Design:

Design documents are attached, and the description of how stuff works is below.

I had taken some time to think about designing this program before I fully read the description. Because of this, I came up with some ideas that I actually could not use due to design constraints, that told me specifically how I had to do things. After reading through the document, I came up with some better stuff. I decided to have not much in the main function, just stuff for creating the file name, and creating a new game for each level. After that the main simply loops until the level is completed, which it sees as a return of true from the game's play function. The game class is where most the actual program is. The constructor calls functions in the class that get the floor size, read in the floor, and get the starting position. All of these are boolean functions, and the game class constructor calls all of these and then if any of them return false it sets a boolean false and then when the main file calls play from the game object if that boolean is false it immediately returns that the floor is done. The Game function to get the floor size reads in the first line of the file. It then looks for a space in the string that it inputs, and everything before the string it converts to an integer and that becomes the x size, and then everything after the space is converted to an int and that becomes the y size. The Game function to read in the actual floor reads in the file character by character, and if the character equals one of the ones that are allowed in file then the function then it places it in the array. The last function that's called by the game constructor gets the start position for the player, and this will be used for resetting them. Finally the characters are created.
For creating the characters, the function loops through the floor array, and if it finds a 'S' then it creates a swan at those coordinates, and if it finds an 'E' then it creates a player there. The player and swan classes are both child classes of the actor class. The Actor class has some variables that it needs. These include the current x and y position, the next x and y position, and the character that was previously in the position the actor currently occupies. This is important because when the character moves to a new position, that char can be used to replace where it used to be. The move function for the swan class gets a random number and then creates new coordinates from the random number, then checks if its a valid move, and then moves there. The player object gets an input character from the user and then creates coordinates from it, or calls another function. It then checks if it's a valid move and then move.
After creating the game, the main function continuously loops through calling the play function until it returns true. The play function loops through the array of characters and calls each of their move function. The headers for the player and swan classes, are both in the character.h file. I did this because they are both similar, and children of the same class and they don't take up much room, and I think it's good to be able to compare them.

Reflection:

As of right now the following are things that work. File checking works, and the program will skip a file if it has any chars that aren't allowed in and go to the next file. Reading the file in works, and it is placed into the array correctly and is printed correctly as well. Creation of the characters

works and they are placed at their correct positions. The swans fully work, and move correctly. The player moves correctly, and picks up the keys and unlocks the doors with them correctly as well. The player can pick up the apples, but not do anything with them. The player can pick up a maximum of three apples and three keys. Exiting with the ladder works and also with the exit, they work exactly the same. The use function for exiting works correctly as well. The floor updates with every turn, and all the players move each turn.
Things that don't work right now include resetting the player when they come within one square of a swan, using the apples to be ignored by the swan for a certain amount of time, and cleaning up memory doesn't entirely work. One major problem I had was having the array of player and swans inside of the game class. needed to call functions that were in the game object, or access variables in the game object, I didn't know how to do this, like calling a function of a class from inside an object that was declared inside that class. I tried passing the game object all the way from the main through the game object itself into the player and swan objects, but that didn't work. In the end I decided to pass the array of the floor to the characters and swans as a reference so that they could edit it and it be updated in the game object.
Another major problem I had for a really long time was using multiple levels. When one level was complete, the program would segfault reading in the floor of the next level. I figured this was a problem with memory freeing, and to finally resolve it I just deleted the floor array and then the pointer to the game object and that works.

Test Plan:

To test the program, I first when ahead and tried using the example floor provided. I tested moving, entering bad key enters as well as bad movement positions to make sure it wouldn't move somewhere where it couldn't. This also tested to make sure that if a bad key was entered, it asked for a new one. I next tested picking up apples, and making sure a max of three could be picked up. I then tested doors, and keys. Making sure a max of three could be picked up and that they would unlock doors, and that it would remove a key when one was used. I tested just having swans a no player, and delaying each turn by two seconds, to observe their movement and make sure they moved correctly and they did. I tested having bad characters in the floor file by adding erroneous characters. Even though I didn't write in much memory freeing, I did run valgrind to see how much memory was lost. I tested using multiple floors by having multiple floor files, and using the ladder of the first one.

Test Results:

The only things that didn't work were the apples, memory freeing, and having multiple floors. Everything else works, and though the swans move very dumbly, they do move correctly.

update: mnow previous position
mnow next position
mnow character char

Char ident
next pos
prev pos

move

swan
create(pos)
move()
checkpos to player

classes
Actor
Swan
Game

Player
create
restart
check use

int's start pos
next pos
prev pos

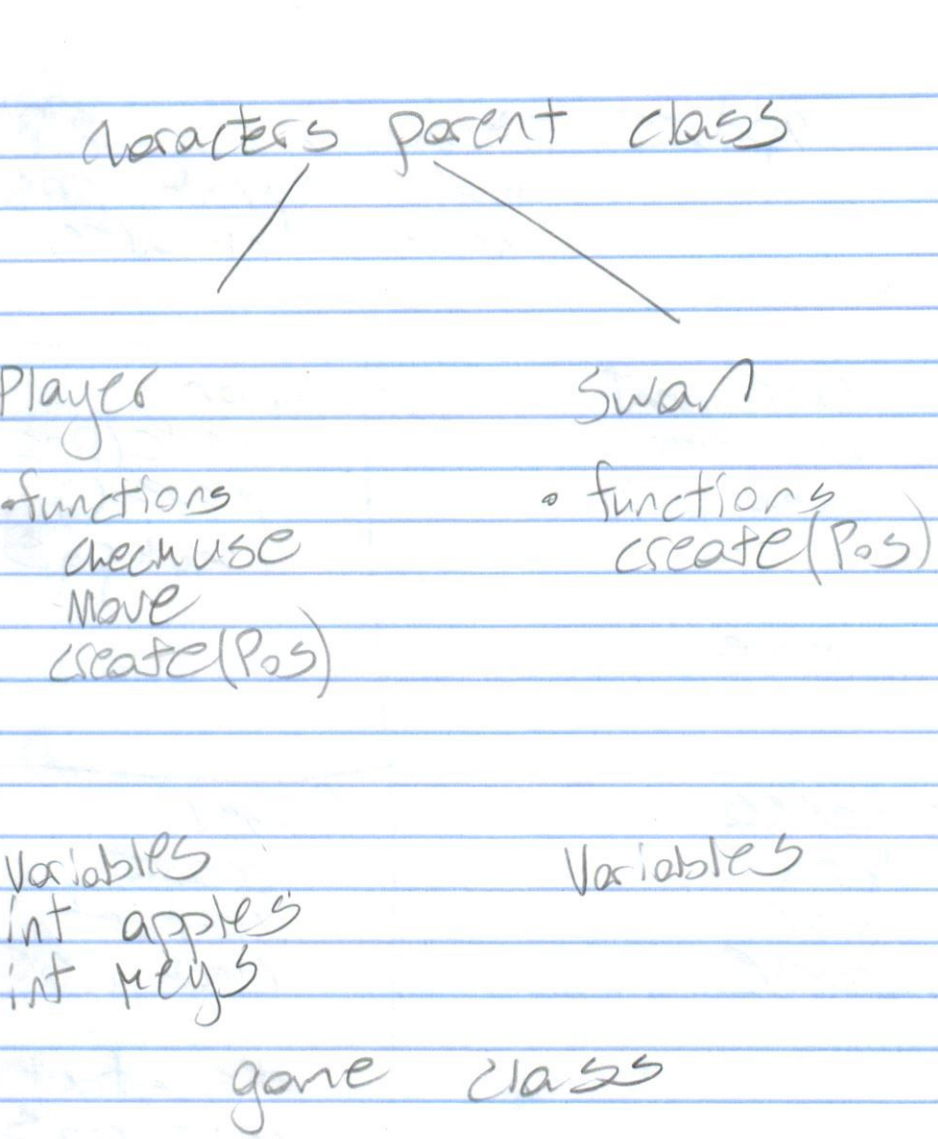
if swan goes on key is it removed? can it go on?

display apples & # of keys at top w/ levels num

20x30

75.99+.15x=

$.99+ $1.4 · .65 · .15 · x


characters parent class
Player
functions
checkUse
Move
create(Pos)
Variables
int apples
int keys
Swan
functions
create(Pos)
Variables
gone class

Checklist

- game.cpp ✓
- game.h ✓
- characters.h ✓
- actor.h ✓
- actor.cpp ✓
- swan.cpp
- player.cpp
- main.cpp

//need to do
- clean up mem
- remove door()
- winning screen

- Check player next
- reset player

Requirements for swan:

move
check if on top of item
check for character
check valid move

Algorithm for swan:

```
loop through until goodmove == true
{
    generate random number for movement between 0 & 8
    if number == one of the numbers
    {
        create new coordinates
        check that square to see if can move there
        {
            if can
            {
                set current square to character previously there
                get character of what is in next square
                set position to that of new square
                check if player next to swan
                    if so then call reset player function
                set bool good move equal to true
            }
            if cant
            {
                goodmove = false
            }
        }
    }
    if number matches other number
        do same thing but move in different way
    if zero then do nothing
}
```

Requirements for player

move (take in way to move)
check on top of item
check for swan
check valid move

Algorithm for player

create coordinates based on input from player

```
check valid coordinates
{
        if not
                do nothing
        if valid coordinates
        {
                set current square to character previously there
                get character of what is in next square
                set position to that of new square
                check if player next to swan
                        if so then call reset player function
        }
}
```

Yes the actor class is a purely abstract class. In my opinion it does not simplify the design, but it is required in order to have a single array of all actors. This is not the way I would have done it but I am required to do so by the design constraints laid out in the assignment.

Some problems that may occur with the swan is not having any valid moves, meaning it would go in
an infinite loop. To resolve this I added a chance for it to do nothing instead.

For the player, an error could be trying to move somewhere where they can't. To resolve this I made it so that does nothing and wastes a turn.