Project 1

Title:

Dungeon Crawler

Student: Hassan Farhat

Submitted to Professor Mark Lehr

Due Date: 28th October, 2016

Course and Section:

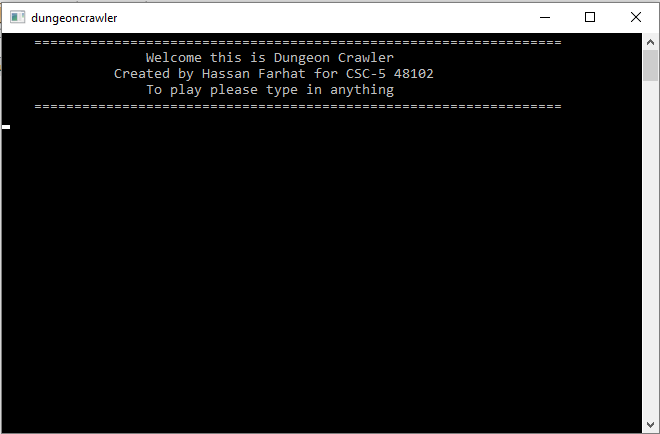
CSC-5, 48102

Table of Contents

1. Introduction
2. Gameplay
3. Code Content:
   1. Number of variables and number of lines.
   2. Table of concepts used and where to find an example.
4. Code Break Up: (Each of these will include at least a psudocode, flowchart, or code sample)
   1. Skill Set Up.
   2. Shop.
   3. Items.
   4. Movement and map.
   5. Spawn and decision to engage.
   6. Battles.
   7. Scoreboard.
5. Pseudocode Representation of the game.
6. Project 2 plans.
7. The full code.
8. **Introduction:**

The adventure game has been around for many years. There are many type of adventure games RPG, MMORPG, JRPG, board games (Dungeons and Dragons) and many more. One of the most prominent features these games have are dungeons. Dungeons are complexes made of many rooms that may contain enemies, and the final room contains a boss. The act of fighting you way through these dungeons is called dungeon crawling.

Dungeon crawling is such a popular concept that many games are solely dungeon crawlers. Many of the first text adventures were dungeon crawlers, and the 1980s is considered the “Golden age” of dungeon crawlers. Early game developers loved to develop these games because they were simple to code, required little if any graphics and were fun. Thus, every gamer has at least one dungeon crawler that they love, be it the modern *Diablo 3* or the old 1981 *Wizardry*).

To honor these games, the application is a classic style dungeon crawler. *Dungeon Crawler* is currently a simple game that has a set number of rooms, and has a set number of rooms. The objective is to navigate through the dungeon, whilst fight off monsters, to fight the boss.

1. **Gameplay:**

The player start the game by typing in anything and hitting enter. Then, the player is required to set up the skills of their character. The player has a total of 10 points that can be distributed amount health, attack, dodge, and luck. This distribution will determine how effective certain actions later in the game are. Attack effects how likely the user is to hit critical hits (does more damage). Agility effect the chance of dodging an attack. Luck effects the ability to sneak past an enemy and how much money is looted. Finally, health is how much health the player has.

After the skill set up, players may purchase items from the store. There are three categories of items in the game. The first will boost player skill points. The second can be used in combat. The third is an item that will revive the player upon death.

The player is now ready to enter the dungeon. They are initially required to move one “O” up to enter the dungeon. However, from now on, the player decides where to move to. With every move, there is a chance that a Monster will spawn. Once a monster spawns, the play can decide to engage it, or try to sneak by. If sneaking fails or the player engaged the monster must be defeated to move one. The final room is the boss room. The player must defeat the boss in 8 turns. The player also collects points throughout the game for accomplishing certain tasks. Points are later shown on the score board if the player won the game. The game ends if the boss is defeated (win), the player ran out of boss fight turns (loose), the player dies at any point without the revive item active (loose).

1. **Code Content:**
2. **Number of variables and lines:**

|  |  |
| --- | --- |
| Line of Code (Just C++ code) | 628 |
| Line of just comments (lines made of only comments) | 70 |
| Total Number of comment lines (including ones after C++ code) | 188 |
| Empty lines | 27 |
| Total lines of Code (Total lines in main) | 725 |

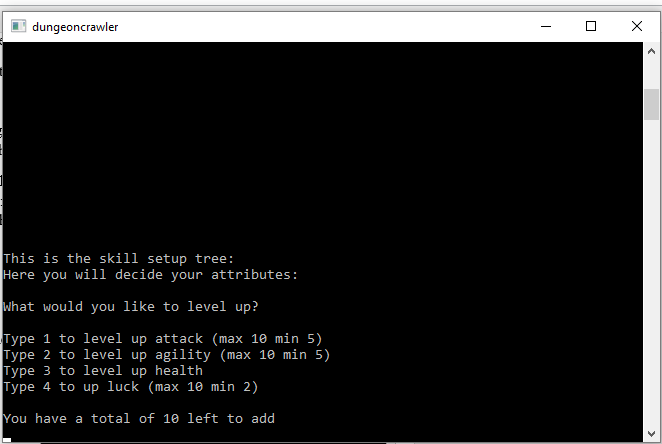
* Number of Variables: 58

1. **Table of concepts:**

*Disclaimer: The location will lead to one example of the concept being used. The concept may be used more frequently throughout the code.*

|  |  |
| --- | --- |
| Constructs | |
| Do while loop |  |
| While loop |  |
| For loop |  |
| if – else statements |  |
| Turnery Operator |  |
| Switch Case |  |
| If statements (in singular form) |  |
| Primitive Data Types | |
| Short |  |
| Int |  |
| Char |  |
| String |  |
| Float |  |
| Booleans |  |
| File-streaming | |
| ifstream |  |
| oftream |  |
| .open(“”) |  |
| .close() |  |
| .clear() |  |
| in |  |
| out |  |
| Math Function and Random Number Function | |
| sqrt( ) (<cmath>) |  |
| rand() |  |
| srand() | Line 27 |
| Time Function | |
| time() | Line 27 |
| Formatting | |
| setw() |  |
| setprecession() |  |
| fixed |  |
| Input formating | |
| getline |  |
| cin.ignore |  |
| cin.clear |  |

1. **Code Break Up:**



1. **Skill Set Up:**

As I previously mentioned, after the title screen the user sets up their skills. A deeper look, into the process will now be discussed.

The user will enter a number (1 to 4) that corresponds to the skill they would like to add points to. A switch case to check what skill was selected.

The range

Then, they decide how many points they will add to the selected skill. There is a do-while loop in place here, to ensure that the amount entered does not exceed the number of points available. The points are then added to the skill, and removed from the spendable points. Then one of three conditions could execute.

The first, the number of points in the skill fit into the range and no changes are done.

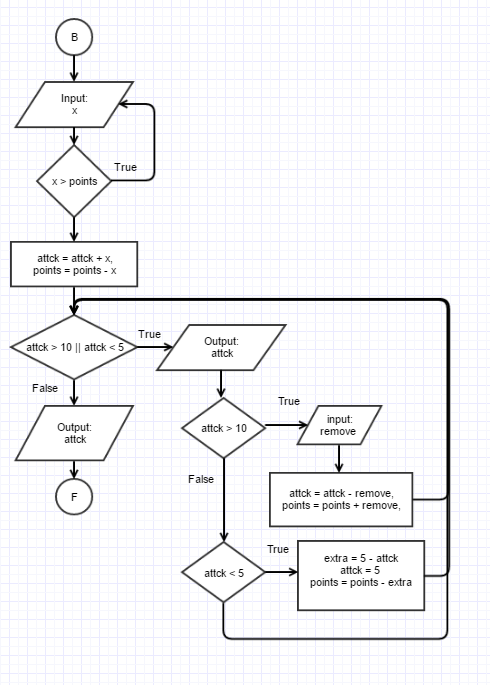
The second, there are too many points in the skill. The program will ask they user how many points they would like to remove. The removed point will then be added back to the spendable points.

The third, there are too few points in the skill. This will only happen when the user removes too many points. The skill will then just reset itself to its minimum amount. All the extra points in the spendable points will be cleared.

The program will continue to switch between condition 2 and 3 until the first condition is met. If else statements are used to decide which condition to execute. This will happen to all the skills except for the “Health” skill.

The “Health” skill has no range, so the number of points added by the user will be added to *health*.

As you can see, if else statements and do while loops are used extensively.



**Pseudo code:**

Flowchart if the player chooses to upgrade the Attack skill.

user inputs number of points to add;  
switch (input){  
case 1,2,4: (1, 2, 4 are individual cases)  
 Points are added to the selected skill.  
 Points are removed from spendable   
 points.  
 while points (are not in range){  
 if (skill points > range) {  
 Ask user to remove points   
 from the skill, and correct the  
 amount of spendable points.  
 }  
 else if (skill points < range){  
 Store the extra points then  
 reset the skill and remove the   
 extra points from the total   
 spendable points.

False

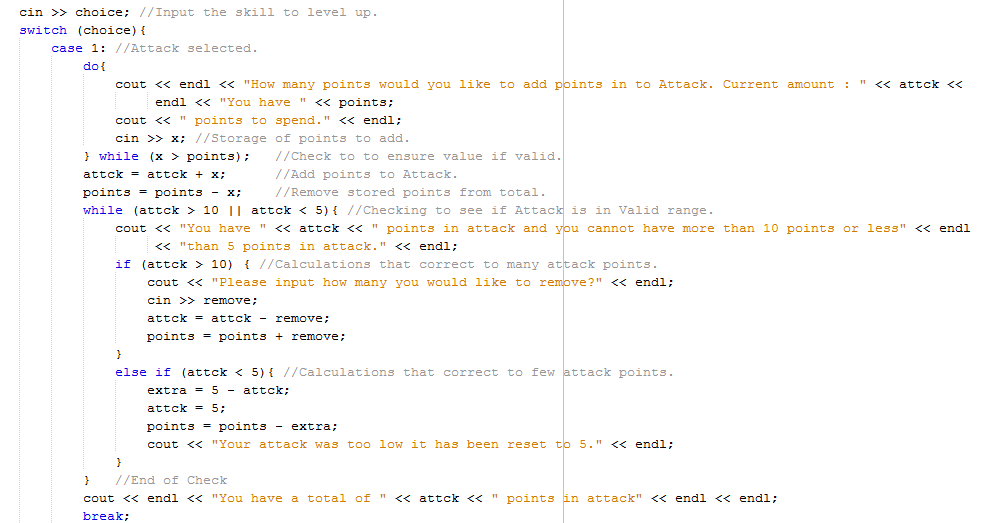
}

}

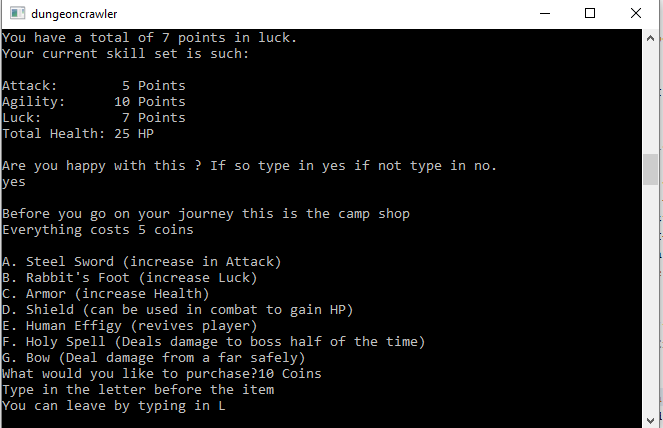
case 3:  
 Add the points health.

}

**Sample Code (if the user chooses to add points into Attack):**



1. **Shop:**



There are two shops located in the game. One at the very start of the game and one just before the final boss battle.

Here, the player can use the coins they collected throughout the game to purchase items. As I previously mentioned there are three types of items in *Dungeon Crawler*.

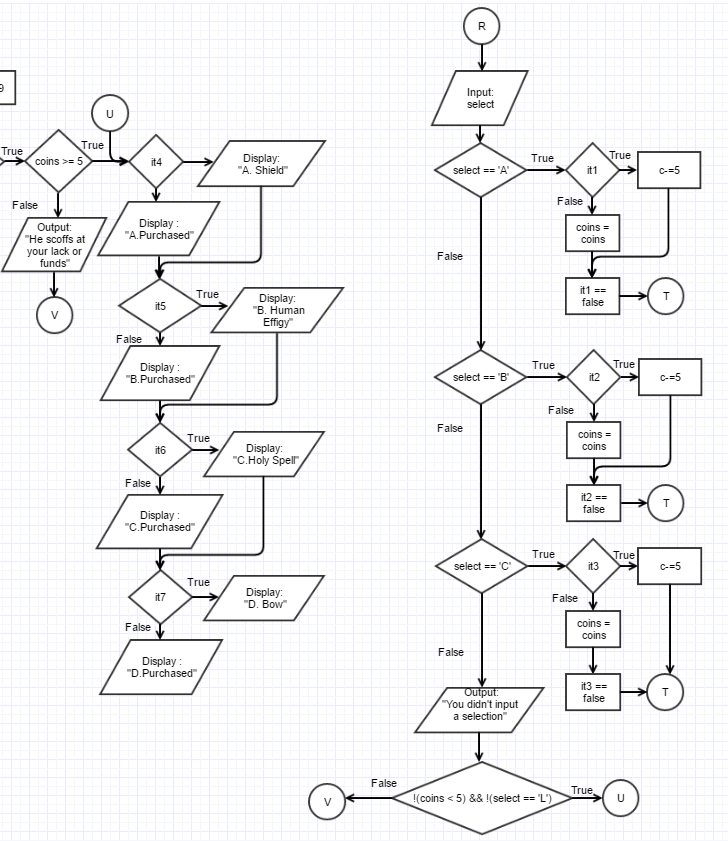
In the first category, the skill boosters, there are three items. These items may only be purchased in the first store. The items in the other two categories can be purchased in both stores.

To make the store function, I took advantage of the Boolean data type. The seven items are stored in the program as Booleans (*it1* to *it7*). They all start off as true, ex. it1 = true. If the item is true, the store will present the items name. Else if the item is false, the store will display purchased.

The user chooses an item to purchase via input of the letter preceding the items name. I then used a switch case to check what the user input and determine what the user purchased. If the user typed in A, for example, the switch case would recognize case A being selected. The Boolean value for the item would then change from true to false. Signaling the program that the item has been purchased and coins should be removed from the player’s wallet.

In the case that the user tries to purchase an item that has already been bought. The program will not remove any coins from the play’s wallet. I accomplished this check using a ternary operator. The player may also leave the store at any time by typing in “L”.

Coins which are the currency of the game are looted after killing monsters. (How many are looted will be discussed later)



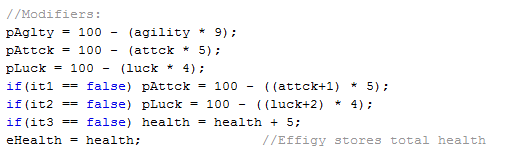
**Flow chart for (store 2):**

1. **Modifiers and Items**

Modifiers are used throughout the game to calculate success rates of certain actions. Modifiers are integers in between 1 and 100; effectively changing them in to simple percent chance numbers. *pAglty* affects dodge chance, *pAttck* affects critical hit chance, and *pLuck* affects loot amount and sneak chance. (all of these will be discussed later)

Modifiers are calculated after the player has set up their skills and bought their items. There is a total of three modifier calculations. They are all based on the number of points allocated in each skill. However, if the player chooses to buy category 1 items (skill boosters), the 2 of these calculations change slightly.

The skills have certain limits to points to prevent the player from having too low or too high modifier values, this can cause them to be successful every time or never be successful in their selected action. As seen below each modifier is being multiplied by specific



Same calculations but the user bought category 1 items, so there are some extra points added.

Standard calculations

There are 2 modifiers that do not affect success rate. The first eHealth modifier stores the player’s health (health variable). This modifier will revive the player upon death, however, it only activates if the player purchased the “Human effigy”(it5); which is a category 2 item (the revive item)*.* (I will discuss this again later). The second is the health modifier that only actives on the purchase of “Armor”; this will add an extra 5 points to the player’s total health.

The category 3 items (items that can be used in battle) don’t effect modifier calculations.

The “Shield”item(*it4*) will add 3 points into the player’s health.

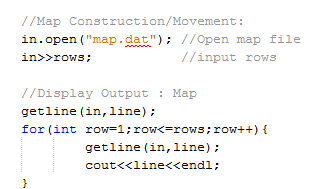
The “Holy Spell”item (*it6*) will subtract ten from the health from the boss (*bHealth* variable); there is a 50 percent success rate, however.

The “Bow” item (*it7*) will subtract three from the boss’s health (*bHealth*) or two from a monster’s health (*mHealth*).

How these items are activated will be explained later.

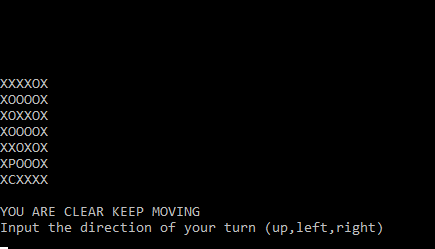
The program identifies usable items based on their Boolean value, remember each item is stores as a Boolean value in the program. If value of an item is *false*, then it can or in some cases will be used. This can be seen in the screen shot up above.

1. **Movement and Map**

The map in this game is a simple grid made of X’s, O’s, and a C. X’s represent walls and O’s represent open spaces. The player types in the direction of their move. They may time in up, left, or right; thereby forcing the player to move continually forward. Every time the player moves one cell, the O is replaced with a P.

This is achieved by using file streaming to manipulate the data in a file. First, the original map is constructed; this map is stored in the “map.dat” file. The parameters (rows and columns) of the map are present in the file itself. The rows are stored in the variable *rows*, and rows (the variable) is then used in combination with a for loop, getline, and the string variable *line* to output the map. (the pseudo code representation of this will be down below).

Movement is achieved by simple math. Each character on the map has their individual (x,y) like coordinate. Therefore, the movement commands entered by the player will change either x or y. In the code *prow* is the y coordinate and *pcol* is the x coordinate. Based on those to values the program will place a P on the map. The player starts off on the C on the map which has coordinates (*prow* = 7, *pcol* = 1). Now every turn the player has a change to change one of those variables.



If the player inputs “up” as their movement then *prow* is subtracted by one. (6,1)

Else if the player inputs “right” then *pcols* is added by one. (7,2)

Else if the player inputs “left” then *pcols* is subtracted by one. (7,0)

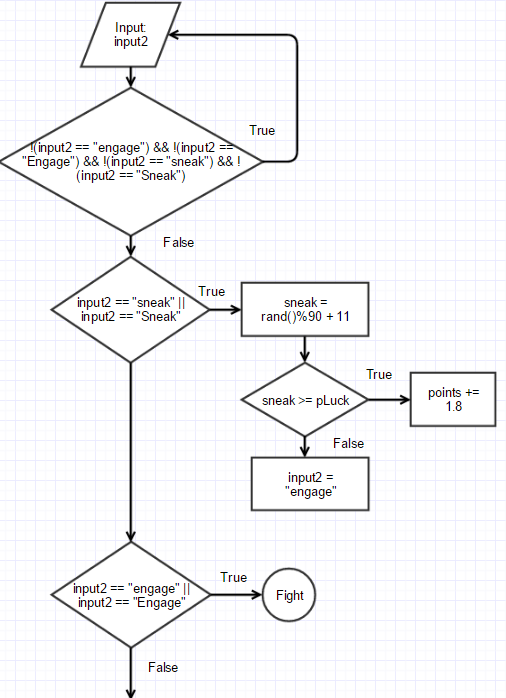
The new map is then constructed. This is not the same map as “map.dat” it is a copy. This copy is saved as “play.dat”. The construction process is the same as above; however, this time there are if else statements in the for loop. These will check the coordinates the player has. If the character with those coordinates is an X then the math done previously is reversed and the player is pushed back to where they were. Else if it’s an O then the O will be replaced with a P.

**Pseudo Code for the construction of map.dat:**

in.open “map.dat”  
in >> rows  
read in the first row into *line*  
for (int row = 1; row <- rows; row++){  
 read into *line*  
 output *line* and endl  
}

1. **Spawn and decision to engage**

There is large do while loop that allows the player to have multiple turns till the end of the game. This do while loop starts just before the player can input their movement option. This large loop ends when the player dies, or defeats the boss. This allows the player to move freely around the map fighting monsters until they die, or decide to fight the boss.



At the start of each turn the player move one unit, then the game will generate a random number, using the random number function, between 1 and 10 in the variable *spawn*. If *spawn* generates a number less than or equal to 6 a monster will spawn; thus, effectively creating a spawn chance of 60 percent.

Once the monster is spawns the game will give the user the option to sneak past the monster or engage it out right. The player must type in what they want to do.

If the player inputs “Sneak” or “sneak” into *input2*, the game will generate a random number in the variable *sneak*. Then, *sneak* is checked against *pLuck* (the modifier from before). If sneak is greater than or equal *pLuck*, the player sneaks by. Else, the sneak attempts fails and *input2* is set to equal “engage”. Because *input2* became “engage” the next conditional will activate.

If the user inputs “engage” or “Engage” into *input2*, the game will start the battle sequence of the turn.

**Pseudo code:**

Player inputs their move into *input2*.  
 if (*input2* == "engage" || *input2* == "Engage") {  
 Generate a random number in *sneak* if (*sneak >= pLuck)* {display that the player snuck by}  
 else {change the input to “engage”}  
 }  
 if (input2 == "Engage" || input2 == "engage"{  
 Battle sequence start (will be discussed later)  
 }

1. **Battles:**

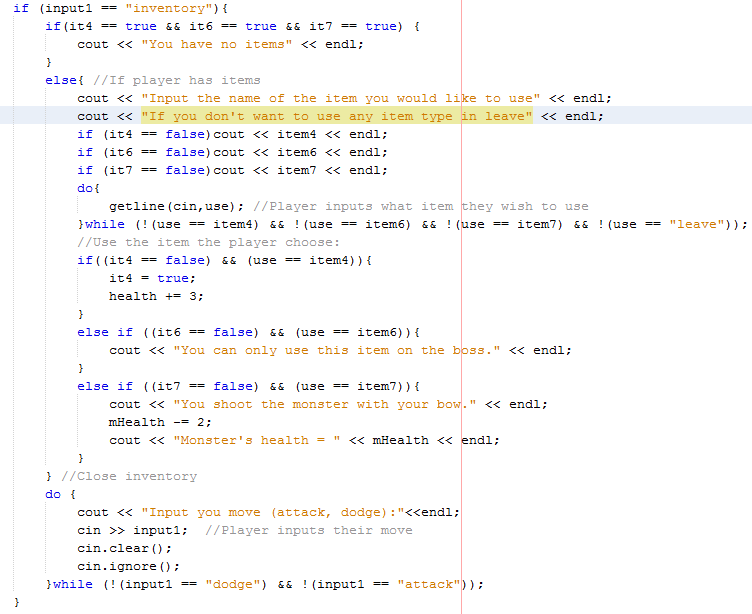
There are two type of battles in *Dungeon Crawler*, verses monsters or verses the boss. Monster battles are frequent throughout the game and the boss battle is at the very end. Since both battle types are technically the same, explain the differences between boss battle and the monster battle at the end.

The easiest way to break down this portion of code is to explain it in 2 parts. 1) Turn Moves, 2) Move interactions.

1. **Turn moves**:

There are three types of moves a player can make. The attack move, the dodge move, or the inventory move. To perform any of these moves, the player must type in “attack”, “dodge”, or “inventory”.

If the user typed in “inventory”, the program will display the items that the player bought (these are category 2 items). Items whose Boolean value is false will be displayed. The user can then activate any item by typing in its name, exactly. This part uses getline. The functions of each item were previously mentioned. After this process, the user can still choose either of the next two options.



If the user inputs “attack”, the program will set variable *pMove* equal to 1.

If the user inputs “dodge”, one of two things could happen. A random number is generated in variable *dodge* and is checked again *pAglty*. If *dodge >= pAglty*, then *pMove=0*. Else, *pMove=0.* This translates to: if *dodge >= pAglty* then the dodge is successful. Else, the dodge was not successful and the player must attack.

There are two enemy turn moves. Attack or dazed. To determine the enemy’s move, a random number is generated, between 1 and 10, in variable *monMove*. Then, *monMove* is divided by four and the remainder is used as its move, *mMove = monMove%4*. This will result in four different remainder values: 0,1,2,3.

If *mMove* = 1,2,3, the monster’s move is an attack. Else the monster’s move is dazed.

1. **Move Interactions:**

Based on the numbers of *pMove* and *mMove*, three occurrences are possible:

If *mMove* = 1, 2, or 3 and *pMove* = 1, both *health*, player’s health,and *mHealth*, enemy’s health, are reduced. The player loses 1 point of health, *health--*, and the monster also loses 1 point of health, *mHealth--*. However, here the Attack modifier, *pAttck*, comes into play. A random number is generated in variable *crit*. If *crit >= pAttck*, then the monster loose three points of health instead of one, *mHealth -= 3*.

If *mMove* = 0 and *pMove* = 0 or 1, only the monster’s health will be reduced. The monster loses one point of health, *mHealth--*. Again, the Attack modifier has the same effect here, as above.

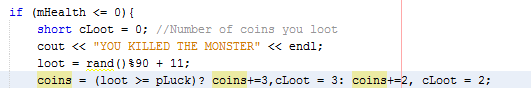
If *mMove* = 1, 2, or 3 and *pMove* = 0, the player dodged the initial attack. Neither *mHealth* and *health* are effected. However, now the player is given the option to counter attack or just start the next turn in the battle. The player can choose to attack by typing in “yes” or “Yes”. If the player chose to attack, one of three things could happen; based on two randomly generated numbers, *cAtt* and *cMAtt*.

If *cAtt <= 40 && cMAtt >= 50,* this means the counter attack failed and the player took damage. (*health*-=2).

If *cAtt <= 40 && cMAtt <= 50*, this means the counter attack failed and the player too no damage.

Else, the counter attack succeeded and the monster took damage. (*mHealth* -= 3)

After killing a monster, *mHealth <= 0*, the player will pick up coins. The standard is 2 coins, however, there is a chance that the player can pick up 4 coins.



The “turns” of a battle sequence is created with a large do while loop the starts just before the user inputs their move and only ends if *health <= 0* or *mHealth <= 0*.

**Differences between the boss battle and the monster battle:**

The coding of the boss battle is the same as the monster battle. However, there are three differences.

First, the player does much more damage per hit, and takes more damage per hit. Example, if the player hit a critical and the monster hits the player, *bHealth -= 5* and *health -=3*.

Second, how the “turns” are constructed. When fighting the boss the player only has 8 turns to beat the boss. I used a for loop with three test expressions. First, the counter *turns<=8*, *turns* starts at 1. The second, is *bHealth <= 0*, the boss was defeated. The third, is *health <= 0*, the player died. If the player ran out of turns (*turns > 8)* the player dies, *health = 0*.



Third, when the boss fight happens. Unlike the monster fights that may or may not happen every turn, the boss fight will happen every time. The weather or not a monster battle happens is based on the *spawn* variable; while the boss battle will always happen when the player reaches a certain point on the map. Using the coordinate like system discussed before, I was about to “place” the boss on (prow = 1, pcol = 4). When the player’s prow and pcol equals the boss’s prow and pcol, the battle starts. The player must fight this boss they cannot “sneak” pass.



1. **Scoreboard**

The game end when the player “breaks” out of the very large do while loop I discussed earlier (in part e). Basically, the loop will not stop unless one of these conditions becomes false:  
(*health > 0 && bHealth > 0*).

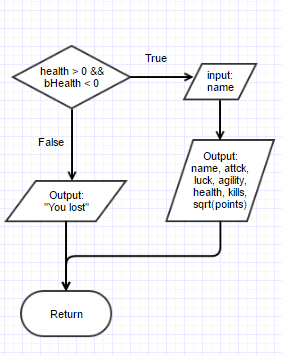
Once the loop ends, one of two things will happen. If the loop ends because the boss was slain, *bHealth <= 0,* the player is presented with the scoreboard. Else, the loop ended because the player died, *health <= 0,* no score board is presented.

The score board itself is simple. It will first ask the player for their name, then it will display: the player’s skill set-up, the players total number of monster kills, and finally their final score.

Score is calculated based on the number of points collected throughout the game. Killing the boss is worth 100 points, killing a monster is worth 8.3 points, hitting an enemy is worth 1 point, hitting an enemy with a critical hit is worth 2 points, a counter attack is worth 1.5 points, a successful dodge is worth 0.3 points, and a successful sneak is worth 1.8 points. These points are added up and then score is calculated by taking the square root of that number, *sqrt(points)*.

**Pseudo code: Flowchart:**

if (*health > 0 && bHealth < 0*){  
 Player inputs their name into *name*  
 Display *name* and “ is the winner.”  
 Display the player’s skills  
 Display how many monsters killed  
 Display the player’s score  
 }  
 else {display “You lost”}



1. **Pseudo Code representation of the game:**
2. **Project 2 plans:**

I am very happy with the result of this game. All the parts are working together nicely, and even thought the gameplay is simple it has some dynamics. There plenty of room for improvement and for more features. If I revisit this game for my second project I will perform the following:

1. Break up main into multiple functions.
2. Change the map into an array.
3. Add different type of attacks.
4. Add more enemy types.
5. Increase the map size and make it more of a maze.
6. Add more features based on the concepts that will be taught.

I greatly enjoyed writing the code for the game. I tried to check off all the concepts that we covered in class. This project (including the write up and flowchart) took around a week to complete. I hope to not only add features to Project 2 but also polish the game.