Project 2

Title:

Dungeon Crawler Version 2

Student: Hassan Farhat

Submitted to Professor Mark Lehr

Due Date: 13th December, 2016

Course and Section:

CSC-5, 48102

1. **Introduction:**

Dungeon crawler was created to honor previous Dungeon crawling games. Dungeon Crawler Version 2 builds on the concepts of its predecessor. It follows the same goal; traverse a dangerous dungeon and defeat the boss at its end.

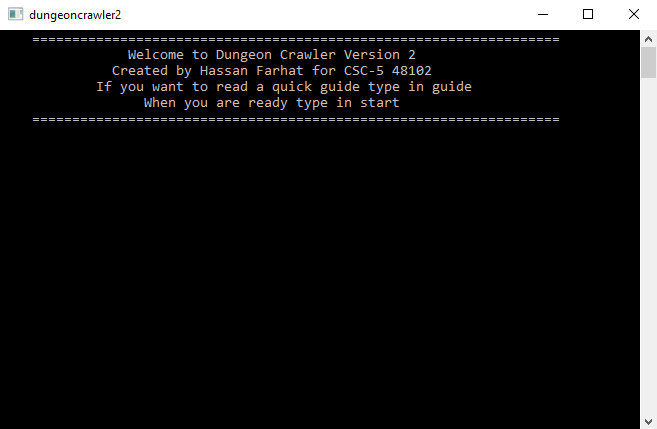
The game can be thought of in 4 parts:

Character Creation  
 Shop  
 Movement and map control  
 Battle Sequences

Each one of these existed in the previous game however they all received some form of overhauls. In summery; all of them received functions, many received arrays, and the battle and map were completely rewritten.

The purpose of these changes was to express the concepts of the class and to make the game more efficient and cut away bloated code. The result allowed me to add more features to the game that both made it more interesting.

A deeper look into the changes will be examined when I discus the important parts of the code.



1. **Gameplay:**

The overall game-play of the game is very similar to its predecessor.

The game starts with an intro screen. The player may start the game directly by typing in “start”, or they can type in guide to read a quick guide about how the new class system works.

After that, the player will be asked to select their class. (The importance will be discussed later). Then the player will decide how to distribute points into their individual skills; essentially determining what their character is best at.

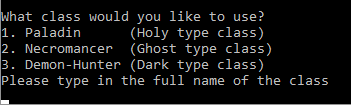
The player is then presented with the first shop in which they can buy items. There are 7 to choose from, each will give the player certain advantages.

When the player is done with all that set up its time for the main section of the game to start. The player can now move, by typing in the direction they want. They player must make their way towards the “B” on the map and fight the boss to end the game. If the player encounters a monster along the way they must fight it. In the fight the player can chose to use an item from their inventory or attack or dodge. Once the monster has died the player may move on.

There are 2 ways the game will end. The player dies or the boss is defeated. If the boss is defeated the player will be presented a score screen. If the player dies a message “YOU DIED WILL” will appear.

1. **Code Content:**
2. **Code Break Up:**
   1. **Class selection:**

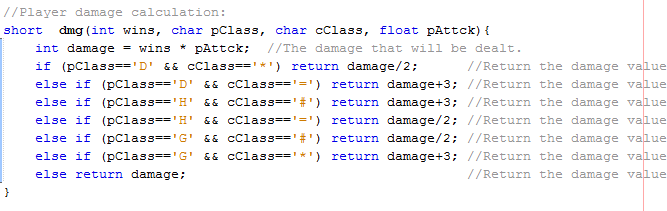
At the start of the game the player must choose their class. There are three to choose from:



1. Dark
2. Holy
3. Ghost

The player will be able to select their class by inputting in the name of the class. Paladin for Holy, Necromancer for Ghost and Demon-Hunter for Dark

They follow a rock-paper-scissors method when inter acting with one another. Dark is weak against Holy but strong against Ghost. Ghost is weak against Dark but string again Holy. And lastly, Holy is strong against Dark but weak against Ghost. If the player where to encounter a monster of a class that counters them. They will both deal less damage and take more damage from the monster, and vis versa for the monster. If you and the monster are of the same class, then you will have no advantage over them and there will be no damage penalty or buff for either of you.



* 1. **Skill Setup:**

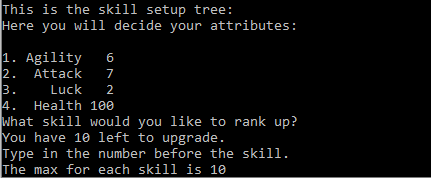
The skill setup portion of the code is mainly untouched from the previous game. The main differences are the starting numbers, and the fact that this portion has been broken into 3 functions. There is a big function which has the switch case. There is the display function which will print out the array. There is an over loaded calculation function one for health and one for the rest. (reason for separation will be discussed farther down).

The functions are:

void skill(int skills[], string skNames[], int points, int min, short n, short choice)  
void skTree (int skills[], string skNames[], short n)  
void skillcal (int skills[],string skNames[],int min,short choice,int &points)  
void skillcal (int skills[], string skNames[], short choice, int &points)

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

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 the 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.

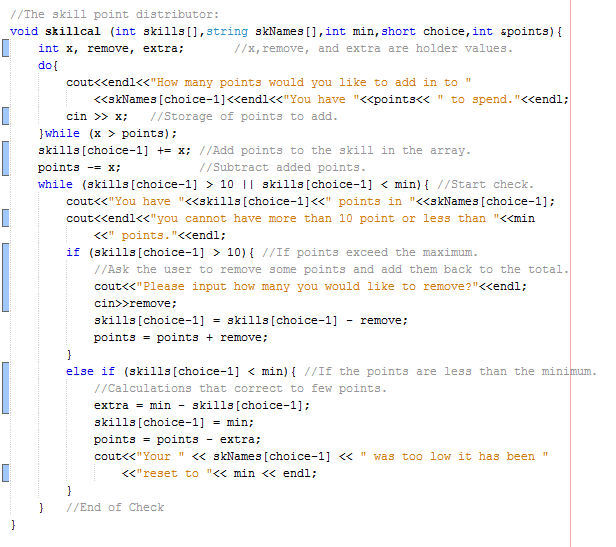
**Sample code of the calculation function:**

Pseudo code for the process:  
User inputs the skill the would like to add too  
and the 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.

}

}

case 3:  
 Add the points health.  
}



* 1. **Shop:**

The shop in this game follows the exact same logic as previous. However, there are changes. One is that instead of each of the 7 items having its own Boolean value there is one array. The Boolean array is parallel with a String array that contains the name of each item.

The shop exists in its own self-contained function. The player’s coins and health are passed though by reference, along with the Boolean and the String array, and the utilized size.

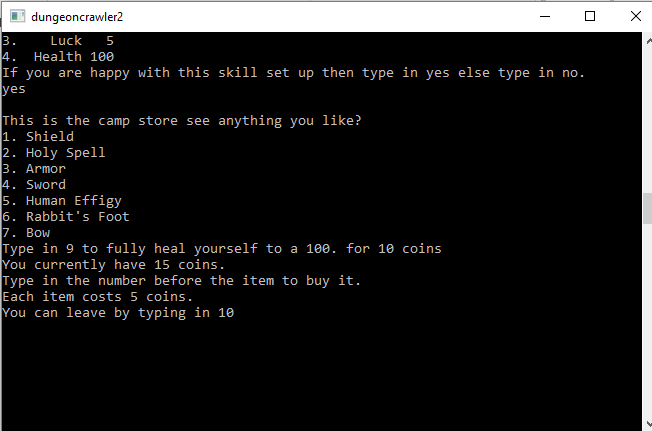
There are two shops located in the game. One at the very start of the game and one just before the final boss battle. The player starts with 15 coins. Here, the player can use the coins they collected throughout the game to purchase items.

To make the store function, I took advantage of the Boolean data type. The seven items are stored in the program as a Boolean array name *it*. All the stored values start off as true, ex. it[0] = true. If the it[n] is true, the store will present the item’s name. Else if the it[n] is false, the store will display purchased.

The user chooses an item to purchase via input of the number preceding the item’s name. I then used a switch case to check what the user input and determine what the user purchased. 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.

There is another items the player may buy which is the heal to 100. This “item” is not stored in either array. This item costs 10 coins.

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 “10”.



* 1. **Modifiers and Items:**

There are 7 items *Dungeon Crawler Version 2*:

* 1. Shield: Adds to health variable when they user uses their inventory.
  2. Holy Spell: Subtracts from the boss’s health upon user use.
  3. Armor: Immediately adds to the user’s health.
  4. Sword: Immediately adds to the pAttck variable.
  5. Human Effigy: Stores user’s health and can revive user, health = ehealth.
  6. Rabbit’s Foot: Immediately adds to pLuck variable.
  7. Bow: Subtracts from any enemy’s health upon user use.

How “upon user use” work will be explained later.

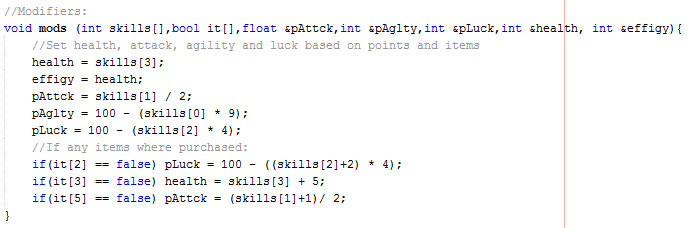
The use and implementation of modifier is the same as that of last game. Primary differences is the use of a modifier function and the removal of sneak from the game.

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 and score amounts. (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 a specific number.

These are the calculations for the modifiers if the user purchased items 3,4, or 6.



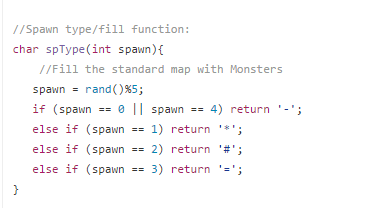
These are the standard modifier calculations for the modifiers.

Passing in the skill values, the items and all the modifier values.

* 1. **Map, Movement, and Spawn:**

This part of the game received a full over haul. Nothing from the previous game remained except for one small part. In the previous game the map and the movement was done through file I/O. However, this time it is done with arrays. Two two dimensional parallel arrays in fact. The first array is *map[n][m]* and the second is *pmap[n][m]*.

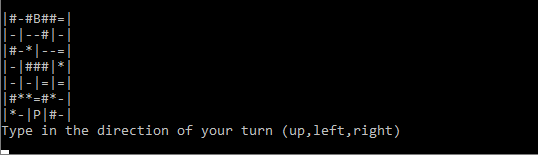
The movement in the game is accomplished by user input. The user can input one of three movement options. Up, right, and left. Once they put in their movement the “P” on the map will move one position in the selected direction.



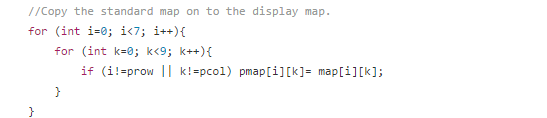
I will full explain the logic behind how this works.

First *map[n][m]* is filled in a function called crtmap, with “B”, “C”, “-“, “|”, in their standard positions. Then for every “- “, a function spType is called, which will replace the dash with: -, \*, #, or =, based on a random number modded by 4. Each one of these represents a monster.

Second the *map[n][m]* is then copied onto *pmap[n][m]* using a for loop. All but for one single component of the *pmap[n][m]* will be replaced. The one that will remain is “P” and its location is store in the following location *pmap[prows][pcols].* The two variables inside the brackets are the location coordinates of the player. They start at prow=6, pcol=4, each time the player input a movement, the values are updated as follows: up subtracts one to prow, right adds one from pcol and left subtracts one from pcol. (Below shows the copy loop and the movements). If the player happens to try and pass through a | it will not work and reset all the prow and pcol changes. This map building process will happen every time they user inputs a move; there by creating a “dynamic” changing map with every turn.



The copying code sample:



PseudoCode:

For (i=0 till i=6)  
 for (k=0 till k=6)  
 So long as i and k are not prow and pcol  
 pmap[i][k] = map [i][k]

The movement code sample:

