CS 162 Assignment #4 Hunt the Wumpus & Recursion
Design Document due: Sunday, 3/10/2024, 11:59 p.m. (Canvas)
Assignment due: Sunday, 3/17/2024, 11:59 p.m. (TEACH)

NO DEMO NEEDED FOR THIS ASSIGNMENT

Goals:

- Practice good software engineering design principles:
  - Design your solution before writing the program
  - Develop test cases before writing the program
- Practice Object-Oriented Programming
- Implement “Is-a” relationship using Inheritance
- Implement polymorphism and practice STL template by using vector
- Implement recursion
- Practice file separation and create Makefile
- Use functions to modularize code to increase readability and reduce repeated code

NOTE: Since there is no demo for this assignment, the submission guidelines are slightly different from the previous assignments. Be sure to understand the readme.txt requirement.

Introduction & Problem Statements

This assignment has two parts. In part 1, you will be implementing a game called “Hunt the Wumpus” using class inheritance and polymorphism. In part 2, you will solve a problem using recursion.
Here is the file you can start with:

Please download the following skeleton code to start.
Command to download:

```
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/assignments/assign4.zip
```

Command to extract:

```
unzip assign4.zip
```

**Part 1: Hunt the Wumpus**

**Motivation**

The goal of this part of the assignment is to start working with polymorphism and C++ template classes from the STL (standard template library). Follow the directions below.

**Introduction**

Hunt the Wumpus is a game set inside the cave of the Wumpus, a scary, gold-hoarding monster who likes to eat people who disturb its sleep. The player's goal is to guide an adventurer to find its hidden gold and escape alive. The Wumpus lives in a large cave of rooms arranged in a grid, where each room has four tunnels leading to the rooms to the north, east, south, and west.

The adventurer starts the game in a random empty room in the Wumpus' cave. Their starting position is also the location of the escape rope that they must use to escape after they've completed their task.

Each room may be empty, or it may contain one (and only one) of four "events" (all described below): two kinds of dangerous hazards, the terrifying Wumpus, and the gold treasure. When the adventurer is in a room that's adjacent to one containing an "event", the player will receive a percept (a message) to inform them about the event the adventurer is close to.

The player wins the game if they can safely guide the adventurer through the cave to pick up the gold and make it back to the escape rope unharmed! During the game, the adventurer also has the option to fire arrows which could “hunt the Wumpus”.

*Note: you are not required to implement the “do-again” functionality, as it is one of the “extra credit options”. (See below for detailed information)*

**Game Elements:**

1. **The Adventurer**

   The adventurer will start with an oxygen level equal to 2 * side length of the cave. The consumption of the oxygen will be further explained in the “Submerged Pool Access” hazard.

   Each turn you may take one of two actions to guide the adventurer:
   
   - **Move**: You can move through a tunnel to an adjacent room.
• **Fire an Arrow**: The adventurer begins the game with three arrows. As long as the adventurer still has at least one arrow, the player can choose to fire one in any direction (i.e. north, south, east, or west). The arrow continues flying in the same direction until it hits a wall or travels through three rooms. If the arrow enters the Wumpus’ room, it pierces the Wumpus’ heart and kills the monster.

2. **The Wumpus**

Usually, the Wumpus is peacefully asleep in its cave. Two things wake the Wumpus up, however: 1) the adventurer entering its room or 2) shooting an arrow and missing it. If the Wumpus wakes up while in the same room as the adventurer, the Wumpus will angrily eat the adventurer, ending the game. If the Wumpus just wakes up from hearing an arrow being fired, though, there’s a 75% chance it will move to a random empty room in the cave to avoid being found.

3. **Hazards**

There are two kinds of hazards in the Wumpus' cave:

• **Stalactites**: Two of the cave’s rooms have stalactites in them. If the adventurer enters a room with a stalactite:
  
  o There is a 50% chance that the stalactites fall on them, and the player loses immediately.
  
  o There is a 50% chance that the stalactites do not fall on them, and the game continues normally.

• **Submerged Pool Access**: Two rooms provide entry to the cave’s underwater pool. When the adventurer enters a room with pool access, they 'dive' in and begin exploring beneath the water:
  
  o Each turn spent underwater reduces the adventurer’s oxygen level by 1, so they must return to the surface in time to avoid running out of oxygen.
  
  o The adventurer can resurface by re-entering the room with pool access, whether it’s the same room they initially entered or the other room with access.
  
  o Upon returning to the surface, the adventurer’s oxygen level is immediately reset to the initial value (2 * side length of the cave).
  
  o While underwater, the adventurer can still see their surroundings (i.e., have percepts printed, see below), but they cannot encounter events or shoot arrows. For example, if an adventurer underwater enters a room with the Wumpus, nothing consequential would happen.”

*Hint for implementation: instead of creating a 2nd grid to represent the submerged pool, consider using a boolean variable to indicate whether the adventurer is at surface or underwater.
4. The Gold

The gold is a treasure sitting in a room that contains neither a hazard nor the wumpus. If the adventurer is in a room containing the gold, they automatically pick it up and take it with them. After that point, the gold should no longer be present in the game (i.e., the room that contained the gold should now be empty, and the gold’s percepts should no longer be triggered—see below).

5. Percepts

When the adventurer is in a room directly adjacent (east, south, west, or north) to a room containing an "event", the player receives the following messages:

- **Wumpus**: "You smell a terrible stench."
- **Submerged Pool Access**: "You hear wind blowing."
- **Stalactites**: "You hear water dripping."
- **Gold**: "You see a glimmer nearby."

Notice that there’s no percept for the escape rope! That means the player will have to remember where they started and find their way back to that tile after they’ve completed their task.

As an example of how the percepts work, if the adventurer is standing in an empty room with the Wumpus one room to the North, the Gold one room to the East, and Pool Access two rooms to the South, they would receive the following messages at the beginning of their turn:

You smell a terrible stench.
You see a glimmer nearby.

Remember, the percepts don’t tell you where the hazard or gold is, just that it’s somewhere close!

**Program Requirements:**

- Your program should allow the user to play ‘Hunt the Wumpus’, as described above.
- The Wumpus’ cave is represented by a square grid. The side length of the grid should be specified as an initial input to your program. Caves smaller than 4 rooms on one side aren't allowed. The maximum number of rooms on one side is 50. You should visualize the grid to allow the user to play the game. In particular, you should display the grid, and indicate within the grid which room the player is in. An example visualization of a 4x4 grid might look like this, where the * character represents the location of the adventurer:
Your program must also accept a second input which will be either "true" or "false". If the input is false, then the program will run as normal. If the third input is specified as "true" then your game must operate in debug mode.

When your program is operating in debug mode, the player's map will show a "cheat view" with locations marked for each of the following: Wumpus (W), pool accesses (P), stalactites (S), gold (G), player (*), escape rope (~). (Note: feel free to pick your own marks for the player and those events)

To navigate the cave system, the player must be able to type "w" (north), "a" (west), "s" (south), or "d" (east). In order to fire an arrow, an "f" should be used, followed by either "w", "a", "s", or "d" to indicate the direction. For example, the user would enter "f" and "d" to fire an arrow towards the east.

Your code must have the following classes: Room, Event, Wumpus, Pool, Stalactites, and Gold. You may create more classes if they would be helpful.

The Event class must be abstract (i.e. it must contain purely virtual functions), and the Wumpus, Pool, Stalactites, and Gold classes should all be derived from Event. Remember, every event does something when the adventurer enters the same room as the event, and will display a message when the adventurer is nearby. Your design for the Event class should reflect this. For example, your Event class might have a percept() function that is called whenever the adventurer is in a room adjacent to the event, where the Wumpus, Pool, Stalactites, and Gold classes implement their own specific versions of the percept() function. Similarly, your Event class might have an encounter() function that is called when the adventurer enters the same room as the event (on surface), with the individual event classes implementing their own specific versions of encounter().

You must use the Event classes polymorphically. In other words, your Room class may only contain a member of the Event class but not members of the Wumpus, Pool, Stalactites, or Gold classes.
• Each Room contains at most one Event, but it’s possible that it contains no Event. The design of your Room class should reflect this.

• The grid representing your cave should be implemented using the std::vector class.

• Your program may not have any memory leaks or memory errors. I strongly recommend that you use valgrind to occasionally test your program as you develop it (even when everything seems to be working).

• The Big 3 must be implemented as appropriate.

• Your program must be factored into interface, implementation, and application. Specifically, you should have one header file and one implementation file for each class, and you should have a single application file containing your main() function. You should also include a Makefile that builds your program.

• Lack of correct coding style will incur an automatic 10 point deduction.

• You must follow the spirit of the assignment.

Hints

• Polymorphism only works when you are working with references or pointers. If you use the value of an object directly, it may not select the correct member function.

• Hunt the Wumpus is a game all about hiding information from the player, which might make it hard to debug! Your life will be easier if you implement the debugging mode first and then finish your implementation of the final version.

Extra Credit (up to 40 pts)

Implement as many of the following extra credit features as you would like. Note, however, that you can only earn at most 40 points of extra credit, even if you complete all of the extra credit options:

• (5 points) Allow the user to play the game again. If the adventurer perishes while searching for the Wumpus, the player should be presented with the option to start the game over with the same cave configuration, start the game over with a new random cave configuration, or quit the game entirely.

• (5 points) Wumpus walks around. Every turn, the Wumpus moves to an empty adjacent room. Note: if all adjacent rooms are not empty, then the Wumpus stays in the same room for that turn.

• (10 points) Hidden passage event. Two empty rooms have hidden passage events in the game (or two on each floor, in case of a multi-level cave system; see below). Walking into a room that has a hidden passage warps you to the other, if the adventurer is not underwater. The percept for the passage event is: "you feel a breeze".
• (10 points) Armor event. There is a room containing the armor. The adventurer can find armor (similar to how they can find the gold), which will protect them from 2 otherwise fatal encounters, after which the armor is destroyed (e.g., it protects them from stalactites twice, or from stalactites and the Wumpus once each, etc.). The percept for the armor event is: “You feel the power.”

• (10 points) A random, empty room in the game (or on each floor, in case of a multi-level cave system) closes off every 5 turns. A closed room cannot be entered / traversed if the adventurer is on surface. No percept necessary for closed-off rooms, but an error message should be displayed when the player tries to enter a closed-off room.

• (10 points) Implement an AI class that plays the game for you. This AI class should use the same interface to the game that the player does. That is, it should use percepts to learn about the world and make decisions.

• (20 points) Make a multi-level Cave. Implement a multi-level cave system by introducing a new class called "Cave" that contains a vector of 2-dimensional grids. Each grid will have the same side length based on the user inputs. Additionally, the program should accept another input specifying the number of levels. In each level, there will still be two stalactite rooms and two accesses to the submerged pool (and two passages, if you implement that extra credit), while the entire cave will have only one Gold and one Wumpus (and one Armor, if you pick that extra credit). Furthermore, each level will include a ladder up (^) and a ladder down (v) to facilitate movement between levels. The winning condition stays the same: the adventurer must collect the gold and return to the starting position (where the rope is) in the first level.

• (5 points) Implement the WASD controls so that the user can simply press the desired key (to move locations) without needing to press "Enter" afterwards. You may use Ncurses library if you are implementing this extra credit. More information about ncurses: http://jbwyatt.com/ncurses.html

Part 2: Recursion Problem
There is a staircase with N steps. Suppose you can take small steps, medium steps, and large steps.
A small step traverses one step on the staircase:

A medium step traverses two steps on the staircase (i.e., it skips one step):

A large step traverses three steps on the staircase (i.e., it skips two steps):
Consider that, for any given staircase, there are many ways to get to the top.

Example 1: If N = 3 (the staircase has 3 total steps), then there are 4 ways to get to the top:
1. Taking one large step
2. Taking a medium step followed by a small step
3. Taking a small step followed by a medium step
4. Taking three small steps

Example 2: If N = 4 (the staircase has 4 total steps), then there are 7 ways to get to the top:
1. Taking a large step followed by a small step
2. Taking two medium steps
3. Taking a medium step followed by two small steps
4. Taking a small step followed by a large step
5. Taking a small step, followed by a medium step, followed by a small step
6. Taking two small steps followed by a medium step
7. Taking four small steps

Example 3: If N = 5 (the staircase has 5 total steps), then there are 13 ways to get to the top:
1. Taking a large step followed by a medium step
2. Taking a large step followed by two small steps
3. Taking a medium step followed by a large step
4. Taking two medium steps followed by a small step
5. Taking a medium step, followed by a small step, followed by a medium step
6. Taking a small step followed by two medium steps
7. Taking two small steps followed by a medium step, followed by a small step
8. Taking a small step followed by two medium steps
9. Taking a small step, followed by a medium step, followed by two small steps
10. Taking two small steps followed by a large step
11. Taking two small steps, followed by a medium step, followed by a small step
12. Taking three small steps followed by a medium step
13. Taking five small steps

Notice: A way of getting to the top of the staircase only counts if it gets exactly to the top of the staircase. For example, taking two medium steps does not count as a valid way of getting to the top of a staircase of size 3 (but it would count for a staircase of size 4).

Complete the ways_to_top() function in stairs.cpp. It accepts N as a parameter and returns the number of ways to get to the top of a staircase with N steps under the rules described above. Note that it does not have to print out all of the ways to get to the top; it only has to compute and return the number of ways to get to the top. There is an example program in recurse.cpp that provides some limited testing of your implementation (though you’re encouraged to test it further).

Do not modify the header of the ways_to_top() function in any way. That is, you may not modify the function’s return type, name, or parameters. You may write additional functions and call them from within the ways_to_top() function if you would like, but it is not necessary.

It’s true that there’s an efficient non-recursive solution to this problem, but you must
implement a recursive solution.

Hint: With recursive thinking, the function only needs about 10 lines of code (including a few recursive calls).

Programming Style/Comments

In your implementation, make sure that you include a program header for both parts of the assignment. Also ensure that you use proper indentation/spacing and include comments! Below is an example header to include. Make sure you review the style guidelines for this class, and begin trying to follow them, i.e. don’t align everything on the left or put everything on one line!

/*****************************************************/
** Program: wumpus.cpp
** Author: Your Name
** Date: 3/10/2024
** Description:
** Input:
** Output:
/*****************************************************/

Design Document – Due Sunday 3/10/2024, 11:59pm on Canvas
Refer to the Canvas Design Guide page for instructions and expectations.

Step 1: Take the Design Exercise 4 on Canvas.

Step 2: Create a design document (Example_Design_Doc.pdf) that contains the following parts:
1. Understanding the Problem/Problem Analysis
2. Program Design
3. Program Testing
(Note: you need to create a design doc for both parts of this assignment!!!)

Electronically submit your Design Doc (.pdf file!!!) by the design due date, on Canvas.

Program Code – Due Sunday, 3/17/2024, 11:59pm on TEACH

Additional Implementation Requirements:
- Your user interface must provide clear instructions for the user and information about the data being presented.
- Your program must catch all required errors (invalid inputs of the same data type) and recover from them.
- You are not allowed to use libraries that are not introduced in class, more specifically, you may not use the <algorithm> library in your program. Any searching or sorting functionality must be implemented "manually" in your implementation.
Your program should be properly decomposed into tasks and subtasks using functions. To help you with this, use the following:
  o Make each function do one thing and one thing only.
  o No more than 15 lines inside the curly braces of any function, including main(). Whitespace, variable declarations, print statements, single curly braces, vertical spacing, comments, and function headers do not count.
  o Functions over 15 lines need justification in comments.
  o Do not put multiple statements into one line.

- No global variables allowed (those declared outside of many or any other function, global constants are allowed).
- No goto function allowed.
- You must not have any memory leaks or memory errors.
- Your program should not have any runtime errors (e.g. segmentation faults).
- Make sure you follow the style guidelines, have a program header and function headers with appropriate comments, and be consistent.

Submission Guidelines: README.txt

Instead of a demo, you will create a README.txt file that include the following information:

1. Your name and ONID
2. Description: One paragraph advertising what your program does (for a user who knows nothing about this assignment, does not know C++, and is not going to read your code). Highlight any special features.
3. Instructions: Step-by-step instructions telling the user how to compile and run your program. If you expect a certain kind of input at specific steps, inform the user what the requirements are. Include examples to guide the user.
4. Limitations: Describe any known limitations for things the user might want or try to do but that program does not do/handle.
5. (Part 2 only) Extra credit: If your program includes extra credit work, describe it here for the user.
6. Your answers to the following questions:
   a. For part 1 (Hunt the Wumpus), what is the hierarchy of the classes? Which classes represent a “has-a” relationship, which classes represent an “is-a” relationship?
   b. For part 1 (Hunt the Wumpus), which function(s) did you make virtual? Pure virtual? Please explain why.
   c. Did you implement the Big 3 for any classes? If yes, why? If not, why not?
   d. For part 2 (Recursive stair problem), what is your thought process when designing the recursive function? How did you determine the base case and recursive call(s) for your function?
   e. In this assignment, which requirement did you find most challenging? How did you overcome it? What would you do differently if given a second chance?

Here is an example README.txt file (with first 5 items) for a hypothetical assignment (not this one): https://web.engr.oregonstate.edu/~songyip/Teaching/CS16X/README_example.txt
Compile and Submit your assignment

In order to submit your homework assignment, you must create a **zip file** that contains your `.h`, `.cpp`, README.txt, and makefile files. This zip file will be submitted to TEACH. In order to create the zip file, use the following command: (Assuming your ‘assignment4’ folder contains ‘part1’ and ‘part2’ folders, which then contain your .h, .cpp, and makefile files)

```
zip assign4.zip ./assignment4 -r
```

Special Notes

This is the final programming assignment for this course, and you are not required to demo this code. Instead, the TAs will grade your code on their own during finals week. This is why it is important for you to create a **readme.txt** with any additional information that you feel the TAs should know.