Author: Peter Nguyen
Date: 5/29/15
Description: CS 162-400, Final Project Reflection

Understanding

Requirements:
Ten-room structure: The final project requires that an adventure game be created based around a basic structure of ten rooms that will be defined as classes. There will be a abstract base room class that the ten rooms will be derived from. There must be at least three derived classes that have different characteristics or functions from the master class. Each room class will contain four pointers (north, south, east west) that point to other rooms. If a pointer doesn't point to another room, it can point to something else such as "outside." This could be implemented as a linked list. There should be a way to keep track of which room the player is in.

Inventory system: There could be a player class that holds all the information for a player, including which room he's in. The player must have some type of inventory system to carry items (with a limit). This could be implemented as a linked list.

Interaction: The player should interact with the rooms or have some specific tasks, rather than just collecting items.

Time limit: There should be some method to increment and keep track of time to create urgency in the game.

Goal: There must be some overall goal that the player is required to find.

Theme: The game must have an overall theme or storyline.

Techniques needed: Object-oriented design techniques including abstract base classes, derived classes, function overriding. Linked list structure for the ten-room structure and inventory system. Pointer manipulation for navigation and inventory updating.

Design

Theme and design: The game will be set in the old West and will involve you trying to catch the notorious criminal El Diablo. You will have to fight various enemies along the way. It will use the Character abstract class from assignment 4 to allow for combat. The main player class as well as the enemies you fight will be derived from the Character class. See attached diagram for a map of the game and all Location classes.

Location class (abstract base class)
- private members:
  - Location* north
  - Location* south
  - Location* east
  - Location* west
  - string item

- printDescription function: prints the location description and menu of navigation and action choices *(pure virtual function)*

- specialAction function: performs a special action *(pure virtual function)*

SomeLocation class (derived from Location class): example of a Location with no special action
- print description of the plains
- printDescription function (void, params: none)
  - print choices:
    - 1. move north
    - 2. move south
    - 3. move east
    - 4. move west

        - 5. search the area

- specialAction function: returns string
    - return “this action not availalble”

SomeLocation2 class (derived from Location class): example of a Location with a special action
- private members:
    - Enemy1 Henchman *(derived from Character class)*
- print description including a mysterious medicine man in a teepee
- printDescription function (void, params: none)
    - print choices:
        - default move & search options
        - fight the Henchman

- specialAction function: fight
    - call player’s combat function
    - if player wins:
        - return “you win!”
    - else
        - return “you lose!”
        - set gameOver to true

See attached diagram for all the Location classes

Player class (derived from Character class)
- private members:
    - Inventory bag
    - Location* playerLocation
    - bool objectiveMet
    - bool gameOver

- action function: updates the player’s playerLocation or executes action
    - call printDescription for playerLocation
    - get input
    - if input is to move:
        - if direction is NULL
            - then print “you can’t go that way”
        - otherwise set playerLocation to new location
    - if input is to search:
        - get item from Location
        - if item is empty
            - print “Nothing found”
        - otherwise:
            - print the item found
            - ask player if they want to add to inventory
            - if yes, call addItem function
    - if input is special action:
        - call Location’s specialAction function

Inventory class
- private members:
    - int maxContents
    - vector<string> contents
    - bool hasGun
    - bool hasBullets

- addItem function: adds item to contents (param: string)
    - if size of contents = max
        - then print “the bag is full”
    - otherwise
        - push string into contents

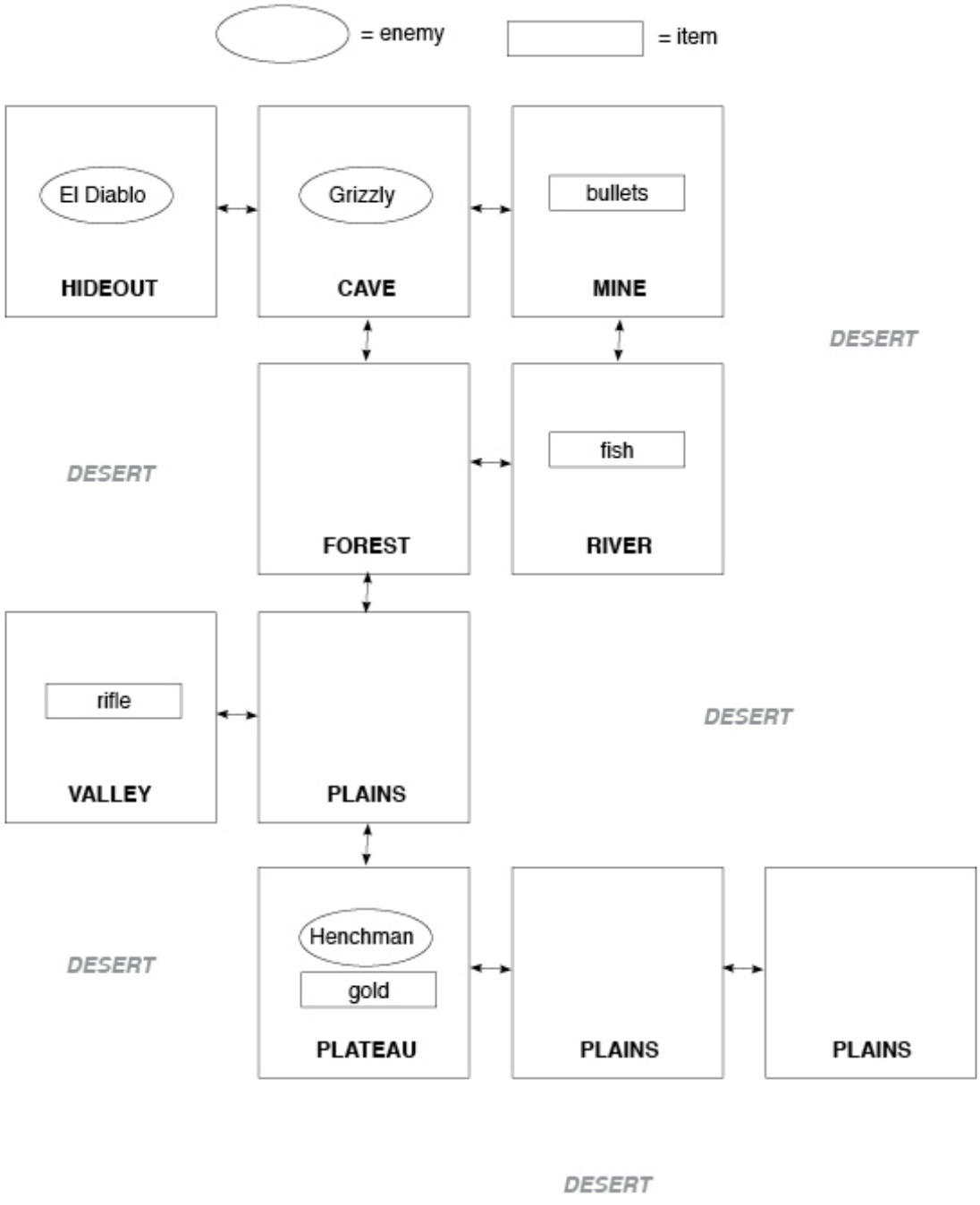
- useItem function: checks if item is in contents and removes it (param: string)
    - loop through contents
        - if string is found then erase it
        - otherwise print “you don’t have that item”

- printContents function: prints contents of inventory
    - loop through contents
        - print item

Game class: tracks goals, keeps track of time
- private members:
    - Location plains1
    - Location plains2
    - Location plains3
    - Location valley1
    - Location valley2
    - Location forest1
    - Location forest2
    - Location mines1
    - Location mines2
    - Location hideout
    - Player you
    - int clock

- play function: plays game while objective is not met or player is not dead
    - loop while player’s objectiveMet is false and gameOver is false and clock has not reached time limit
        - call player action function
        - increment the clock
    - if objectiveMet is true
        - print “congratulations! you won!”
    - if clock has reached time limit
        - print “you ran out of time!”

Main method
- Create a Game object
- Call the Game object’s play function


= enemy
= item
El Diablo
HIDEOUT
Grizzly
CAVE
bullets
MINE
DESERT
DESERT
FOREST
fish
RIVER
rifle
VALLEY
PLAINS
DESERT
DESERT
Henchman
gold
PLATEAU
PLAINS
PLAINS
DESERT

Testing Plan

Test	Sample input	Expected output
Test all structure pointers	Enter each location and test all four directions of movement	Links should take you to the next correct location. If it's not possible to move in that direction, it should tell you. Each location should display correct description and action options.
Standard menu options	Test all four direction options Test heal option Test special actions (below)	Movement in all four directions should behave as expected. If fish is in inventory, heal +5 points, otherwise prints "you don't have fish."
Special action: combat	Select the fight option	Combat should execute as expected, with the player either winning or being killed. If enemy has an item, the item should be added to inventory. If inventory is full, should print "bag is full."
Special action: buy item	Select the purchase option	If you have gold, the item should be added to inventory and gold removed from inventory, otherwise prints "you don't have gold."
Special action: catch fish	Select catch fish option	Should display random behavior consistent with a 1/4 chance of catching fish. If fish is caught, should be added to inventory. If inventory is full, should print "bag is full."
Special action: search for bullets	Select search option	Should display random behavior consistent with a 1/2 chance of finding bullets. If bullets are found, should be added to inventory. If inventory is full, should print "bag is full."
Objective met / Game over / Time runs out	Play through the game until the objective is met. Play the game until the player's characte dies. Play the game until time runs out.	Once all the necessary conditions of the game are met to finish it, it should print appropriate message and exit. If the player dies, it should print that message and exit. The clock counter should increment after each turn. When it reaches the time limit, it should print that message and exit.
Inventory system	Add items to bag until full. Use items in bag. Print bag contents.	Items should be added as expected. When bag reaches limit of 5 items, should print "bag is full." As items are used, they should be removed from the bag. If bag is empty, it should print that. (Note: bullets are not considered a depletable item.) Bag contents should print correctly.

Gun / bullets attack	Acquire the gun separately. Acquire the bullets separately. Acquire both the gun and bullets.	If either the gun or bullets are acquired separately, there should be no effect. If both the gun and bullets are acquired, player's attack power should double.

Reflection on Design Implementation

One of the main challenges of the final project was working out how the different Classes should interact with each other. As this was certainly one of the primary learning objectives of the project, it gave a lot of insight into how careful design planning is essential for all the Class components of a program to work together harmoniously. The decision to use the Character class from assignment 4 worked out well and demonstrated the useful reusability aspects of objects. The necessary functions for this game were seamlessly added onto the existing abstract class to derive the Player class, as well as other new enemy classes, used in the game.

One example of this class interaction involved the Player class and the Location class, with each Location containing a special action, such as fighting an enemy or trying to catch fish. In the case of combat (which is a Player function), it made sense to have the Location class call the Player class's combat function. This necessitated modifying the Location's specialAction function to take a pointer to the Player object so that the combat function could be called.

The original design called for a separate search action that would search the area for an item. Consequently, each Location object would contain a string variable containing the item name. If a Location had no item, the string would simply be a space. However, as the game design progressed, it evolved so that there would be only one Location that would use this function. Therefore, it was eliminated as well as the variable containing the item name. In the end, it was simpler to just make this search function one of the "special actions" for that particular Location.

The original design also had pseudocode for if the special action involved combat with an enemy. However, it did not include the design for the special actions of two of the Locations, which also involved a random element. During implementation, I decided that the "catching fish" action at the River Location would have a 1 out of 4 chance of success. The fish would be used to heal the character (it was also originally planned to double in being able to distract enemies such as bears, but in the end seemed impractical for the gameplay). The "search for bullets" action at another Location had a higher 1 out of 2 chance, as it was quite crucial to find this item.

Early on in the design process, I had considered making the inventory system a linked list. However, given its small size (five items) and basic functionality, using a vector instead helped to simplify the overall design and execution. Another change during implementation involved the useItem function in the Inventory class. Originally, it was merely a void function. However, late during the coding, it was necessary to make it return a boolean value whether the item was actually found in the inventory and used.

One of the most fascinating aspects of the process was tweaking the gameplay mechanics to make a more fun game, as a result of the testing outlined below. Aside from the coding, the initial design took much longer simply because it was much more challenging to think of gameplay elements that would actually make a fun and interesting game (much more challenging than it seems).

I've always liked the evocative nature of simple text descriptions in a game like Zork, so it was also fascinating to write the descriptions of the storyline, characters and different locations. Sometimes the sparseness of language can stimulate your imagination much more than the most complex graphics.

Reflection on Testing

One of the main things that came out of testing, was tweaking the difficulty level of the game. It also had to be taken into consideration that the game should not be so hard that the grader would have difficulty finishing it. Therefore, the Player's stats (such as strength points) were adjusted slightly higher so that he wouldn't die so often. Also the time limit, which was initially 40 turns, was increased to 50 and then finally to 60 because of the rather dramatic change in gameplay detailed below.

Originally, when the Player dies, it was always intended that the game should simply end. However, during testing,

it just seemed too harsh to make the user (and the grader) have to start the game all over again if they got a bad roll of the dice and died. So rather surprisingly late in the implementation and testing, I decided to change it so that the Player doesn't die — if he's defeated by an enemy, the game continues, but he is left with only five hit points. This simple adjustment made the game much more interesting and enjoyable. This made the catching fish dynamic much more important, as now the Player would have to heal more often after a bad encounter with an enemy. Consequenly, the time limit had to be increased because it could take many turns before a fish was caught. This also then made the inventory limit of five items crucial because of the number of fish that one could carry.

Part of the original design involved a gameOver flag which would be set to true if the Player dies. However, as the game could now only end if the objective was met or time ran out, this flag could now be eliminated.

Overall, I learned that testing was a much more involved part of this project because of the need to test gameplay. It wasn't simply a program that needed to output correct results. Because it was a game, the actions had to flow in a logical way and, most importantly, had to result in a sense of accomplishment and (hopefully) fun.