

always initialized to 1 at the start of the function, and additional conditions were added to the if statements which checks if the damage the player is dealing is the maximum amount of damage. This way, the program would only return true for one if statement.

As mentioned above, specific monsters are resistant to certain weapons. Therefore, another block of code has to be implemented for the player to deal specific amounts of damage to the monster in interest. This can be easily achieved by simply utilizing if statements which checks if the player has the weapon that is effective towards the monster in his possession(item location is -2), and if the room that the player is in is where the monster is at. When these conditions return true, the damage will be updated accordingly. Code Block 1.7 shows the code that was explained above.

```
/*DAMAGE CALCULATION*/
int attackDamage(adventureGame *ag){
    ag->damage=1;
    for(int i=0;i<ag->totalRoom;i++){
        //DAMAGE FOR NEUTRAL CIRCUMSTANCES
        if(ag->itemLoc[i]==-2){
            if (strcasecmp(ag->itemArray[i], "Sword") == 0 && ag->damage<4){
                ag->damage=4;
                }
            if (strcasecmp(ag->itemArray[i], "Silver Spear") == 0 && ag->damage<3){
                ag->damage=3;
                }
            if (strcasecmp(ag->itemArray[i], "Diamond Cross") == 0 && ag->damage<2){
                ag->damage=2;
                }
        }
    }
        //DAMAGE FOR SPECIAL CIRCUMSTANCES
        if(ag->itemLoc[11]==-2 && ag->room==4){
            ag->damage=6;
        }
        if(ag->itemLoc[9]==-2 && (ag->room==6 || ag->room ==10)&& ag->damage<6){
            ag->damage=6;
        }

        if(ag->itemLoc[5]==-2 && (ag->room==6 || ag->room ==10)&& ag->damage<2){
            ag->damage=2;
        }
}
```

Code Block 1.7: Attack Damage Function

This function is not a command that can be executed by the player, but rather a function that aids in the damage selection which is used in later parts of the code. The outcome of this code will be made apparent in future examples.

3.3.2 CRTICAL HIT FUNCTION

This function adds some degree of randomness to the game, where the player is granted a 10% chance for a critical hit, which deals twice the amount of the normal damage. This function accepts an integer as a parameter, which is damage. Through the rand function, a set of pseudo random values can be obtained. However, this only generates pseudo random number, not the probability of an event occurring. To obtain the probability, a formula where the value obtained is divided by the maximum number that the rand function can pick from is devised. The outcome is typecasted as a double to obtain the probability. If the probability is less than or equal to 0.1(10%), the function will print a message to notify the player they landed a critical hit and returns twice the damage to the function. If it returns false, regular damage is just applied. It is important to note that these pseudo random numbers occur in the same pattern each time the program is run. In other words, the occurrence of a critical hit will be deterministic, which is not ideal. To fix this, the srand function and the time library can be imported to seed the random number generator, where time(NULL) is the parameter for srand, as the time(NULL) returns the current time in seconds which is a unique value. This ensures that the values generated varies each time the program is run. Code Block 1.8 shows the code written for the critical hit function. The seed for the rand function will not be shown in the code block below as it is declared in the main file. Similar to the attack damage function, this function is an extension that helps in calculation of damage and cannot be called as a command by the player.

```c
/*RNG + DAMAGE CALCULATION*/
int crit(int damage){ //critical hit for 2x damage
    double P =(double)rand()/RAND_MAX;
    if(P<=0.10){  //10% chance
        printf("You land a critical hit!\n");
        return 2*damage;  //returns twice the damage
    }
    return damage;  //if fail, return regular damage
}
```

Code Block 1.8: Critical Hit Function

3.3.3 DODGE FUNCTION

This function provides the player with an edge when fighting against enemies, whereby the player has a 20% chance to completely dodge the opponents attack, receiving 0 damage. The logic applied in this function is identical to the critical hit function, whereby values are tweaked. The dodge function also accepts a integer parameter which is damage, if the probability statement returns true, a message stating that the player has dodged the attack is printed, and the damage returned will be 0 (0*damage = 0). If it fails, the player will just take damage as usual. Code Block 1.9 shows the code for the dodge function.

```
/*RNG + DAMAGE MITIGATION*/
int dodge(int damage){  //DODGE FOR 0 DAMAGE
    double P =(double)rand()/RAND_MAX;
    if(P<=0.2){
        printf("You dodged the attack!\n");
        return 0*damage;
    }
    printf("You take damage!\n");
    return damage;
}
```

Code Block 1.9: Dodge Function

3.3.4 PROCESS DAMAGE MONSTER FUNCTION

This function processes the damage dealt to the monster during combat. When the player types “killmonster” or “km” or “killMonster”, the program will recognize that the player intends to harm the monster. Essentially, there are 3 states of a monster in the game. There could be no monster in the room, the monster is alive or the monster is dead. The status of the monster can be determined through the monster status array stated previously where the values for each state respectively are -1, the health of the monster which is greater than 0, or 0. If the player is in a room where no monsters reside(-1), a message will be printed that notifies the player that no monsters are there for them to slay, and the function is returned.

If the player is in a room where the monster is already dead(0), a message will be printed accordingly to notify the player.

Finally, if the player is in a room where the monster is alive, the player is allowed to do combat with it. The attack damage function is called to calculate the damage that will be dealt based

on the attack items in the player's inventory. The monster status array of the monster in combat will then be reduced said amount of damage, by calling the critical chance function to factor in probability for a critical hit to occur. Allowing the status value of the monster to decrease beyond 0 would not be ideal. Hence, a function called checkNegative will ensure that the status value of the monster will be limited to decrease up until 0. This function will be explained in further detail in later parts of the report.

Additionally, when the monster is alive and takes damage(not 0), a message will be printed to notify the player that the monster has taken damage. If the monster dies during combat, messages will be printed to notify the player that they defeated the monster. This condition is packaged under the if statement for when the player is able to combat the monster, to ensure a clear difference that the monster can only dies once and does not interfere with the condition where the player tries to kill an already dead monster, where the condition is also if the monster status array is a value of 0. Code Block 2.0 shows the code explained above. Outcomes of the code will be shown at the end of this damage function segment as this function involves many other functions that have yet to be discussed.

```
/*KILL FUNCTION*/
void processDamageMonster(adventureGame *ag) {
    if (ag->monsterStatusArray[ag->room] == -1){
        printf("There is nothing here to kill...\n");

    } else if (ag->monsterStatusArray[ag->room] == 0) {
        printf("%s", ag->monsterDead[ag->room]);

    } else if (ag->monsterStatusArray[ag->room] > 0){
        attackDamage(ag);
        ag->monsterStatusArray[ag->room]-=crit(ag->damage);
        checkNegative(ag);

        if (ag->monsterStatusArray[ag->room]!=0){
            printf("The %s takes damage!\n", ag->monsterArray[ag->room]);
        }
        if (ag->monsterStatusArray[ag->room]==0){
            printf("The %s falls to the ground...\n",ag->monsterArray[ag->room]);
            printf("%s",ag->monsterDeath[ag->room]);
        }
    }
    return;
}
```

Code Block 2.0: Damage Monster Function

3.3.5 ALTERNATE PROCESS DAMAGE MONSTER FUNCTION

This function is just an alternate function for the function explained above. The key difference is in the way the player calls the command. For this function, the player is able to specify the monsters name that they want to kill, ("kill (monster name)"). This can be achieved through a pointer to a character with the address of the fifth index of the command typed by the user. Through the strcasecmp function, the program will compare the typed monster name with the name of the monster in the current room. If it returns true it will execute the following code accordingly. The logic for determining if the monster is dead, alive or not in the game, and how to deal damage to the monster is identical to the previous function. Some minor differences are if the player types an invalid monster name, a message is printed to notify the player that the monster is invalid. Code Block 2.1 shows the alternate process kill code.

```
/*ALTERNATE KILL FUNCTION*/
void alternateDamageMonster(adventureGame *ag) {
    char *monsterName = &ag->command[5];
    if (monsterName[0] == '\0') {
        printf("You can't kill something that doesn't exist...\n");
        return;
    }
        if (strcasecmp(monsterName, ag->monsterArray[ag->room]) == 0) {
            if (ag->monsterStatusArray[ag->room] == -1) {
                printf("There is nothing here to kill...\n");
                return;
            } else if (ag->monsterStatusArray[ag->room] == 0) {
                printf("%s", ag->monsterDead[ag->room]);
                return;
            } else if (ag->monsterStatusArray[ag->room] > 0) {
                attackDamage(ag);
                ag->monsterStatusArray[ag->room] -= crit(ag->damage);
                checkNegative(ag);
                processDamagePlayer(ag);

                if (ag->monsterStatusArray[ag->room] != 0) {
                    printf("The %s takes damage!\n", ag->monsterArray[ag->room]);
                }
                if (ag->monsterStatusArray[ag->room] == 0) {
                    printf("The %s falls to the ground...\n", ag->monsterArray[ag->room]);
                    printf("%s",ag->monsterDeath[ag->room]);
                }
            }
        }
        else printf("Maybe you should try looking around...\n");
```

Code Block 2.1: Alternate Process Kill Function

3.3.6 PROCESS DAMAGE PLAYER FUNCTION

This function allows the monster to retaliate when the player damages them. In this adventure game, most monsters will deal different amounts of damage to the player. Zombie and Monster will deal 1 damage to the player, Lizard-Man will deal 2 damage, the werewolf will deal 4, Dracula and dracula's bride will deal 5 and the dragon will deal 6 damage.

The function first checks if the status of the monster in the room using an if statement. If it is alive, it then goes on to check the location of the player for the monster to deal the appropriate amount of damage to the player. Therefore, several if statements with the condition being the location of the player were used to ensure the player receives the proper amount of damage. The health of the player will decrease accordingly but also calling the dodge function into the calculation, to factor the chance for the player to dodge the attack. If the status of the monster is dead, the function simply returns.

After damage calculation is done, the checknNegative function is called again to ensure that the health of the player does not drop beyond 0. The health of the player is then printed accordingly.

To account for the fact that the might die in combat, an additional if statement is added in which the condition was if the health of the player reaches 0, it will notify the player that they died, displays the score, and terminates the program with the exit(0) function. Code Block 2.2 shows the code written for the section explained above.

```
/*KILL FUNCTION*/
int processDamagePlayer(adventureGame *ag){
    if (ag->monsterStatusArray[ag->room]>0){  //Zombie and Monster
        if ((ag->room == 2) ||(ag->room == 5)){
            ag->healthPoints-=dodge(1);
            checkNegative(ag);
        }
        if (ag->room == 9){ //Lizard-Man
            ag->healthPoints-=dodge(2);
            checkNegative(ag);
        }
        if(ag->room == 4){  //Werewolf
            ag->healthPoints-=dodge(4);
            checkNegative(ag);
        }
        if((ag->room == 6) || (ag->room == 10 )){ //Dracula and Dracula's Bride
            ag->healthPoints-=dodge(5);
            checkNegative(ag);
        }
        if(ag->room == 12){  //Dragon
            ag->healthPoints-=dodge(6);
            checkNegative(ag);
        }
    }
    else return;
    if (ag->healthPoints==0){
        printf("You died.\n");
        score(ag);
        printf("Score: %d",ag->exPoints);
        exit(0);
    }
    printf("Health: %d\n",ag->healthPoints);
```

Code Block 2.2: Damage Player Function

3.3.7 CHECK NEGATIVE FUCTION

This function checks if the player health and the enemies health would drop beyond zero. It achieves this by using the terniary operator, the deciding factor is whether the health is greater than zero. If it is, the health value would remain the same. However, is this condition is false, the health value would be updated to be zero. This serves as a clear and concise way to check the health of the player and the monster. Code Block 2.3 shows the check negative function code.

```
/*NEGATIVE VALUE CHECKER*/
void checkNegative(adventureGame *ag){  //RESETS NEGATIVE VALUES
    ag->monsterStatusArray[ag->room]= ag->monsterStatusArray[ag->room]>0 ? ag->monsterStatusArray[ag->room] : 0;
    ag->healthPoints= ag->healthPoints>0 ? ag->healthPoints: 0;
}
```

Code Block 2.3: Check Negative Function

The section below displays several outcomes during battle and also show the probability of critical chance to land and dodge to occur. The damage dealt will also be displayed. These information will only be shown as proof in the report, and will not be displayed in actual gameplay. Figure 3.0 to 3.6 depicts several possible outcomes regarding combat.

```
+------------------------
km
+------------------------
Damage: 4
Crit: 0.307230
Monster Health: 1
The zombie takes damage!
Dodge: 0.928709
You take damage!
Health: 11
+------------------------
i
+------------------------
You are carrying:
 1. Sword
+------------------------
```

Figure 3.0: Damage against Neutral Monsters

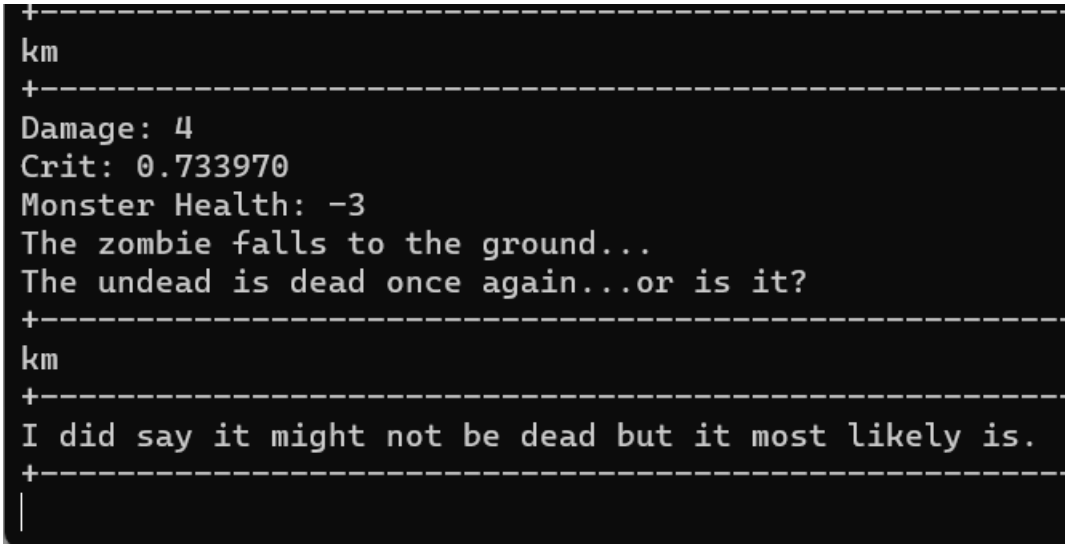


Figure 3.1: Monster Died and Cannot be Slain Again

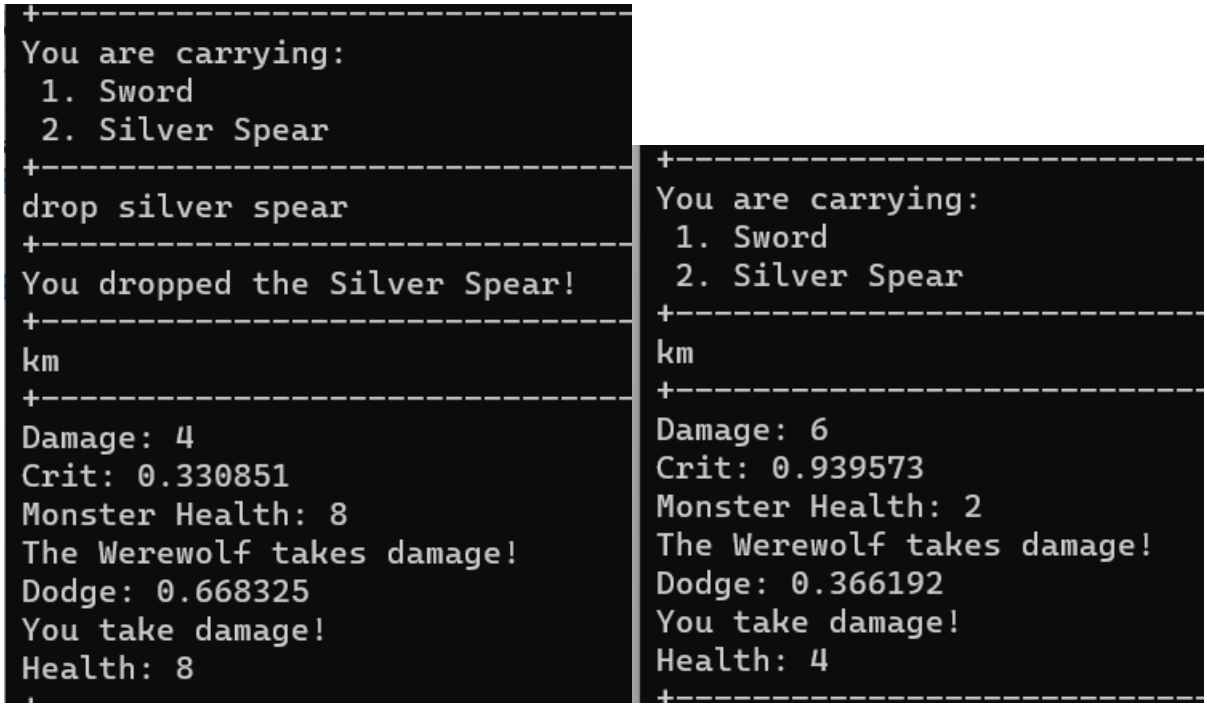


Figure 3.2: Highest Damage Output Weapon is Automatically Chosen

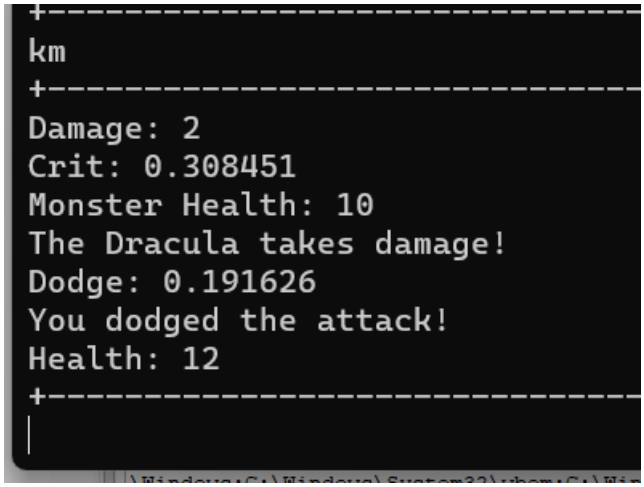


Figure 3.3: Reduced Damage Dealt to Specific Enemies

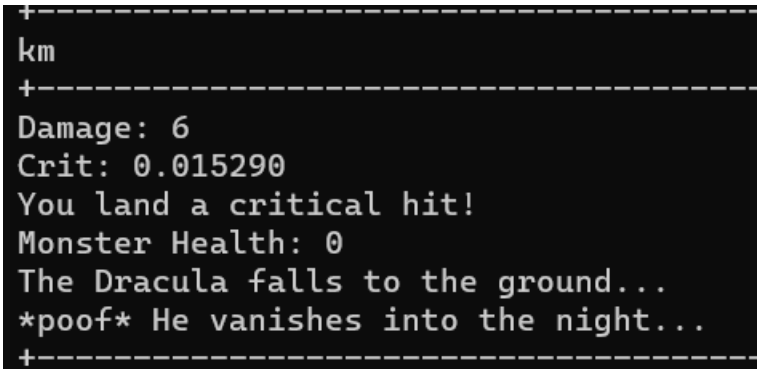


Figure 3.4: Extra Damage By Using the Correct Weapon

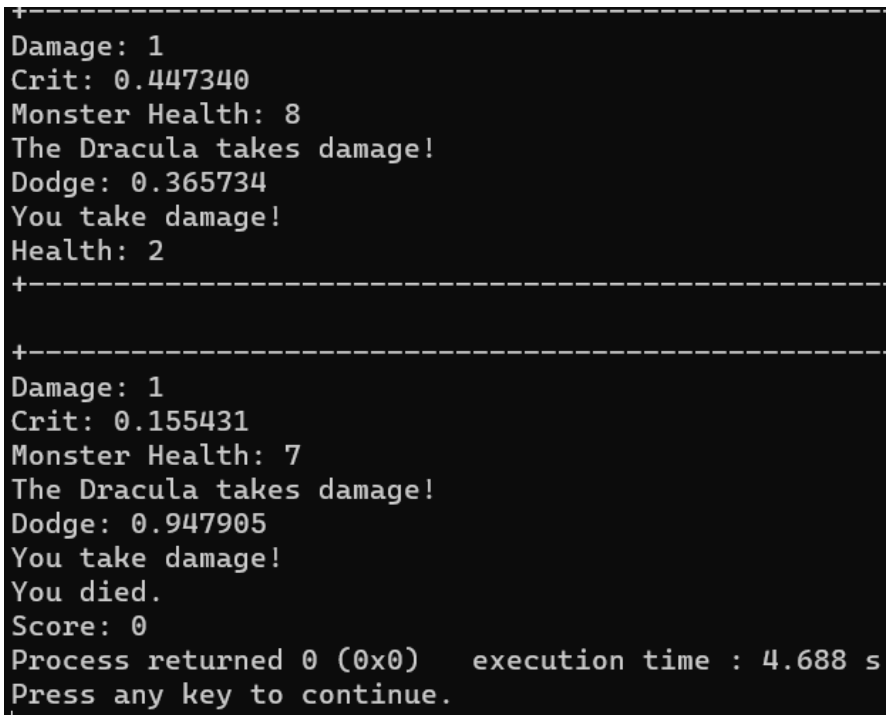
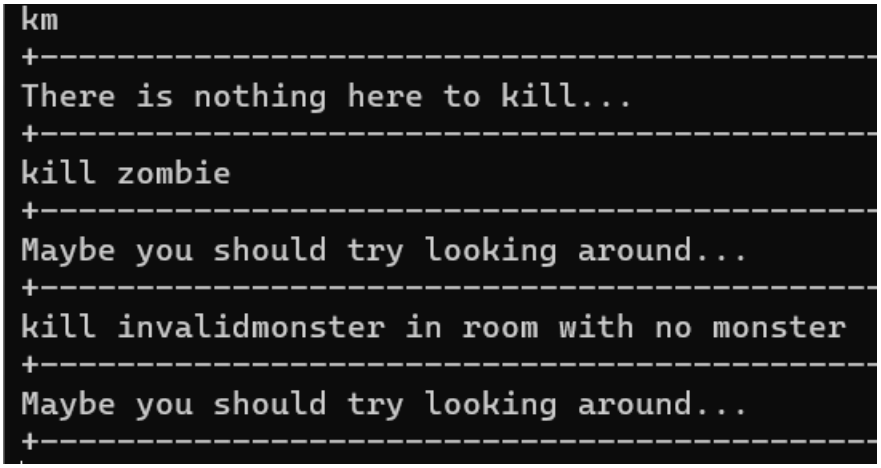
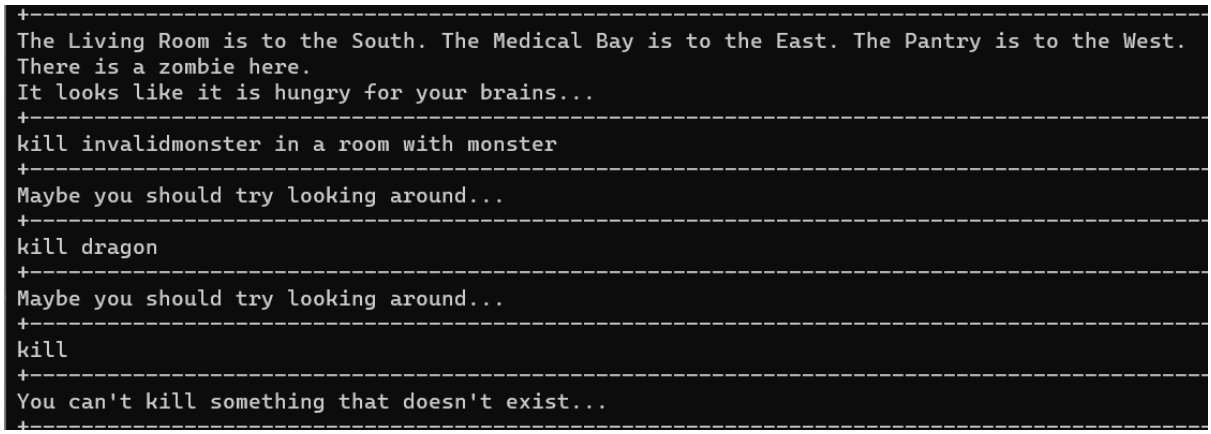


Figure 3.5: Program Terminates when Player Dies





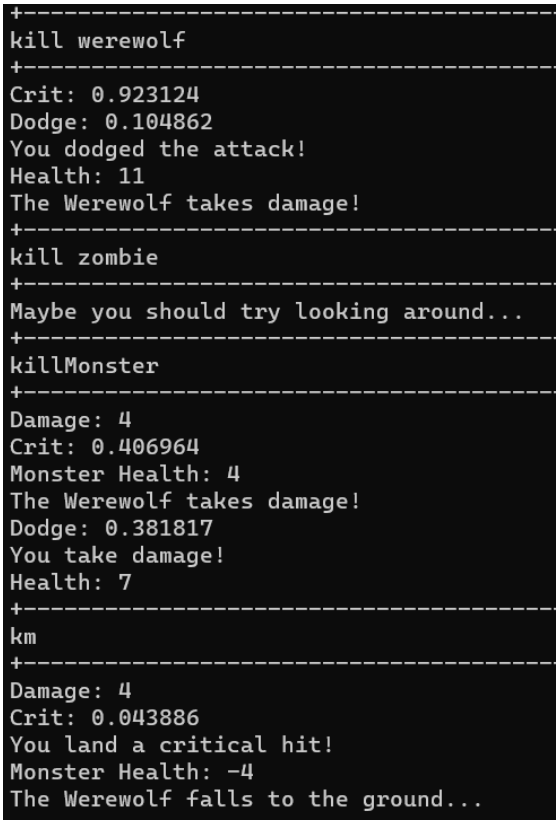


Figure 3.6: Error Messages for Different Circumstances

3.4 LOOK FUNCTION

The members of the struct adventure game related to the look function will be briefly explained here. ag.rooms as previously mentioned, is an array of constant characters that stores the name of all the rooms in the game. ag.moveToArray is a 2D array that contains the value of the room that the player will be in if the player decides to go in that direction. -1 means that the player is not allowed to travel in said direction, -2 means that that area is locked and any value greater 0 means that player is allowed to go in that direction. ag.direction is an array which contains the direction the next room is at. ag.monsterStatusArray is an array of integers with 13 elements, which has a purpose of recording the health or status of the monster in each room. ag.monsterDesc is an array of constant characters that contains the description of the monster encountered in the room. ag.monsterArray is an array with all the names of the monster in the game. ag.itemArray is an array with all the name of the items in the game and ag.itemDesc contains the description.

3.4.1 PROCESS LOOK FUNCTION

This function provides the player with information on his whereabouts by describing the room around them when called upon. The function is able to inform the player on where to go next, the status of the monster in the room if there are any and the item that is currently present in the room.

To start, the function processes the rooms adjacent to room the player is at. With a nested for loop in tandem with an if statement to loop through all possible directions and to check whether the room is accessible(value of movetoArray is not -1), the function will then infrom the playetr by printing the name of the room and the direction the player must travel to. The index used to access the possible room that the player can enter is the condition of the if statement mentioned above, as ultimately the value returned for the if statement will be the room value of the room that is accessible. It was mentioned that the condition to check whether the player is able to move in a specified direction is checked by using an if statement where the condition is if the value of the moveToArray is not -1. This logic has a flaw whereby locked rooms have a value of -2, which also returns true for the if statement. This results in the direction to travel to go to locked rooms not being printed. The player should be notified of the rooms existence although it is locked. To fix this issue, the condition for the if statement was changed to greater than -1,

and two additional if statements were added to print the messages for locked rooms. By checking whether the value is equal to -2 and the room the player is standing in is adjacent to the locked room, the function can successfully handle the case for locked rooms. This if statement will not be entered once the locked room is unlocked and the value of the moveToArray is changed to the room value, whereby the initial if statement will be true and still print the direction of the room with no issues. Figure 3.7 to 3.9 shows the output for notifying the player about the direction of adjacent rooms.

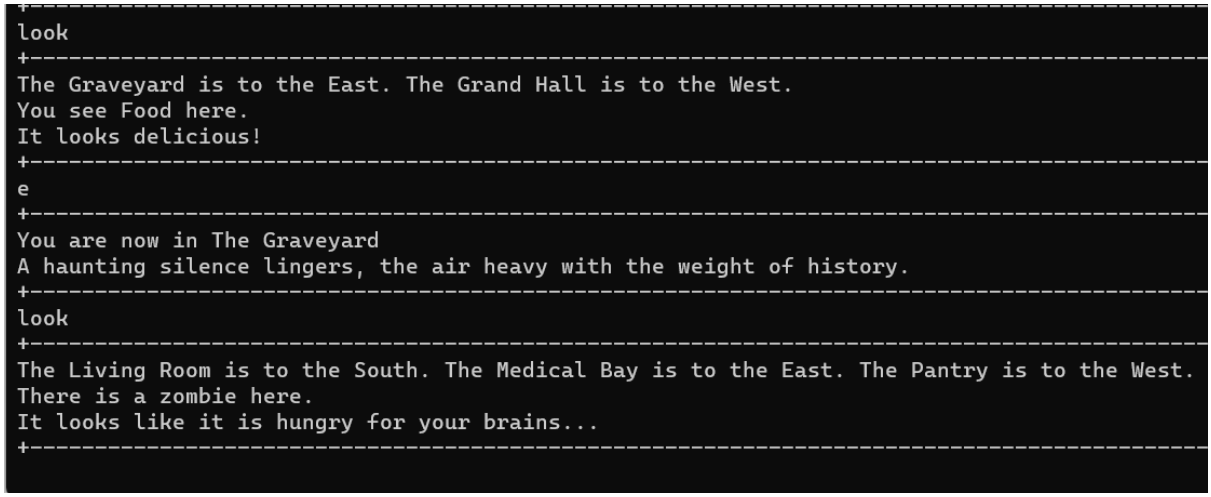


Figure 3.7: Display Direction and Name of Adjacent Rooms

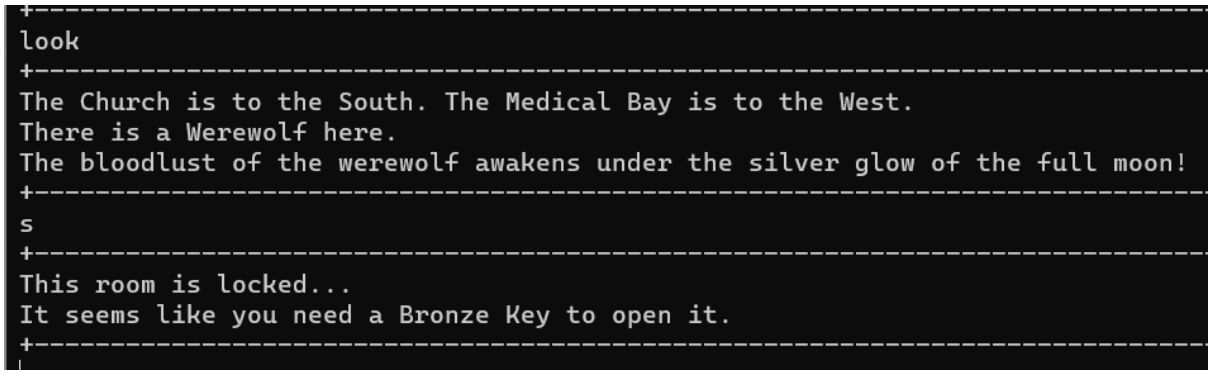


Figure 3.8: Message Prints Although Room is Locked

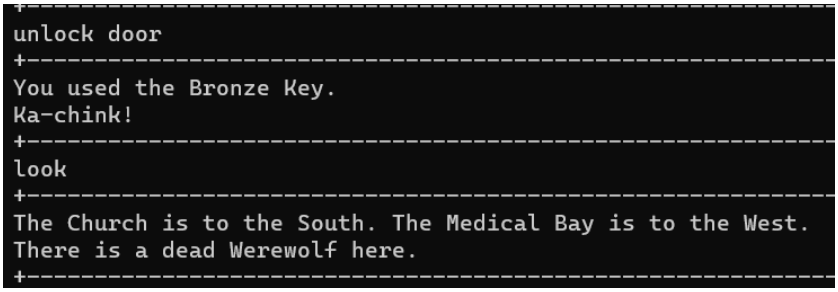


Figure 3.9: Message Prints as Normal After Unlocking Door

Next, the function try to print the status of the monster in which case, the monster could only be alive or dead (value greater than 0 or 0). The function can check for these conditions by using two if statements. If it is alive, the function will print the name and the description of the monster by using the value of the room the player is in as an index to access each respective arrays for the name and description. If the monster is dead, the same logic is applied but instead a message signifying the monster is dead is printed. If there are not monsters present in the room, these two if statements will not return true and therefore be skipped. Figure 4.0 to 4.2 shows various output for calling the look command for the status of the monster.

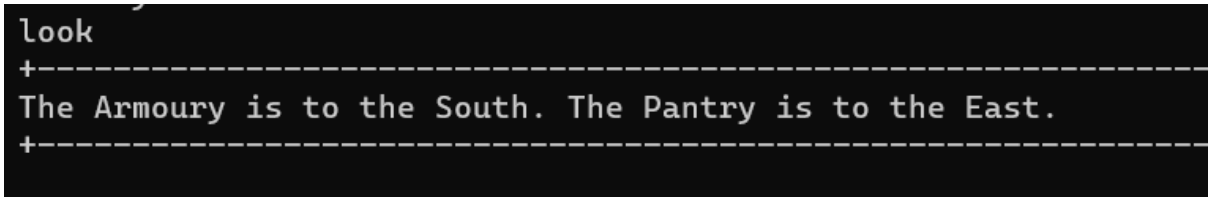


Figure 4.0: No Monster Present in the Room

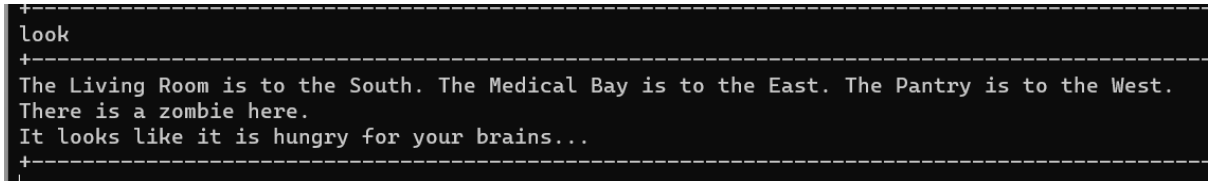


Figure 4.1: Monster Status and Description is Printed When Alive

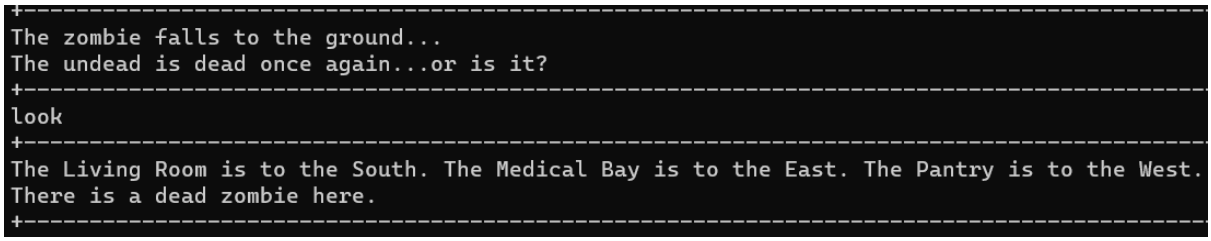


Figure 4.2: Monster Status is Dead and Message is Displayed

Finally, the function will check for items in the room. By using a for loop to iterate through the item location array, the function is able to check whether an item is present in a room where the player is standing. This is possible as the item location array has 13 elements, where each element's value corresponds to the value of the room they are currently residing in. If the value of the item location matches the room the player is in, the item is able to be picked up, and the player is notified as such by printing a message stating that the player sees an item in the room. At the same time, the item description is also displayed. Figure 4.3 to ??? shows output regarding items for the look function.

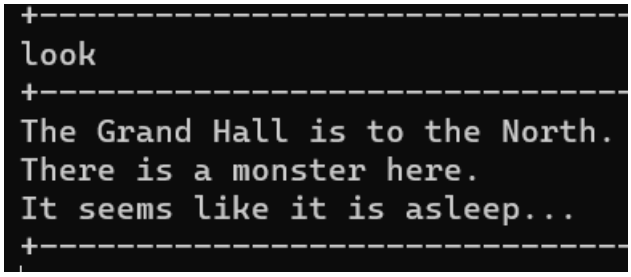


Figure 4.3: No Item Present in the Room

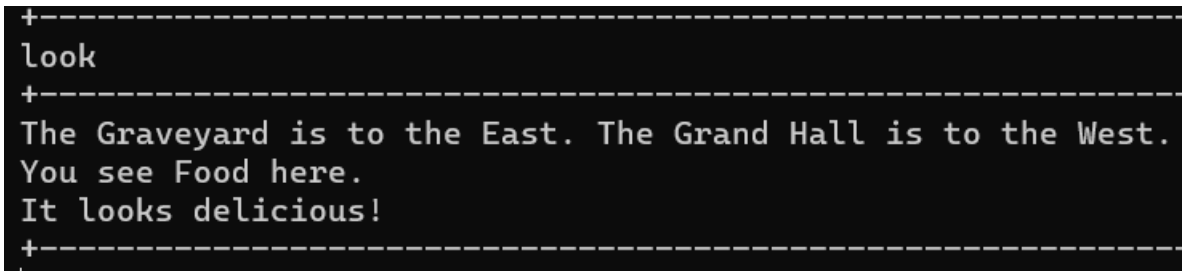


Figure 4.4: Item Name and Description is Printed when Spotted

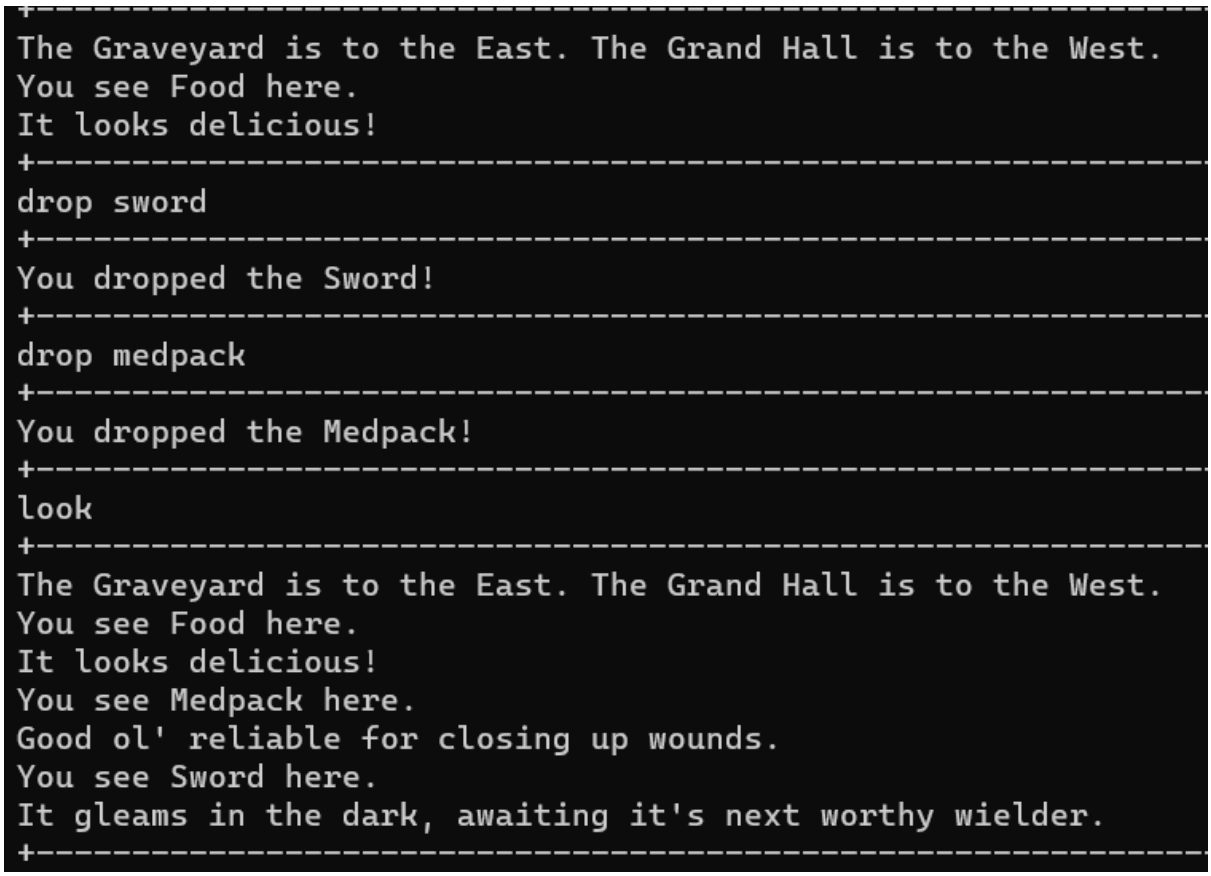


Figure 4.5: Message is Printed For All Items Currently in the Room

Code Block 2.3 depicts the code which was typed for the process look function. The code is also commented to be able to communicate the critical principles behind the each part of the code.

```
/*LOOK FUNCTION*/
void processLook(adventureGame *ag) {
    for(int i=0;i<4;i++){  //loop through all possible directions
        if(ag->moveToArray[ag->room][i]>-1){  //if room is accessible
            printf("%s is to the %s. ",ag->rooms[ag->moveToArray[ag->room][i]],ag->direction[i]);
        }
/*FOR LOCKED ROOMS*/
        if(ag->moveToArray[ag->room][i]==-2 && ag->room == 4){  //-2 means room is locked
            printf("%s is to the %s. ",ag->rooms[9],ag->direction[i]);
        }
        if(ag->moveToArray[ag->room][i]==-2 && ag->room == 11){
            printf("%s is to the %s. ",ag->rooms[12],ag->direction[i]);
        }
    }

    printf("\n");
/*FOR MONSTER*/
    if (ag->monsterStatusArray[ag->room]>0){  //if monster is alive
        printf("There is a %s here.\n",ag->monsterArray[ag->room]);
        printf("%s",ag->monsterDesc[ag->room]);
    }
    else if (ag->monsterStatusArray[ag->room]==0) {  //if monster is dead
        printf("There is a dead %s here.\n",ag->monsterArray[ag->room]);
    }
/*FOR ITEMS*/
    for (int i=0;i<ag->totalRoom;i++) {  //loops through all indexes of item location
        if (ag->itemLoc[i] == ag->room){  //checks if the item is in the room
            printf("You see %s here.\n",ag->itemArray[i]);
            printf("%s",ag->itemDesc[i]);
        }
    }
}
```

Code Block 2.3: Look Function

3.5 HEAL FUNCTION

The members of the struct relating to healing commands will be explained in this section. ag.healthPoints is an integer that can be modified at any point. It is useful as the value will change depending on if the player takes damage or regains health. ag.inventory is an integer that is updated based on the number of items the player is carrying. ag.itemLoc is a integer array of 13 elements that has values representing the location of all the items in the game which are assigned to the room number of each item that resides there. A value of -1 represents that the room does not have an item or the item is removed from the game, a value of -2 represents the item is in the players inventory and a value greater than 0 represents the item is in the room it is placed at.

3.5.1 PROCESS (EAT,USE,DRINK) FUNCTION

This function allows the player to regain health by inputting the heal commands which are (“eat/use/drink (heal item)”). Note that the item name inputted for the 3 heal commands food, medpack and elixir respectively or the command will not run. All three functions have identical logic except for minor index differences. The function starts by declaring a pointer to a character and initializing it as the item name typed by the player. This will be used to check whether the item name typed by the player matches the name of the item in the array by using the strcasecmp function. First the function checks whether the player is at full health, if the player is, it would not allow the player to heal and returns the function early. It does this by using an if statement with three conditions that has to be satisfied which is the player typed the correct item name, the item is in the players inventory and the player’s health is full.

The next block of code is responsible for restoring the player’s health if the command is successfully called upon. If the comparison returns true, the player’s health points are incremented by a certain value, (food heals for 5, medpack and elixir heal for 10) and a message is printed to notify the player as such. As the item is consumed, the item will be removed from the players inventory and the item will be effectively removed from the game. This is achieved by setting the value of the item location of the item used to -1, which signifies the item is not in the game. If the comparison returns false, a message printed saying that the item name is not recognized and the function exits.

Additionally, if the player does not have tries to heal with an item that was removed from the game, a message will also printed that states that the player does not have any more healing items. Conversely, if the item is still in the game but not in the player's inventory, a message is printed to inform the player as such. The Process Use Function and Process Drink Function use identical logic and therefore will not be explained again. Figure 4.6 to 5.2 shows some outcomes of the process heal functions.

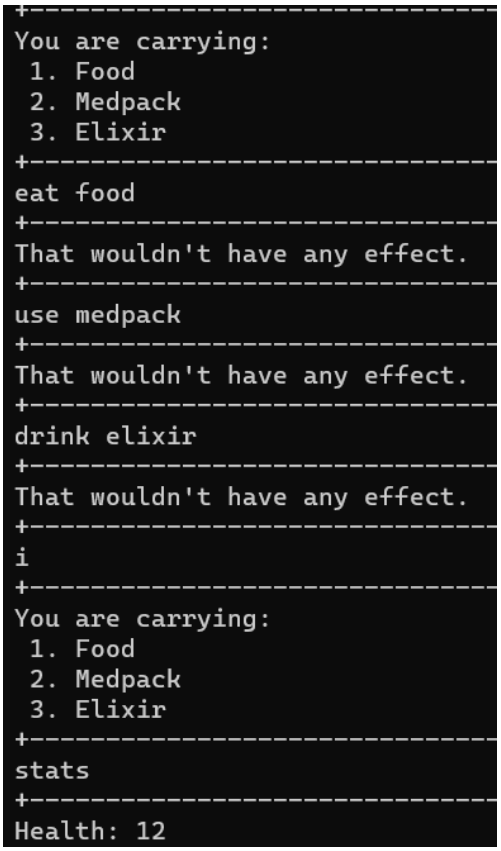


Figure 4.6: Player Is Not Allowed to Heal at Full Health

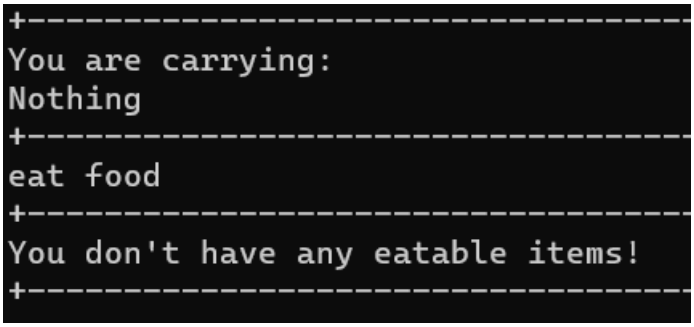


Figure 4.7: No Eatable Items

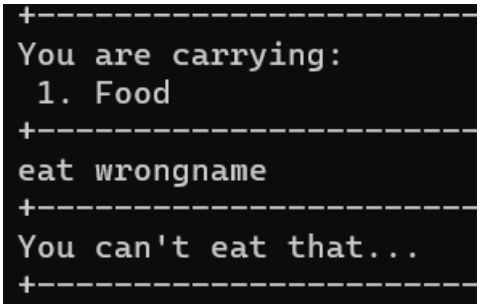


Figure 4.8: Invalid Item Name

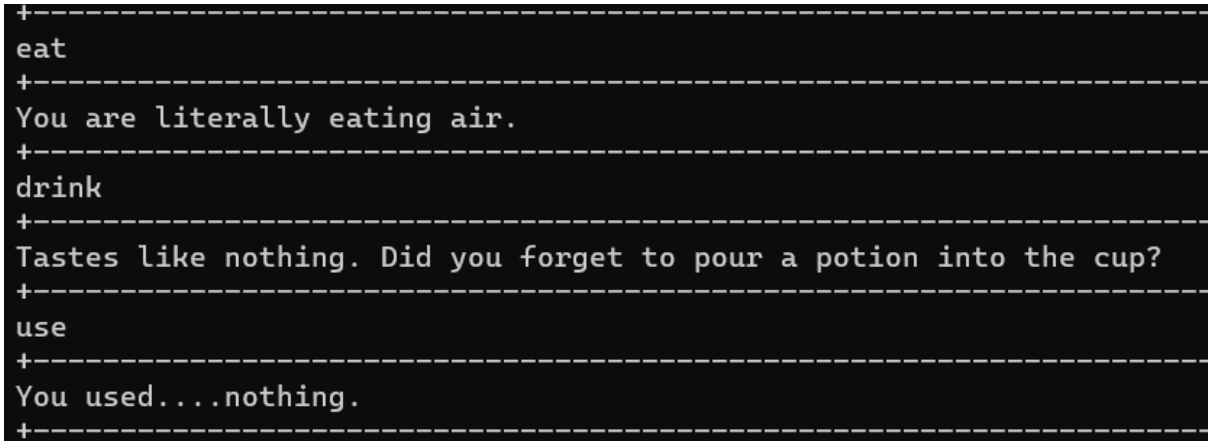


Figure 4.9: No Item Name Typed

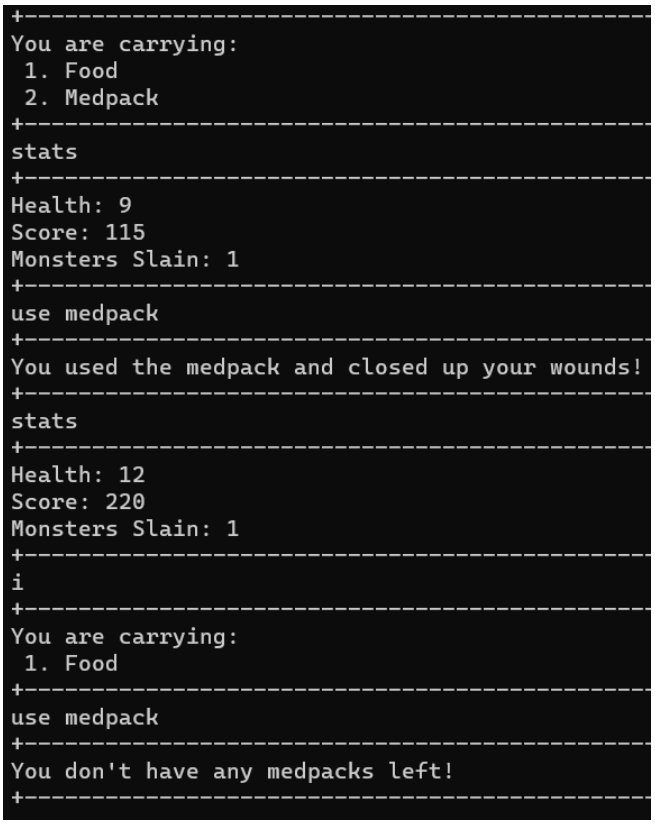


Figure 5.0: Message Printed if Item Is Removed From the Game

Code Block 2.4 shows the code typed for the process “heal” commands.

```c
/*HEALING FUNCTION*/
void processEat(adventureGame *ag){
    char *itemName = &ag->command[4];
    if (itemName[0] == '\0') {
        printf("You are literally eating air.\n");
        return;
    }
    if(ag->healthPoints==12 && ag->itemLoc[1]==-2 && strcasecmp(itemName,"Food")==0){
        printf("That wouldn't have any effect.\n");
        return;
    }
    if(ag->itemLoc[1]==-2){
        if(strcasecmp(itemName,"Food")==0){
            ag->healthPoints+=5;
                checkHealth(ag);
                printf("You ate the food and regained some stamina!\n");
                --ag->inventory;
                ag->itemLoc[1]=-1;
        }
        else printf("You can't eat that...\n");
    }
    else if(ag->itemLoc[1]==-1){
        printf("You don't have any food left!\n");
    }
    else if(ag->itemLoc[1]>0){
        printf("You don't have any eatable items!\n");
    }
}
```

```c
/*HEALING FUNCTION*/
void processUse(adventureGame *ag){
    char *itemName = &ag->command[4];
    if (itemName[0] == '\0') {
        printf("You used....nothing.\n");
        return;
    }
    if(ag->healthPoints==12 && ag->itemLoc[3]==-2 && strcasecmp(itemName,"Medpack")==0){
        printf("That wouldn't have any effect.\n");
        return;
    }
    if(ag->itemLoc[3]==-2){
        if(strcasecmp(itemName,"Medpack")==0){
            ag->healthPoints+=10;
                checkHealth(ag);
                printf("You used the medpack and closed up your wounds!\n");
                --ag->inventory;
                ag->itemLoc[3]=-1;
        }
        else printf("You can't use that...\n");
    }
    else if(ag->itemLoc[3]==-1){
        printf("You don't have any medpacks left!\n");
    }
    else if(ag->itemLoc[3]>0){
        printf("You don't have any usable items!\n");
    }
}
```

```c
/*HEALING FUNCTION*/
void processDrink(adventureGame *ag){
    char *itemName = &ag->command[6];
    if (itemName[0] == '\0') {
        printf("Tastes like nothing. Did you forget to pour a potion into the cup?\n");
        return;
    }
    if(ag->healthPoints==12 && ag->itemLoc[8]==-2 && strcasecmp(itemName,"Elixir")==0){
        printf("That wouldn't have any effect.\n");
        return;
    }
    if(ag->itemLoc[8]==-2){
        if(strcasecmp(itemName,"Elixir")==0){
            ag->healthPoints+=10;
                checkHealth(ag);
                printf("You drank the elixir and feel rejuvenated!\n");
                --ag->inventory;
                ag->itemLoc[8]=-1;
        }
        else printf("You can't drink that...\n");
    }
    else if(ag->itemLoc[8]==-1){
        printf("You don't have any elixir left!\n");
    }
    else if(ag->itemLoc[8]>0){
        printf("You don't have any drinkable items!\n");
    }
}
```

Code Block 2.4: Heal Commands

3.5.2 CHECK HEALTH FUNCTION

This function checks if the health of the player goes beyond the maximum health the player has. Similar to the check negative function, this is a lambda function with the condition of whether the health of the player is greater than the maximum(>12). If it isn't, the health points are at a normal value and remains the same. If it is, the health is reset to 12. By equating the health point to this whole ternary function, the value of health point can be checked and changed accordingly whenever the function is called. Code Block 2.5 shows the lambda function for checking the health of the player.

```
/*HEALTH CHECKER*/
void checkHealth(adventureGame *ag){  //reset values beyond max hp to 12
    ag->healthPoints = ag->healthPoints>12 ? 12 : ag->healthPoints;  //is player health beyond max? if yes set to 12
}
```

Code Block 2.5: Check Health Function

3.6 COMMAND FUNCTIONS

The members of the adventure game struct that is related to command functions are as follows. ag.command is an array of characters which is used as a buffer for user input in the terminal. ag.buffer is another array of characters that temporarily store the commands that typed by the user. ag.helpCommands an array of characters that stores the category of similar functions while ag.subHelpCommands store the actual commands and the way they are typed to be able to successfully execute them.

3.6.1 PROCESS COMMAND FUNCTION

This function will process the command typed by the player, and select the correct command to execute based on the command inputted by the user. By using multiple if statements, and comparing the keyword of the command and what the user typed, it will enter the correct if statement accordingly and run the specific command. Once it enters the if statement, the function can be called and ran. As seen in the main file, a while loop is used to constantly obtain

user input as long as the process command function does not return 0. Therefore, every function that is not an exit command will return 1 and break the function so that the program continues to take inputs from the user and does not continue down the loop. Figure 5.1 depicts an example how the program recognize commands and returns a value.

```
if (strcmp(ag->command,"exit")==0) {
    score(ag);
    winCondition(ag);
    exitMessage(ag);
    return(0);
}
if (strcmp(ag->command,"look")==0||strcmp(ag->command,"l")==0) {
    processLook(ag);
    return(1);
}
if(strcasecmp(ag->command,"killMonster")==0 || strcasecmp(ag->command,"km")==0){
    processDamageMonster(ag);
    processDamagePlayer(ag);
    return(1);
}
if (strncmp(ag->command,"kill ",5)==0){
    alternateDamageMonster(ag);
    return(1);
}
```

Figure 5.1: Comparing Keywords with User Input

3.6.2 REPLACE NEW LINE FUNCTION

This function replaces the new line character in the fgets function. Fgets will always generate a new line character at the end of the input. This is not ideal when the player types a command that requires a function to check the name of the item or monster typed, such as the processGet function of alternateKillMonster function as the name that the user typed will automatically have a newline character at the end. When compared with the name of the item or monster in the array, the comparison will always return false as “itemName\n” does not equal to “itemName”. To solve this issue, the new line character generated is replaced with a end of string character(\0). A for loop is implemented to cycle through all characters of the command typed by the user. Once it encounters a new line character, it is replaced with a end of string character. Code Block 2.6 shows the logic behind the function.

```
/*REPLACE NEW LINE*/
void replaceNewLine(adventureGame *ag) {
    for (int i=0;ag->command[i]!='\0'; i++) {  //iterate through all characters of the command
    if (ag->command[i]=='\n') {  //encounters new line
        ag->command[i]='\0';  //replace with end of string
        return;
        }
    }
}
```

Code Block 2.6: Replace New Line Function

This function is called as the first line in the process command function and does not require an if statement, as this function that should be called by the player. Besides, this function should always be active to remove the new line character in every command typed.

3.6.3 HELP FUNCTION

This function displays all possible commands that the player is able to call upon. This function has not much logic behind it but rather formatting using string specifiers. For better reusability, a for loop is used to print the contents of the table should more functions be added in the future. Figure 5.2 shows the output of the command when called.

```
help
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
|                  LIST OF COMMANDS                                             |
+-------------------------------------------------------------------------------+
|    MOVEMENT COMMANDS | north, n, south, s, east, e, west, w                   |
+-------------------------------------------------------------------------------+
|    LOOK COMMANDS     | look,l                                                 |
+-------------------------------------------------------------------------------+
|    FIGHT COMMANDS    | killMonster,killmonster,km, kill (monster name)        |
+-------------------------------------------------------------------------------+
|    GET COMMANDS      | get (item name)                                        |
+-------------------------------------------------------------------------------+
|    DROP COMMANDS     | drop(item name)                                        |
+-------------------------------------------------------------------------------+
|    INVENTORY COMMANDS| inventory, i                                           |
+-------------------------------------------------------------------------------+
|    HEAL COMMANDS     | eat food, drink elixir, use medpack                    |
+-------------------------------------------------------------------------------+
|    UNLOCKING DOORS   | unlock door                                            |
+-------------------------------------------------------------------------------+
|    STATISTICS        | stats                                                  |
+-------------------------------------------------------------------------------+
|    EXIT GAME         | exit                                                   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
```

Figure 5.2: Displays all Possible Commands

3.6.4 REPEAT COMMAND FUNCTION

This function helps to repeat the previous command typed when the user does not type anything(\n). This is achieved by using a simple if else statement. If the command typed by the user is not a new line character, e.g the player types anything, the command is copied into an array of characters, and the function ends. If the player does not type anything, e.g a new line character, the function will not copy the command. Instead, it would replace the command typed by the user (new line character), with the command that was previously copied. This command is not called within the process command function however called inside the while loop of the main file. This is made so that the command can be changed from a new line character to whatever it was previously before it enters the process command function. Code Block 2.7 shows the logic behind this function. Figure 5.3 to 5.4 shows the outcome when new line character is typed.

```c
/*REPEATS COMMAND*/
void repeatCommand(adventureGame *ag){
    if(strcmp(ag->command,"\n")!=0){  //while command typed is not newline character
        strcpy(ag->buffer,ag->command);  //copies the command into a buffer
    }
    else strcpy(ag->command,ag->buffer);  //if it is a newline character, the command is replaced with the buffer command
}
```

Code Block 2.7: Repeat Command Function

```
+--------------------------------------
kill dragon
+--------------------------------------
Maybe you should try looking around...
+--------------------------------------

+--------------------------------------
Maybe you should try looking around...
+--------------------------------------
```

Figure 5.3: Command is Repeated for Valid Commands

```
+------------------------------------------------------------------
invalid command
+------------------------------------------------------------------
Unknown command "invalid command" type "help" for list of commands.
+------------------------------------------------------------------

+------------------------------------------------------------------
Unknown command "invalid command" type "help" for list of commands.
+------------------------------------------------------------------
```

Figure 5.4: Command is Repeated for Invalid Commands

3.7 MISCELLANEOUS FUNCTIONS

The members of the adventure game struct related to the remaining functions in the game are as follows. ag.exPoints is an integer that is used to store the value of experience points the player has. Additionally, ag.exMonsterValue and ag.exItemValue are arrays of integers that store the value of experience points given when a monster is defeated or an item is in the players possession respectively.

3.7.1 SCORE FUNCTION

This function helps to calculate the score of the player upon exiting the game. It achieves this by using a for loop to check the status of the monster and the location of the item. If the monster is dead, the respective index of the monster experience array will be accessed and the value will be added to the experience point of the player. The same logic is applied to item obtained. Although, the items that are in the player's inventory upon exit or completion of the game are only taken into consideration. This results in various combination of scores although the player has ultimately won, creating replayability of the game. Code Block 2.8 shows the logic behind the score function. This function is also an function that helps in calculation of the score and is not able to be called by the player through a command.

```
/*SCORE CALCULATION*/
void score(adventureGame *ag){
    for(int i=0;i<ag->totalRoom;i++){  //loops through every room
        if(ag->monsterStatusArray[i]==0){  //checks for dead monsters
            ag->exPoints+=ag->exMonsterValue[i];  //adds experience points for monster
        }
    if(ag->itemLoc[i]==-2){  //checks for items in inventory
        ag->exPoints+=ag->exItemValue[i];  //adds experience points for item in inventory
    }
    }
}
```

Code Block 2.8: Score Function

3.7.2 STATS FUNCTION

This function displays the total monsters current slain by the player and the health that they have. It simply uses a for loop to iterate through all rooms and checks to see which monster is dead. If it is dead, a variable is incremented and will be displayed as a kill count in the printed message. Code Block 2.9 shows the logic for the stats function. Figure 5.5 shows an output of the stats function.

```
/*STATISTICS FUNCTION*/
void stats(adventureGame *ag){
    score(ag);
    printf("Health: %d\n",ag->healthPoints);
    monsterSlain(ag);
}

void monsterSlain(adventureGame *ag){
    int kills=0;
    for(int i=0;i<ag->totalRoom;i++){
        if(ag->monsterStatusArray[i]==0){
            kills++;
        }
    }
    printf("Monsters Slain: %d\n",kills);
}
```

Code Block 2.9: Stats Function

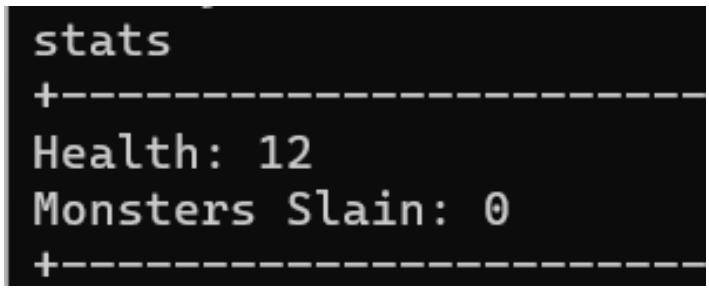


Figure 5.5: Output When Game Initialized

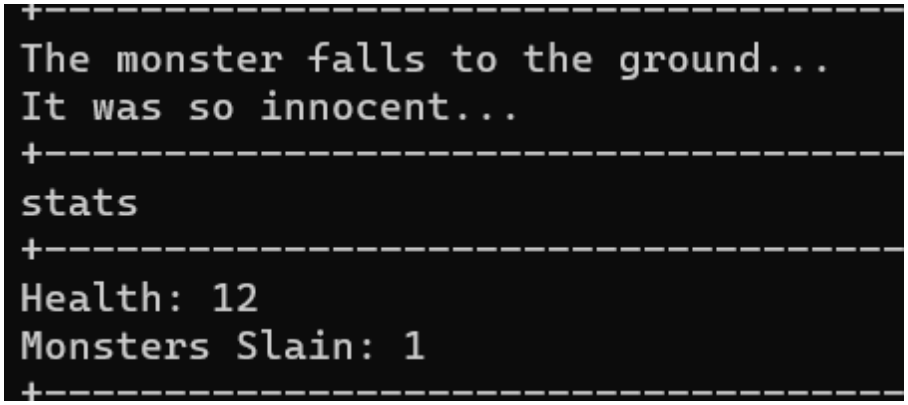


Figure 5.6: Updated Monster Slain

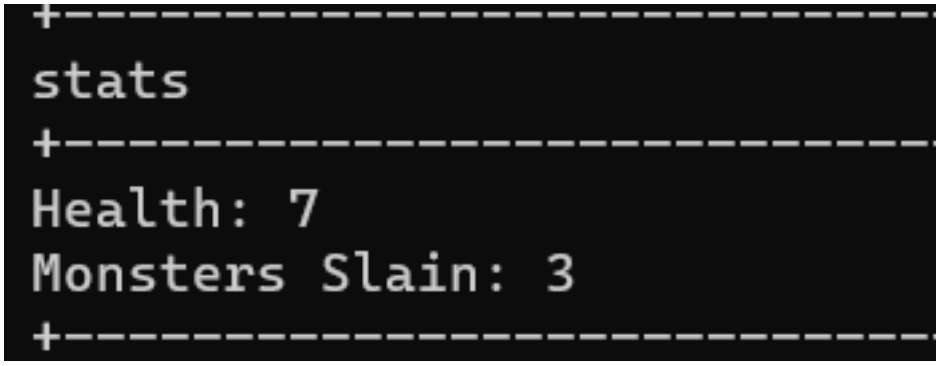


Figure 5.7: Updated Health and Monster Slain

3.7.3 EXIT FUNCTION

This function deals with the player exiting the game by printing a congratulatory message or just simply exiting. If the player is holding the golden chalice and returns to the starting room and exits the game, a message congratulating the player will be printed. This is achieved by using an if statement that checks the location of the player (room value) and the item location of the golden chalice (whether it is -2/in the players inventory). The score and monster slain will also be printed. This is done by calling the score and monsterSlain function to calculate the values of said variables. By using the exit(0) function, the function is terminated and the program stops.

In another case where the player voluntarily exits the game without the golden chalice or winning, a message that says the player exits the game is displayed and the score and monster slain is also printed by using the same logic stated above. It is also important to note that these two functions are separate but called consecutively when the “exit” command is typed. The

condition for winning will be processed first, as the function is able to exit early if the player wins. If the condition does not return true, the function simply moves on to the normal exit condition and returns 0, effectively breaking the while loop and also terminating the program. Code Block 3.0 shows the code written for these two exit functions. Figure 5.8 shows the output for two exit conditions.

```c
/*WIN AND EXIT FUNCTION*/
void winCondition(adventureGame *ag){
    if(ag->room==0 && ag->itemLoc[12]==-2){
        printf("You win!\n");
        score(ag);
        printf("Score: %d\n",ag->exPoints);
        monsterSlain(ag);
        exit(0);
    }
}

/*EXIT FUNCTION*/
void exitMessage(adventureGame *ag){  //Function to print exit message
    printf("You exit the game.\n");
    printf("Score: %d\n",ag->exPoints);
    monsterSlain(ag);
}
```

Code Block 3.0: Exit Function

```
+------------------------
You are carrying:
 1. Medpack
 2. Sword
+------------------------
exit
+------------------------
You exit the game.
Score: 3790
Monsters Slain: 3

Process returned 0 (0x0)
Press any key to continue.
```

```
+------------------------
You are carrying:
 1. Sword
 2. Golden Chalice
+------------------------
exit
+------------------------
You win!
Score: 12775
```

Figure 5.8: Exit When Winning and Not Winning

CONCLUSION

In conclusion, this adventure game program is able to take in a certain set of inputs and execute the commands accordingly when called upon. The fighting algorithm has a set of pre-determined damage for each weapon against each monster and vice versa. However, elements such as critical hits and dodge introduce randomness into the game to create a more interactive and engaging gaming experience. Custom messages for each monster, monster death, item encounter and location descriptions also enriches the gaming experience, by immersing the player in a vivid narrative.

The adherence to modularity principles ensures a well-organized and maintainable codebase, allowing for easy expansion and modification in the future. Some shortcomings of the code are some values are hard coded without a variable which makes it hard to make modifications on a larger scale. A key component in this whole program is the fgets function and arrays. The sizes for both of these functions are declared statically and may be prone to overflow.

All in all, the program is still able to execute commands as expected and provide a proper gaming experience.