Assignment 8: UNLVScape

1 Purpose

This assignment introduces foundational programming concepts in C++ by teaching students how to manipulate 2D arrays, use structs as lightweight objects, and implement key game mechanics. Students will parse and load structured data from a CSV file into a 2D array using methods like ‘string.find‘ and ‘string.substr‘, which are practical for processing real-world datasets. By programming functions for movement, resource collection, and inventory management, students will develop problem-solving skills in working with constraints such as inventory limits and object interactions.

This assignment will focus on using structs to represent objects. Focusing on structs instead of classes allows students to grasp the core principles of object-oriented design in a simpler context, reinforcing the relationship between data and behavior. These skills are directly applicable to various fields, such as game development, simulation programming, and software systems that involve managing grid-based or spatial data. After college, the ability to process external files, manipulate data structures, and implement rule-based logic will help students build efficient, scalable solutions in a variety of technical domains.

2 Task

In this assignment, the objective is to create a functional game-like simulation of the game **Old School Runescape** by reading and processing a map file, managing player actions, and implementing inventory and banking mechanics using structs and 2D arrays. The tasks focus on reading structured data, manipulating a grid-based map, and programming logical rules for player interactions within a defined set of constraints.

Steps to Complete the Assignment:

1. Reading the map:
   - Write the `readMap()` function to parse `map.csv` and populate the `2D array` representing the game map. You may only use `string.find()` and `string.substr()` to read from the .csv files. No other method will be graded. For a video on this method see: `https://www.youtube.com/watch?v=S2pvOeWyqBc`
   - If a tree or rock is read in they will have a probability that you are able to chop or mine them, respectively. If a tree is read in, then set the chance attribute of the tree to 0.8 (80% chance of chopping). If a rock is read in, then set the chance attribute of the rock to 0.4 (40% chance of mining).
2. Implementing game mechanics:
   - Write the `move()` function to handle player movement. The player cannot go off the edges of the map and cannot move to positions other entities (trees/rocks/bankers/walls) already occupy.
   - Write the `chop()` function to allow the player to chop trees. To chop, the player must be directly next to a tree horizontally or vertically (not diagonally), and their inventory cannot be full.
   - Write the `mine()` function to allow the player to mine rocks. To mine, the player must be directly next to a rock horizontally or vertically (not diagonally), and their inventory cannot be full.
   - Write the `bank()` function to allow the player to bank items from their inventory. To bank, the player must be directly next to a banker horizontally or vertically (not diagonally), and their bank cannot be full.
3. Call game mechanics from the `main` function in the provided spots.

3 Understanding 2D Array Access in AS8

In this assignment, the map is represented as a 2D array stored in memory, where data is stored row by row. This means the first index of the array corresponds to the row (y-coordinate), and the second index corresponds to the column (x-coordinate). However, when working with coordinates in this assignment, students will receive data as `x, y` pairs, where:

- `x` refers to the horizontal position (column).
- `y` refers to the vertical position (row).

To correctly access elements in the 2D array, the order of indices must follow the array's storage format: `map[y][x]`.

- The `y` (row) value corresponds to the first index of the array.
- The `x` (column) value corresponds to the second index of the array.

For example:

- If a record specifies `x = 5` and `y = 3`, the corresponding element in the array would be accessed as `map[3][5]`.
- Attempting to access the array as `map[5][3]` would result in incorrect behavior, as the row and column values would be swapped.

This difference between intuitive `x, y` coordinate ordering and the memory layout of the 2D array (`row, column`) is critical to understand for correctly implementing this assignment.

4 Object Descriptions

This section describes the two primary object types used in the program: `Entity` and `Player`. These objects are implemented as `structs`, encapsulating related data fields to represent map elements and player attributes.

`Entity`

The `Entity` struct represents objects on the game map, such as trees, rocks, walls, and bankers. It contains the following attributes:

- **`type` (string):** Describes the type of the entity (e.g., `"tree"`, `"rock"`, `"wall"`, `"banker"`). An empty string indicates no entity is present at that position.
- **`chance` (double):** Represents the success probability for actions performed on the entity:
  - `0.8` for chopping wood from trees.
  - `0.4` for mining rocks.
  - Default value is `1.0` unless explicitly set.

The `Entity` struct is used to populate the 2D map array, allowing each position to hold information about any present objects. It is also used to populate the 1D inventory array attached to the player.

`Player`

The `Player` struct represents the player in the game and tracks their state, including position, inventory, and actions. It contains the following attributes:

- **`position` (int[2]):** An array storing the player's current x and y coordinates on the map. Default starting position is `(18, 15)`.
- **`inventorySize` (int):** Tracks the number of items currently in the player's inventory. Maximum capacity is `MAX_INVENTORY` (24 items).
- **`inventory` (`Entity[MAX_INVENTORY]`):** An array storing the entities collected by the player, such as `"wood"` or `"rock"`.

The `Player` struct is used to manage the player's interactions with the game world, including movement, resource collection, and inventory management.

5 Function Descriptions (Game Mechanics)

`readMap(Entity[MAP_Y_SIZE][MAP_X_SIZE])`

The `readMap` function populates a 2D array representing the game map by reading data from a CSV file (`map.csv`). The function performs the following tasks:

1. **Opening the File:** The function opens the `map.csv` file using an `ifstream` object to read its contents.
2. **Skipping the Headers:** It skips the first line of the file, which contains column headers.
3. **Reading Records:**
    - For each subsequent line, the function reads the x-coordinate, y-coordinate, and entity type (e.g., `tree` or `rock`) using the `string.find` and `string.substr` methods to parse the comma-separated values.
4. **Populating the Map:**
    - Using the parsed x (column) and y (row) coordinates, the function updates the corresponding position in the 2D array (`map`).
    - It assigns the entity type to the `type` field of the `Entity` struct at that position.
    - It sets the `chance` field of the `Entity` struct:
        - If the entity type is `"rock"`, the chance is set to 40% (0.4).
        - If the entity type is `"tree"`, the chance is set to 80% (0.8).
5. **Closing the File:** Once all records have been processed, the file is closed to free system resources.

`string move(Player &, Entity[MAP_Y_SIZE][MAP_X_SIZE])`

The `move` function allows the player to move to a new position on the game map, provided the destination is valid and unoccupied. The function performs the following tasks:

1. **Getting Input:**
    - The function prompts the user to input an x-coordinate between `0` and `MAP_X_SIZE-1`.
    - It also prompts the user to input a y-coordinate between `0` and `MAP_Y_SIZE-1`.
2. **Validating the Location:**

- The function checks if the location at the given x and y coordinates is unoccupied (i.e., there is no entity present in the `map[y][x]` position).
- If the location is occupied, the function displays an error message indicating that an entity is blocking the path.
- The function then prompts the user to enter a new x and y position until a valid, unoccupied location is provided.

3. **Updating Player Position:**
   - Once a valid position is provided, the player's x and y coordinates are updated to the new values.
   - A success message, `"moved to position x, y"`, is generated, where `x` and `y` are replaced with the actual coordinates.

4. **Returning the Message:**
   - The success message is returned from the function for display to the user from `main`.

`string chop(Player &, Entity[MAP_Y_SIZE][MAP_X_SIZE])`

The `chop` function allows the player to chop wood from a nearby tree if certain conditions are met. The function performs the following tasks:

1. **Checking for Nearby Trees:**
   - The function checks the spaces directly above, below, left, and right of the player's current position for a tree entity (`"tree"`).
   - If no tree is found in these spaces, the error message `"no wood nearby to chop"` is returned from the function to be displayed in `main()`.

2. **Checking Inventory Capacity:**
   - If a tree is found, the function checks if the player's inventory is full by comparing the `inventorySize` attribute of the player with `MAX_INVENTORY`.
   - If the inventory is full, the error message `"inventory too full to chop wood"` is returned from the function to be displayed in `main()`.

3. **Attempting to Chop Wood:**
   - If the inventory is not full, the function generates a pseudo-random number by calling the provided `rng()` function and saving it's returned pseudo-random number to a variable. The `rng()` function generates deterministic pseudo-random numbers each time it is called, ensuring the same sequence of numbers is produced each time the program runs.

- The function checks if the generated random number is below the chopping chance (`chance` attribute of the tree entity).
  - If the number is not below the chance, the error message `"unsuccessful at chopping wood"` is returned from the function to be displayed in `main()`.
  - If the number is below the chance, a `"wood"` entity is added to the player's inventory, the `inventorySize` attribute is incremented by 1, and the message `"successfully chopped wood"` is returned from the function to be displayed in `main()`.

`string mine(Player &, Entity[MAP_Y_SIZE][MAP_X_SIZE])`

The `mine` function allows the player to mine rocks from a nearby rock entity if certain conditions are met. This function works very similar to how the `chop` function worked. The function performs the following tasks:

1. **Checking for Nearby Rocks:**
   - The function checks the spaces directly above, below, left, and right of the player's current position for a rock entity (`"rock"`).
   - If no rock is found in these spaces, the error message `"no rocks nearby to mine"` is returned from the function to be displayed in `main()`.
2. **Checking Inventory Capacity:**
   - If a rock is found, the function checks if the player's inventory is full by comparing the `inventorySize` attribute with `MAX_INVENTORY`.
   - If the inventory is full, the error message `"inventory too full to mine rocks"` is returned from the function to be displayed in `main()`.
3. **Attempting to Mine Rocks:**
   - If the inventory is not full, the function generates a pseudo-random number by calling the provided `rng()` function and saving it's returned pseudo-random number to a variable. The `rng()` function generates deterministic pseudo-random numbers each time it is called, ensuring the same sequence of numbers is produced each time the program runs.
   - The function checks if the generated random number is below the mining chance (`chance` attribute of the rock entity).
     - If the number is not below the chance, the error message `"unsuccessful at mining rocks"` is returned from the function to be displayed in `main()`.
     - If the number is below the chance, a `"rock"` entity is added to the player's inventory, the `inventorySize` attribute is incremented by 1, and the message `"successfully mined rocks"` is returned from the function to be displayed in `main()`.

`string bank(Player &, Entity[MAX_BANK], Entity[MAP_Y_SIZE][MAP_X_SIZE], int &)`

The `bank` function allows the player to deposit items from their inventory into the bank if certain conditions are met. The function performs the following tasks:

1. **Checking for a Nearby Banker:**
   - The function checks the spaces directly above, below, left, and right of the player's current position for a banker entity (`"banker"`).
   - If no banker is found in these spaces, the error message `"there doesn't seem to be any bankers around"` is returned from the function to be displayed in `main()`.
2. **Depositing Items:**
   - If a banker is found, the function transfers items from the player's inventory to the bank until:
     - The player's inventory is empty.
     - The bank reaches its maximum capacity (`MAX_BANK`).
   - For each item deposited:
     - The item is placed from the player's inventory array into the bank array. Items are placed from the inventory to the bank starting from the back of the inventory working towards the front of the inventory.
     - The bank's size (`bankSize`) is incremented.
     - The player's inventory size (`player.inventorySize`) is decremented.
   - If the bank becomes full during this process, any remaining items in the player's inventory are not deposited.
3. **Returning a Success Message:**
   - The function returns a success message of the form:

     `"banked X items"`

     where `X` is the number of items successfully deposited.
   - If the bank becomes full while depositing items, the message is appended with:

     `" - bank full"`.

6 Success Messages

1. If valid x, y coordinates are entered when the player wants to move, return from the function to output in main: `"moved to position x, y"` where `x` and `y` are replaced with the actual coordinates.
2. If a tree is successfully chopped, return from the `chop` function to output in `main`: `"successfully chopped wood"`.
3. If a rock is successfully mined, return from the `mine` function to output in `main`: `"successfully mined rocks"`.
4. Once all items in inventory are placed in the bank or the bank is full, return from the `bank` function to be output in `main`: `"banked COUNT items"` where `COUNT` is the actual count of items banked. If the bank becomes full append `" - bank full"` to the end of the message.

7 Error Messages

1. If an invalid x-coordinate is entered when the player wants to move, output the error message: `"Invalid x-coordinate\n"`;
2. If an invalid y-coordinate is entered when the player wants to move, output the error message: `"Invalid y-coordinate\n"`;
3. If the player wants to move to a space where an entity already exists, output the error message: `"There's a TYPE in your way!\n"` replacing `TYPE` with the type of entity that is already there.
4. If there are no trees around for the player to chop, return the error message: `"no wood nearby to chop"` from the `chop` function to be output in `main`.
5. If the player's inventory is too full to chop, return the error message: `"inventory too full to chop wood"` from the `chop` function to be output in `main`.
6. If the a random number is not low enough to chop, return the error message: `"unsuccessful at chopping wood"` from the `chop` function to be output in `main`.
7. If there are no rocks around for the player to mine, return the error message: `"no rocks nearby to mine"` from the `mine` function to be output in `main`.
8. If the player's inventory is too full to mine, return the error message: `"inventory too full to mine rocks"` from the `mine` function to be output in `main`.

9. If the a random number is not low enough to mine, return the error message: `"unsuccessful at mining rocks"` from the `mine` function to be output in `main`.
10. If there are no bankers around for the player to bank, return the error message: `"there doesn't seem to be any bankers around"` from the `bank` function to be output in `main`.

8 Example Output

For reference output see: provided files in Canvas, automated tests in CodeGrade.

9 Criteria for Success

File Reading and Map Initialization:

- The `readMap` function correctly opens the `map.csv` file and reads its contents.
- The function skips the header row and accurately parses each line into x-coordinate, y-coordinate, and entity type using `string.find` and `string.substr`.
- The map 2D array is populated correctly, with entity types assigned to their respective positions.
- The `chance` field of the `Entity` struct is correctly set (e.g., `0.8` for trees, `0.4` for rocks).

Move:

- The `move` function correctly validates that the target position is within bounds and unoccupied.
- If the target position is occupied, an appropriate error message is output, and the user is prompted to enter new coordinates.
- If the target position is valid, the player's position is updated, and a success message is returned.

Chop:

- The `chop` function checks the spaces directly adjacent to the player for a tree.
- If no tree is nearby, the appropriate error message is returned.
- If the player's inventory is full, the appropriate error message is returned.

- The function uses the `rng()` function to determine success, based on the chopping chance.
- On success, a `"wood"` entity is added to the player's inventory, the inventory size is updated, and a success message is returned.

Mine:

- The `mine` function operates similarly to `chop`, but for rocks, with appropriate error messages and success criteria.
- The function uses the `rng()` function to determine success, based on the mining chance.
- On success, a `"rock"` entity is added to the player's inventory, the inventory size is updated, and a success message is returned.

Bank:

- The `bank` function checks for a nearby banker.
- If no banker is nearby, an appropriate error message is returned.
- The function deposits items from the back of the player's inventory into the back of the bank until the inventory is empty or the bank is full.
- A success message is returned, indicating the number of items banked. If the bank is full, the message includes `" - bank full"`.

Error-Free Code:

- Ensure the program runs without syntax or runtime errors.
- Test various input scenarios to ensure correct validation and output for both valid and invalid data.
- Ensure proper handling of full inventory and full bank conditions.
- Return clear and concise error messages for invalid or impossible actions.

Output and Display:

- All messages (errors and successes) are returned appropriately from functions and displayed in the main game loop.

Code Quality:

- The code is well-structured and uses proper formatting.
- Comments are included to explain the logic of implemented functions.
- All functions meet the requirements outlined in the assignment.

Header Comment:

- Include a detailed header comment at the beginning of your program.
- The header must include all required information.

Compilation and Execution:

- Compile the program using the `g++` compiler to ensure it runs without errors.
- Verify that the output matches the expected result when executed.

Submission:

1. Save the program as `main.cpp` and submit it to CodeGrade before the deadline.
2. Use the feedback provided by CodeGrade to identify and correct any errors, then resubmit if necessary.

Rubric and Grading:

- Refer to the specific rubric grading criteria available on CodeGrade, which is linked through the Canvas page, to self-evaluate the work and ensure all assignment requirements are met.

10 Hand-In Procedure

1. **Save** your code as `main.cpp`. Do not ignore this step or save your file(s) with different names.
2. **Submit** your program source code via CodeGrade as a properly named `.cpp` file prior to the deadline to receive full credit. Any submissions after the deadline will be subject to the class' late policy.