Dylan Markovic

CS162 Final

12.6.2016

For the final, several requirements needed to be fulfilled for this text based game. I needed to create an abstract room class with 4 pointer variables to other rooms. I needed at least 6 instances of 3 derived classes derived from the room class. The player needs a goal and the game needs a theme. Also, the player needs some kind of container to carry a finite number of items. Also, at least one of these items needs to be necessary to win the game. The controls need to not be free form, and some kind of time limit must exist as well. Also, the player must interact with the rooms, and not just collect things. Lastly, a menu must be present to inform the user of the games goal.

For the final, I decided to create a dungeon crawler style game.

Please Note, due to the size of this assignment, I have grouped the class designs and testing together. This should make checking my testing methods easier.

STORY:

The plot is as follows: a member of the rogue's village has stolen snake eggs from a Sacred Serpent Tower. The Snake God was clearly angered by the harm to his snakes, and has cursed the village. The rogue, characterized with the @ char, will travel through the Serpent Tower in search of a cure. However, the rogue is a villager as well, so each step he makes takes a piece of his life away.

The rogue starts outside of the temple. He cannot enter the door until he has a torch because the Tower is very dark. After he has the torch, is able to enter and exit the tower as he pleases.

The first room inside the tower, called the Snake Room, is full of snakes. Snakes are characterized with S characters. The snakes travel back and forth in this room. Also, there are three locked doorways and four oil pits. Once the rogue lights all of the oil pits with his torch, the snakes get really scared. In an attempt to make the rogue leave before he starts more fires, the snakes use their snake-magic and open the locked doors. The rogue is now free to enter and exit any room attached to the Snake Room.

Above the Snake Room is the Wind Room. When the rogue enters, he cannot get out of the doorway because strong winds block his path. Wind is characterized with the ~ character. A voice call out and says the rogue must have the Emerald and Ruby Snake Eyes to access this area.

To the right of the Snake Room is the Rattle Snake Room. The rattle snake room is obviously full of rattle snakes. Rattle snakes are characterized with & characters. The rogue will find some bongos, in this room. When he grabs them and starts drumming, the rattle snakes get annoyed and give him the Ruby Snake Eye so that the rogue will leave them alone. The rogue now has 1 of 2 items he needs to unlock the final room.

To the left of the Snake Room is the Cobra Room. Cobras are characterized with 6 characters and clearly roam the room. The rogue can find a flute in this room. When he grabs and plays the flute, he charms the cobras and they give him the Emerald Snake Eye to thank him. Finally, the rogue can see what lies past the Wind Room.

Upon entering the Wind Room, the winds have separated. There is now a direct line to the door at the top of the room. A voice tells the Rogue he is now worthy to enter because he has the two Snake Eyes.

After the rogue walks past the wind columns, he finds himself in a room with the Snake God. The Snake God knows he has obtained the snake eyes without harming any serpents, and gifts the rogue with the cure to the curse. The rogue is informed to exit the Snake Temple and go back to his village to rid his people of the curse.

DESIGN:

This game clearly displays all the requirements of the game. The rooms are linked to each other using pointers. There are 6 derived classes from the abstract room class. These are the entryRoom, snakeRoom, cobraRoom, rattleRoom, windRoom, and snakeGodRoom. The overall goal is presented to the user with a menu at the start and reminded once he has the Snake God's Cure item. Messages above the room will always give a blurb of what is happening, such as, "OH NO, COBRAS(6)!!!! You better grab and play that Flute!" or , "The Cobras dig your Tune, They give you the Emerald Snake Eye!" so assist the user in what to do. The user needs a torch to enter the temple, he needs to light all the oil pits to unlock doors, he needs the Emerald and Ruby eye to move the wind, and he needs the Snake God's Cure to proceed out of the temple to win the game.

THE PAWN CLASS:

The pawn class is an abstract class that has a char variable named body, and integer for both the x and y position on the room maps, an integer for health, a boolean for direction, a string for a name. It has a constructor and a pure virtual destructor. It has two functions setX and setY that take integer parameters to set the xPos and yPos, alongside constant getX and getY to return them. A constant getBody function returns the body, and the constant getHealth Function returns the Health. A setHealth changes the value of the health, and a setBody changes the value of the body. The function setDirection will change the direction bool, and the constant getDirection returns it. getName is a constant function that returns the name.

PAWN'S DERIVED CLASSES:

The pawn class has many derived classes. The bongo, cobra, firePit, flute, rattlesnake, snake, torch, wall, and wind are all derived classes. They vary only in name and body. The different snakes and the wind are controlled by the rooms, and will change xPos and yPos accordingly. Also, the rooms will cause the firePits to change their body from O to * once the player lights them on fire.

The other derived class is the player class. The player class has a health of 800, which allows the user to make 800 steps before dying. The player also has a string array that can hold the 6 items he needs to complete the game. Additionally, the player class has an addToInventory function which adds a string to the inventory, a checkInventory that returns true if the string parameter matches a string in the inventory array, and a printInventory function that prints the contents of the array to the screen. Also, the player has a decreaseHealth function, which decreases the player health by 1. There is a getHealth function that returns the health, and a kill function that sets the player health to 0.

TESTING THE PAWN CLASSES:

Testing of the pawn and its derived classes was very simple. The majority of the pawn functions are setters and getters. Testing of all of these are as simple as printing the getter function return to the screen, changing the value with the setter function and then using the getter once again to print to the screen. For the getDirection and setDirection, a similar testing approach occurred. The default direction is true. I tested these functions by having the program print true if the getDirection returned true, and false if the getDirection returned false. Using the getter and setter allowed me to print true, then false, then true, showing it worked correctly.

Testing the player class was very similar. I used printInventory before adding any string, than used a loop to add “thing” to the inventory six times and print the inventory after each time. The results were:

```
thing

thing thing

thing thing thing

Etc.
```

The getters and setters were also only needing the trivial print the getter, use the setter, and print the getter to insure they were behaving correctly. The checkInventory was tested by seeing if “thing” was in the inventory before and after the aforementioned loop, and printed false and true accordingly using if statements.

THE ROOM CLASS:

The room class has four room pointers top, bottom, left, and right. It has two integers rows and cols, a pawn ***map which becomes a dynamic 2d array of pawn pointers of dimension rows x cols, a player pointer called rogue, and a wall called wall1. The wall1 is used to limit the dynamic memory of the room and derived room classes. Instead of needing to create new walls for each position of the map at which I want to create a wall, I can assign the pointer the value &wall1. This way I can delete the map in the destructor with less effort.

Only three functions of the room class are not virtual. The constructor takes a player pointer as a parameter and assigns it to rogue. It dynamically allocates map into a rows x cols matrix. And it assigns the outer most elements of map the &wall1 value. The next non-virtual function is controlPlayer. Control player uses user input and switch statements to assign the rogue to different locations of the map matrix, and adjust the rogues xPos and Pos accordingly. This function checks to make sure the location the rogue is to move to is NULL, which is why the border of map is all walls, so insure no out of bounds. Also, control player allows the rogue to collect items, buy deleting the a pawn with an appropriate body, and adding the name it the rogue's inventory. moveEnemyis a very similar function. However it takes a pawn pointer as a parameter. It moves the pawn according to its direction value and its body value. S move left to right, 6 and & move up to down. The xPos and yPos values are changed accordingly using the setters. If the pawn pointer tries to move to a non NULL value, its bool is reversed and the pawn pointer begins to move in the opposite direction.

The following are all of the virtual functions, alongside descriptions of them:

virtual ~room() = 0;
The pure virtual destructor frees the dynamic 2d map array

virtual void printRoom() const;
The printRoom function calls the printMessage function,
calls rogue->printInventory, calls rogue->printHealth,
and finally prints the body values of the map elements.
If the element is NULL, a '.' Is printed.

virtual void moveCharacters();
TheMoveCharacters function calls controlPlayer.
In derived classes it may also call moveEnemy.

virtual room* getRight() const;
returns right pointer.

virtual room* getLeft() const;
returns left pointer.

virtual room* getTop() const;
returns top pointer.

virtual room* getBottom() const;
returns bottom pointer.

virtual void setRight(room*);
sets right pointer.

virtual void setLeft(room*);
    sets left pointer.

virtual void setTop(room*);
    sets top pointer.

virtual void setBottom(room*);
    sets bottom pointer

virtual void printMessage() const;
    prints “\nthis is the parent class message\n”
    In child classes it prints different messages depending on the inventory of the rogue and the state of the room.

virtual bool traverseTop();
    Does nothing in parent class.
    In child classes it returns true if the rogue is able to move to the top room.

virtual bool traverseBottom();
    Does nothing in parent class.
    In child classes it returns true if the rogue is able to move to the bottom room.

virtual bool traverseRight();
    Does nothing in parent class.
    In child classes it returns true if the rogue is able to move to the right room.

virtual bool traverseLeft();
    Does nothing in parent class.
    In child classes it returns true if the rogue is able to move to the left room.

virtual void setRogueTop();
    Does nothing in parent class
    In child class it places rogue in the appropriate spot after a room change and sets the rogue’s xPos and yPos accordingly.

virtual void setRogueBottom();
    Does nothing in parent class
    In child class it places rogue in the appropriate spot after a room change and sets the rogue’s xPos and yPos accordingly.

virtual void setRogueLeft();
    Does nothing in parent class
    In child class it places rogue in the appropriate spot after a room change and sets the rogue’s xPos and yPos accordingly.

virtual void setRogueRight();
    Does nothing in parent class

In child class it places rogue in the appropriate spot after a room change and sets the rogue's xPos and yPos accordingly.

THE ROOM'S DERIVED CLASSES:

The room's derived class functions are all summarized above. However, the rooms also have additional data members and variations of the map.

The entryRoom class a set of walls shaped like a U in the center of the map. This is to signify a door way. Since the entryRoom is where the game starts, it also places the rogue on the map. Also, the entryRoom has a torch data member which is also placed on the map.

The cobraRoom and rattleRoom are very similar. The cobra room has walls to create a doorway on the left of its map, and the rattleRoom has walls to create a door on the right of the map. The cobra room has a flute data member, and the rattleRoom has a bongo data member. Once the rogue adds flute to its inventory, the cobraRoom adds Emerald Eye to the rogue's inventory in the printMessage function. Likewise, after the rogue's inventory has Bongo, the rattleRoom adds Ruby Eye to the rogue's inventory in the printMessage Function.

The windRoom has walls to signify doors on the top and bottom of the map. It also has two wind arrays for the columns of wind blocking the room. The one array's position is changed once the rogue has the two eyes.

The snakeGodRoom is very simple. It has walls to keep the rogue in the very bottom of the room and no other changes to the data members. The significant difference is the print functions. The printRoom function displays a large text image of a snake and adds the "Snake God's Cure" to the inventory of the rogue. It only prints the bottom fourth of the map. The walls were placed to keep the rogue in a viewable location.

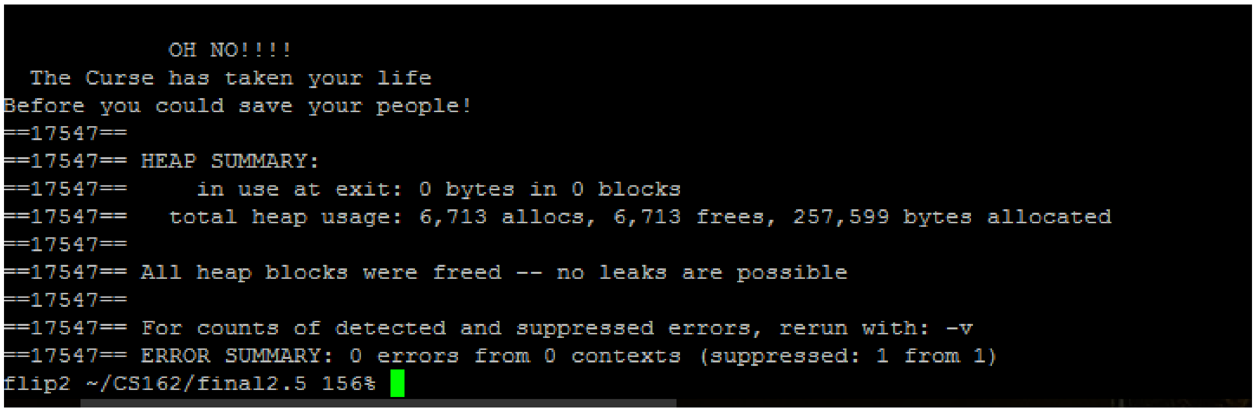
The snakeRoom has walls on every side of the map to indicate doors. It has an array of snakes and an array of firePits. Once the firePit body variables are changed by the rogue into *, the rogue is able to travel to the right, left and top pointers once he is in the doorways using the traverse functions.

The destructors of the derived classes free the dynamic memory allocated to the respective rooms. In the chance of the rogue dying before he is able to collect(and thereby destroy) the torch, bongo, or flute, these data members are destroyed as well by using an if statement alongside the checkInventory function of the rogue.

TESTING ROOM AND CHILD CLASSES:

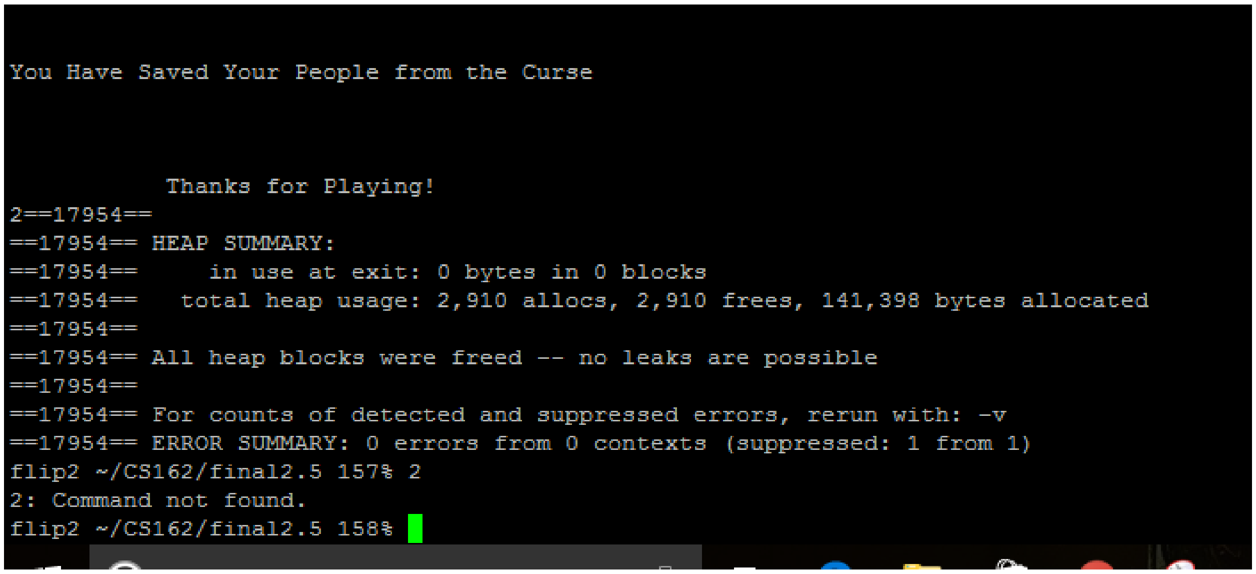
Testing Dynamic memory accomplished using valgrind. It was used while having the user win the game, and while having the user not collect any items at all. The reason for this is that collecting items frees that particular pawn from memory.

Upon losing and not collecting any items:



```
            OH NO!!!!
  The Curse has taken your life
Before you could save your people!
==17547==
==17547== HEAP SUMMARY:
==17547==     in use at exit: 0 bytes in 0 blocks
==17547==   total heap usage: 6,713 allocs, 6,713 frees, 257,599 bytes allocated
==17547==
==17547== All heap blocks were freed -- no leaks are possible
==17547==
==17547== For counts of detected and suppressed errors, rerun with: -v
==17547== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
flip2 ~/CS162/final2.5 156%
```

Upon beating the game:



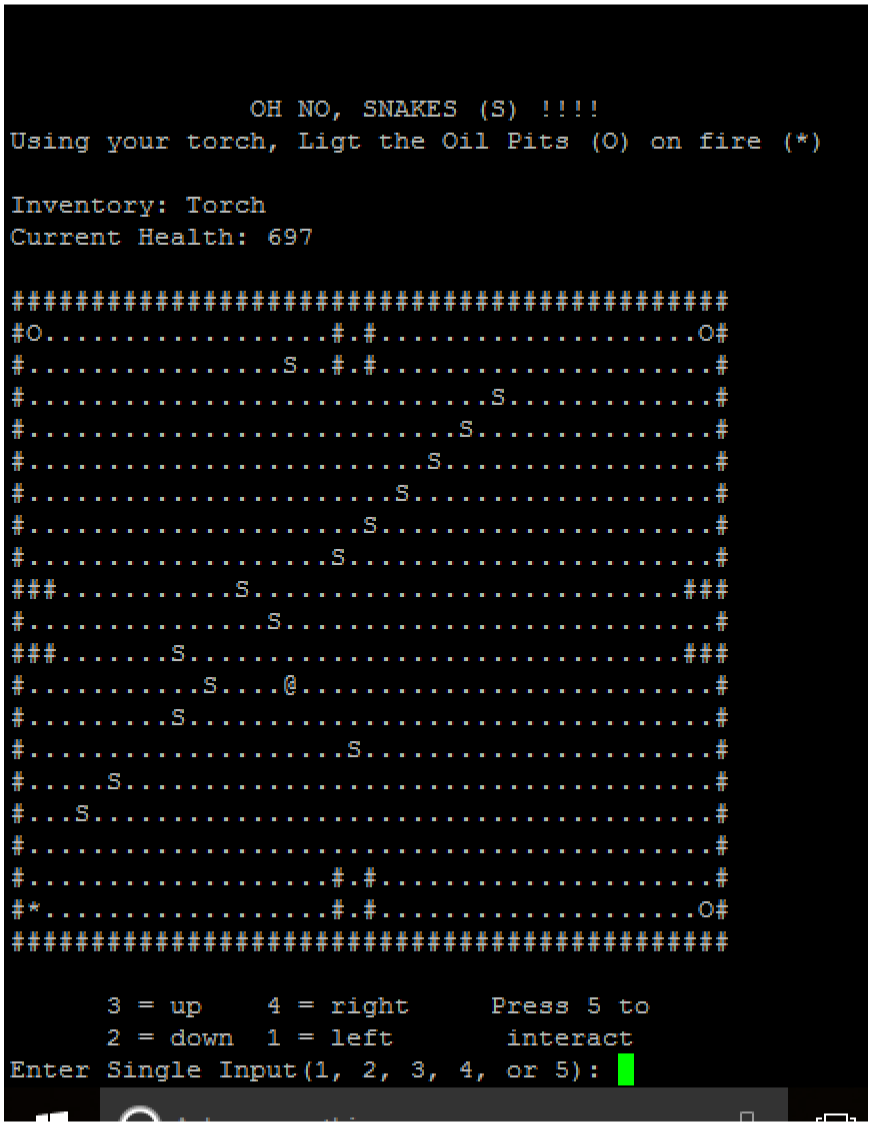
```
You Have Saved Your People from the Curse



           Thanks for Playing!
2==17954==
==17954== HEAP SUMMARY:
==17954==     in use at exit: 0 bytes in 0 blocks
==17954==   total heap usage: 2,910 allocs, 2,910 frees, 141,398 bytes allocated
==17954==
==17954== All heap blocks were freed -- no leaks are possible
==17954==
==17954== For counts of detected and suppressed errors, rerun with: -v
==17954== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
flip2 ~/CS162/final2.5 157% 2
2: Command not found.
flip2 ~/CS162/final2.5 158%
```

Testing the other functions visually was necessary. Making sure that snakes travels left to right, cobras and rattleSnakes traveling up and down was checked simply by playing the game repeatedly.

Wandering the snakeRoom after several turns:



```
               OH NO, SNAKES (S) !!!!
Using your torch, Ligt the Oil Pits (O) on fire (*)

Inventory: Torch
Current Health: 697

#############################################
#O..................#.#....................O#
#................S..#.#.....................#
#.............................S.............#
#...........................S...............#
#.........................S.................#
#.......................S...................#
#.....................S.....................#
#...................S.......................#
###...........S...........................###
#...............S...........................#
###.......S...............................###
#...........S....@..........................#
#.........S.................................#
#....................S......................#
#.....S.....................................#
#...S.......................................#
#...........................................#
#...................#.#.....................#
#*..................#.#....................O#
#############################################

      3 = up    4 = right     Press 5 to
      2 = down  1 = left       interact
Enter Single Input(1, 2, 3, 4, or 5):
```

Aside from the pawn’s “motion” and the dynamic memory allocation, the changing in messages upon the rogue playing the flute, or lighting the oil pits, or grabbing the bongos was all accomplished by playing the game. Making sure the winds blocked the rogue until the rogue had the two eyes was also checked via playing the game. Testing the snakeGodRoom print functions were similarly tested by playing the game.

By having the pawns move correctly, by having the rogue traverse through the rooms when he was placed correctly and held the appropriate items and by not having any memory leaks or core dumps, the program is functioning correctly.

REFLECTION:

This final project went relatively smoothly for me. The difficult parts of the assignment was deciding how to change rooms, and how to keep track of all of the dynamic memory allocations, and controlling the rogue.

Upon realizing I could treat the room changes very similarly to the way you traverse a list or a queue, it became very simple. The only difference was needing to apply logic gates to when the current room. were able to be switched to their top, bottom, right or left pointers.

Initially I had every wall element in the map arrays assigned to a new wall. This led to an immense mess of trying to free all of the dynamic memory. Luckily, I realized it didn’t matter how many wall elements I had. I only needed to make each map element that needed a wall value to point to the same wall element address. I decided wall wall1 would become an element of the parent class, and every wall element would be assigned its address. This made managing the memory as simple as freeing only the new enemies and key items of the derived classes.