ILLINOIS TECH

CS 350 Computer Organization and
Assembly Language Programming
Fall 2023

CS 350: Project Two

Project Due:
December 1, 2023, 11:55 pm
Code submission only

In this project, you will create a game that user can play on console, using MIPS!

The game consists of a 2D map (top-view), obstacles, portals, demons, potions, you name it!
The main objective of the game is straight forward:
The user should navigate the game hero through the dungeon and get to the Exit Gate.

Along the way, the user should avoid the potential demons, but can use the portals and drink the potions for score!

There are only three specifications/requirements to this project:

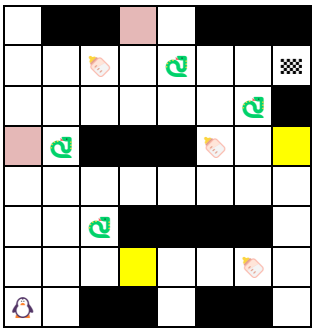
1) We provide the map. Your program should incorporate the map we provide, by hardcoding it into your data segment. The format will be explained.
2) The game should be interactive, and the user should be able to use WASD keys to move the hero in the map:
   - W for up
   - A for left
   - S for down
   - D for right
3) The program should terminate when the player reaches the exit gate of the map.

The rest is suggestions and extra features for your program and bonus points.
We'll leave it up to your imagination to make your game more interesting and have fun with it!

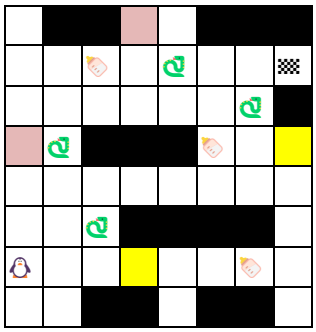
You have learned how to break down a problem, and you should apply the same principle here.
In order to give you a head start, this handout breaks up the tasks and helps you succeed.

Gameplay:

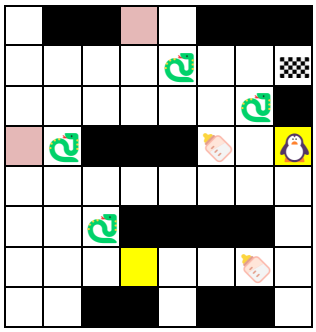
Think of the game as a top-view adventure game. The map consists of cells in a 2D space, similar to this:



Initial State



After the first W



WWWWSADDWDDD

The user can move the hero in the map (the little pinguin in the figure),

- If the player will be gone outside of the map borders, or into obstacle/walls (black cells in the figure). There won't be any move and the player will stay put.
- If the player moves to the exit gate cell (checkered flag), the hero wins, and the game terminates.

Additionally, you can implement these features:

- If the player moves to a cell where there is a potion (bottles in the figure), the hero picks it up and scores. The potion will not be there anymore after that.
- If the player moves to a cell where there is a demon (snakes in the figure), they lose all their points, and will start at the beginning. However, "the game" does not restart. E.g. the potions won't come back.
- If the player moves to a cell where there is a portal, she will end up on the other end of the portal (portals are shown in the figure with yellow/pink cells).

An example is shown at the top of the screen. Starting from the initial state, follow each move:
The player moves up 4 times. With the last move, she lands on the pink portal, which causes her to appear on the other pink cell instead.
The player moves down and left, picks up the potion and scores.
However, she takes two right steps and meets the demon. This means losing the score and showing up at the initial position.
Following the next few moves, the hero lands on the yellow portal, which is why she is now at the other end of the yellow portal.

Guidelines

The first questions that you might ask are:
How to approach this? How to represent the game? How does the user interact with the game?
One way that you can approach this project is to disassociate the user inputs, and the rest of the game. This game consists only of static components, except user input. You can think of your game as a state machine.
You have learned about state machines before, and we have talked about them in the scope of hardware and processor design. You can apply the same principle here.
Your game and all of its contents are included in the “state of the game”, which should be stored in memory. Your program waits for user input, parses the user input, takes the necessary action which means changing from one state to another. That’s it!

Game State

You can think of your game as being constantly saved/loaded. I.e. the state of the game, should be all the information needed to represent the map/hero/etc. With all the details.

- The static map
- The location of the player in the map
- The state of all the things that can change. In this game, the only dynamic part is the potions. Potions are not static in the map, since they vanish once the hero drinks them.
- Whether the game is still running, or terminated.
- The score

We provide you with the static map which includes starting position, exit gate, obstacles, demons, initial potions locations (explained in detail). You will include the map as given, in your code. The rest is up to you. We recommend that you define variables in memory for:

- Where the player is. This value should be initialized based on the map data, when the game starts. However, it changes afterwards during the game, based on user actions.
- The score
- Location of the potions. This should be initialized based on the map data, but it may change during the game, based on user actions.
- A dynamic “current” map which gets refreshed on every move.
- There might be other things that you need to store in memory, temporarily or permanently, so keep your options open.

The Map:

The map representation is in memory. However, the game should present the map to the user and allow the user to interact. We'll use the console for this purpose. You should be able to display the map in the console by printing appropriate characters. It's totally up to you what you choose. This handout just provides the defaults.

For simplicity, we'll use a fixed size map: 8x8 cells, similar to the figure above.
The representation of the map in memory, which is how we provide a map to you, is as follows:
An array of 64 bytes, at the fix address of 0x10002000
Each byte represents the content of a cell. The cells are ordered from top to bottom, left to right.
This represents the offsets within the byte array, for each cell:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	.	.	.	.	.	.	.
.							.
.							.
.							.
.							.
56	.	.	.	.	.	.	63

e.g. the byte that is at 0x10002003 will represent the cell which is on the top row, and fourth column from left.

The content of the byte describes WHAT is at that map location.
Here are the values and their meanings:
83 = 0x52 = 'S': Player's initial position
42 = 0x2A = '*': Exit Gate
32 = 0x20 = ' ': empty space
35 = 0x23 = '#': obstacle/walls

Additional items (these can be ignored by replacing them empty spaces):
43 = 0x2B = '+': potions
64 = 0x40 = '@': demons
48..57 = 0x30..0x39 = '0'..'9': portals

The choice of the numbers allows the map itself to be easy to understand, when hardcoded.
e.g. for the initial map figure in the handout, you can simply copy these lines into your code:

```
.data 0x10002000
.ascii " ##0 ###"
.ascii "  + @  *"
.ascii "      @#"
.ascii "1@###+ 2"
.ascii "        "
.ascii "  @#### "
.ascii "   3  + "
.ascii "S ## ## "
```

However, these are just 64 consecutive bytes in memory, one after another (Note that we are using .ascii and not .asciiz). So it is up to you to interpret this data, as the 2D map.

That is generally how 2D arrays are represented in memory:
e.g. Consider the array A[2][3] which has two rows and three columns. You may think of it as a two dimensional matrix:

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]

But essentially, it consists of 6 elements, one after another.
Let's assume the array starts at address 1000, and each element is one byte:

Row	Row 0			Row 1		
Column	Col0	Col1	Col2	Col0	Col1	Col2
element	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]
Memory Address	1000	1001	1002	1003	1004	1005
Offset from base address	0	1	2	3	4	5

Note how by knowing the Column and the Row, you can calculate the Offset,
given N = 3 (number of columns in each row)
Offset = Row * N + Col
And vice versa, knowing the Offset, you can find out the Row and Col values:
Row = Offset / N (integer division)
Col = Offset % N (mod, remainder of the integer division)

This will allow you to:
Given a cell index in the map, what cell is to the left/right/above/below
Whether a cell is on the left/right/top/bottom border/edge of the map.

Portals

To simplify the game and the use of the map, we have chosen ‘0’-’9’ For portals. This makes it a bit simpler to identify if the user has stepped on a portal. Note the range of these values. Additionally, we are using odd/even portal pairs: ‘0’ and ‘1’ are two ends of one portal, ‘2’ and ‘3’ are the ends of one portal, etc.
This should make it easier for you to find “the other end” of a portal.
Pay attention to the binary representations.

Potions

Potions are just there to include a dynamic object in the map, and increase the score and fun! When a player lands on a potion, it should no longer be part of the map. This requires a separate representation for locations of potions.

Demons

Demons are there to remind the hero, that no one is perfect, and to be humble. And maybe reminds them of a couple of other positive virtues. Anyways, they do not like the player, nor their score! When a player moves to a cell where there is a demon, all the accumulated score is lost and the player should begin at the “initial position” again.

Suggested order of implementation

Task 1) Represent your initial map

Reserve a dynamic map in your memory. This map is the same size of the map (64 bytes), but gets re-written at every step. This allows you to modify the current dynamic map, without corrupting the initial game map.
e.g. the “current” map consists of all the static elements from the initial map (free space, obstacles, exit gate, demons, portals), but then the dynamic elements are added to it (hero, currently available potions).

For now, just copy the whole game map (which is at 0x10002000 in your data segment) to your dynamic map. i.e. load each byte from game map, and save it in your map, repeat for 64 times. You have the map!

Then work on using the print character syscall and considering the size of each row (the map has 8 cells in each row), represent each row of the map, in one line of the console.

Remember to incorporate modular design. This should be a function that you can call, not your main program.
Given the example map above, after running your function, the user should see the map on the console:

```
##0 ###
  + @  *
      @#
1@###+ 2

  @####
   3  +
S ## ##
```

Task 2) Create the main program

We suggest that you create the structure of your main program early.
Your main program should call an initializer function that sets up everything at the beginning, and then your main program should enter a loop:

1) Display the current map
2) Check if the game is terminated, exit the loop.
3) Wait and get user input
4) Apply action (which changes the game state)

You can get user input in the main loop, but for other actions, define functions and write empty skeletons for each of them, where for now they only print a prompt message and return.

Task 3) Start your initializer function

The initializer that is called from the main function, has multiple roles:

- Parse the map to populate the player position variable.
- Initialize the score=0
- Any optional prompt (e.g. to let the user know how to play)

Task 4) Make the map dynamic

Update your map function from Task 1, such that instead of copying the **whole** initial map:

- It initializes your dynamic map with all empty spaces (‘ ‘)
- It copies **only the static elements** of the initial map.
  At first only pick:
  - Obstacle/walls
  - Exit gate

  You can ignore/skip all other values. Remember, you have initialized your dynamic map with empty spaces, so ignoring means empty space!
  Demons and portals are also static, but they are optional.

- Reads the player position from your defined variable, and replaces the target cell in the current map, with the appropriate value (an ASCII character of your choice!)

Task 5) Make it move!

The next item is to allow the user to move the hero. Modify your action function, to respond to user inputs:

- ‘w’ move the player position up.
- ‘a’ move the player position left.
- ‘s’ move the player position down.
- ‘d’ move the player position right.
- ‘t’ sets the terminate variable explicitly. This allows you to manually test terminate functionality.

Note that with your modular design, a “move” of the player is simply updating the position variable.
For now, skip checking the feasibility of the move. Just update the variable according to the direction.

Task 6) End Game!

After moving the player, your program should look for possible outcomes.
The most important outcome, is to end the game! Check the new position of the player in the map. If that cell is the Exit Gate cell, then the game should finish. Note that with your modular design, an “End Game” is simply updating the terminate variable.

Task 7) Check obstacles and boundaries

The user action is only feasible, if the hero is not going out of the boundary of the map. Your program should add some checks to prevent that.
Additionally, the player may end up in a wall/obstacle after a move. Your program should have checks and conditions that prevent that. You don't need to implement any prompt or particular action. It only skips updating the player position variable.

Finalize the project

After completing the above tasks, you have a fully functional game, congratulations!
Now you can make it more fun by make it more user-friendly, check for the demon encounters, adding potions and scores, using portals, etc.

Suggested timeline

Nov 17	Nov 18	Nov 19	Nov 20	Nov 21	Nov 22	Nov 23	Nov 24	Nov 25	Nov 26	Nov 27	Nov 28	Nov 30	Dec 1
Task 1			Task 2	Task 3	Task 4					Task 5	Task 6	Task 7	Wrap up