Assignment 1

Overview

In this assignment, you will make a simple, text-based exploration game. In the game, the player will use text commands (`'n'`, `'s'`, `'e'`, and `'w'`) to explore a simulated 2D world. The game will print a suitable message and terminate when the player enters the quit (`'q'`) command, or if the player reaches a special victory location or any of a number of death locations. For any other location, the player will be shown a description and then asked `"Next?   "`. The process of receiving input from the player and responding will repeated until the game terminates.

The main purpose of this assignment is to ensure that you understand how to use two-dimensional arrays, including declaring types using two-dimensional arrays and passing two-dimensional arrays to functions. For Part A, you will define a `World` type to represent the game world and implement some associated functions. For Part B, you will write a `main` function to play the game.

Another purpose of this assignment is to familiarize you with multi-file programs. To keep the assignment from getting to be too long, the files are small and the functions are mostly short. Many functions only contain a single line of code.

Warning: The test programs used in this course sometimes captures and tests the output from the `cout` buffer. Therefore, you <u>must</u> use `cout` to print your output. If you print it in some other way, (e.g. using `printf` or `cerr`), the test program will not see it. Then you will get a mark of zero for that portion of the assignment.

The World Data

The world is represented by a 2-dimensional array of integers, each of which is referred to as a node. The player can move in any of four directions: north, east, south, and west, but may not enter any node with a value of 0 or pass any boundary of the array. This world will have `24` possible legal values (numbered `0` to `23`) for nodes, with the following meanings:

Value	Meaning
0	Inaccessible
1	Start game message
2	Game over message
3	Death node
4	Start node
5	Victory node
6+	Ordinary node

The following world with 6 rows and 9 columns is used in this assignment:

```
                    NORTH

      0    9    9   10   11   12   13   13    3

      9    9    9    0   15   14    3    3    3
 W                                                 E
 E    9    9    0    3   16    3    3    0    0    A
 S                                                 S
 T    0    7    8    3   17   18    0   25    5    T

      6    4    3    3    0   19    0   19    0

      3    3    3    0   21   20   22   23   24

                     SOUTH
```

Among the values shown above, the `3` in the northeast corner indicates that node `(0, 8)` has value `3` (death node), the `13` immediately to the left indicates that node `(0, 7)` has value `13`, and so forth. Similarly, the `24` at the bottom right indicates that node `(5, 8)` has value `24`. These node values also indicate which description is used for each node. For example, node `(0, 7)` has value `13`, so description number `13` should be used for that node.

Loading Data

The data for this assignment is the "blizzard" game map, which is stored in two data files: `blizzard_grid.txt` and `blizzard_text.txt`. We will load additional data files and other game maps in later assignments.

The nodes values in the `blizzard_grid.txt` file are represented as integers. The file contains one value for every position in the array. The first line gives 9 values for the first row of the array, the second line gives 9 values for the second row of the array, and so forth. You can simply read the values in from the file to your array using the extraction operator (>>).

There descriptions in the `blizzard_text.txt` file are represented as text. The first line of the file is the number of descriptions to read. The next line is blank. After that, each description is given as one or more non-blank lines in the file. A new description starts after each blank line. The first description is the file is description `0`, the second one is description `1`, and so forth.

Test Programs

Three test programs are provided, one for testing each of parts A, B, and C. They are named `TestWorld1A.cpp`, `TestWorld1B.cpp`, and `TestWorld1C.cpp`. Only one of these files can be compiled into the program at any one time. The tests in these programs check functions in a certain order. If some function is not working properly, the later functions may appear not to work even if they are correct. Therefore, you should concentrate on getting the functions to work in the order shown.

Requirements

Part A: The **World.h** Header [15% test program]

In Part A, you will define types and constants related to the world. Put them in a file named `World.h`. Throughout this course, make sure that you use exactly the specified file names, which are case sensitive. Here, `W` is uppercase and the remaining letters are lowercase.

Perform the following steps:

1. As the first line in `World.h` and every other header (`.h`) file for the rest of the course, put the following line:
   `#pragma once`
   This line tells the C++ compiler to only compile this file once even if it happens to be `#include`d multiple times.

   1A. Add a line that `#include`s the `string` library. Also put "`using namespace std;`".

2. In `World.h`, define `ROW_COUNT` and `COLUMN_COUNT` as `int` constants with the respective values `6` and `9`.

3. Use a `typedef` command to define the `NodeValue` type to be the same as `unsigned int`.

   - **Hint:** Do <u>not</u> put `const` in the typedef.

4. Define `INACCESSIBLE`, `START_MESSAGE`, `END_MESSAGE`, `DEATH_NODE`, `START_NODE`, and `VICTORY_NODE` as `NodeValue` constants with the respective values `0`, `1`, `2`, `3`, `4`, and `5`.

5. Use a `typedef` command to define the `World` type to be a 2D array of `NodeValue`s with dimensions `ROW_COUNT` by `COLUMN_COUNT`.

6. Test your `World.h` header file with the `TestWorld1A.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program.

   - **Note:** All the test programs for the assignments will use the same `TestHelper.h` and `TestHelper.cpp` files.

   - To simplify compiling, you should have all .cpp and .h files for your program together in the same folder. You can download the `TestHelper.h` and `TestHelper.cpp` files from UR Courses or you can make empty files and copy/paste the contents of the files into them. You do not need to edit these files.

   - If you are using replit, you can compile with: `g++ TestWorld1A.cpp TestHelper.cpp -o game`

     and then execute with `./game`

   - If you are using replit and you want to use the big green RUN button, you should create a `main.cpp` file with just one blank line in it. (The RUN part of replit believes there should always be a file named `main.cpp`.) Then you can compile all .cpp files in your folder by hitting the RUN button. If there are some .cpp files in the folder that you don't want to compile, you can put add /* at the beginning of the file and */ at the end of the file and then it will appear to be empty of code.

Part B: The World Data [35% test program]

In Part B, you will add functions that work with the `World` type. Put the function prototypes in your `World.h` file and the function implementations in a new file named `World.cpp`.

By the end of Part B, your `World` type will have associated functions with the following prototypes:

```
•  void worldClear (World world);
•  void worldLoadAll (World world, string game_name);
•  void worldLoadNodes (World world, string filename);
•  void worldDebugPrint (const World world);
•  bool worldIsValid (const World world,
                      int row, int column);
•  bool worldCanGoNorth (const World world,
                         int row, int column);
•  bool worldCanGoSouth (const World world,
                         int row, int column);
•  bool worldCanGoEast  (const World world,
                         int row, int column);
•  bool worldCanGoWest  (const World world,
                         int row, int column);
•  bool worldIsDeath   (const World world,
                        int row, int column);
•  bool worldIsVictory (const World world,
                        int row, int column);
•  void worldFindValue (const World world,
                        int& result_row, int& result_column,
                        NodeValue value_to_find);
```

There will be additional functions added in Part C.

Perform the following steps:

1. In `World.h`, copy in the function prototypes shown above. The rest of Part B will go in `World.cpp`.
2. Near the top of `World.cpp`, add `#include`s for the `<string>`, `<iostream>`, and `<fstream>` libraries. As always when you use the standard libraries, you will need the line:

   ```
   using namespace std;
   ```

   - **Note:** Do not put `#pragma once` in any source (`.cpp`) file.
3. Add an `#include` for your own header file, "`World.h`". Never `#include` any source (`.cpp`) file into any other file.
4. Copy in the function prototypes into `World.cpp`. For each function, provide a stand-in implementation. For functions that do not return values (i.e. that have a return type of `void`), this should be an empty function body (just `{ }`). For functions that return a `bool`, it should be `{ return false; }`.

- **Note:** We are adding these stand-in implementations because we want to compile out program, and the compiler won't let us until every function has an implementation. We write the real implementations later in Part A

5. Compile and link your `World` module with the `TestWorld1B.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program. It should give you a mark of `12 / 35`.

   - **Note:** This time when you are compiling, you will need to compile `World.cpp`, `TestWorld1B.cpp`, and `TestHelper.cpp`. You must not compile `TestWorld1A.cpp` (because each of the `TestWorld` files contain a function named `main` and only one `main` function is allowed in a program.). For replit:

     ```
     g++ World.cpp TestWorld1B.cpp TestHelper.cpp -o game
     ```

6. In `World.cpp`, add code to the function named `worldIsValid`. This function takes a `World`, a row, and a column as parameters. The row and column represent a location, and the function should return a `bool` indicating whether that location is a valid world location. Valid world locations have a non-negative row value that is strictly less that `ROW_COUNT` and a non-negative column value that is strictly less than `COLUMN_COUNT`.

   - **Hint:** You will need a total for four checks. These can be four separate `if`s or a single `if` that combines four checks inside it.
   - **Observation:** You will not need to use the `World` parameter inside the function. This parameter will be used in later assignments.

7. Add code to the functions named `worldIsDeath` and `worldIsVictory`. Each of these functions should take a `World` and a row and column, representing a location, as parameters and return a `bool` indicating whether that node is of the indicated type.

8. Add code to the four functions named `worldCanGoNorth`, `worldCanGoSouth`, `worldCanGoEast`, and `worldCanGoWest`. Each of these functions should take a `World` and a row and column, representing a location, as parameters and return a `bool` indicating whether the player can move in the indicated direction from that node. If the move would take the player out of the array, the function should return `false`. Also return `false` if the move would take the player to a node with a value of `INACCESSIBLE`. Otherwise, the player can move in that direction and the function should return `true`.

   - **Hint:** You can use the `worldIsValid` function to check whether a location is in the array.

9. Add code to the function named `worldFindValue`. The function should takes a `World`, two references to `int`s and a `NodeValue` as parameters. The `int`s represent a row and a column, and the `NodeValue` is the value to search for. First, the function should set the row and column parameters to both be `-1`. Then it should use two nested FOR loops to search the `World` for a node with a value equal to the `NodeValue` parameter and, if it finds one, set the row and column parameters to that node's location. If there is more than one such value, you can use the location of any of them.

- **Note:** At the end of this function, the row and column will have values of `-1` if and only if the `World` does not contain search value.

- **Warning:** The `TestWorld1B.cpp` file cannot be used to test this function until `worldLoadAll` is working (a later step).

10. Add code to the function named `worldClear`, which takes a `World` as a parameter. It should set every element of the array to `INACCESSIBLE`.

11. Add code to the function named `worldDebugPrint` that takes a `World` as a parameter. It should print out all the node values in the world in a grid and separated by tabs.

    - **Hint:** Display the values in the array using two nested for-loops with `cout` statements inside. Print a newline (`endl`) after the last value in each line and a tab (`'\t'`) after each other value.

    - **Warning:** You must match this format exactly, including using tabs (not spaces) to separate values on the same line. Otherwise, the test program will think the output is incorrect and dock you marks.

    - **Warning:** The `TestWorld1B.cpp` file cannot be used to test this function until `worldLoadAll` is working (a later step).

12. Add code to the function named `worldLoadNodes` that takes a `World` and a `string` as parameters. The `string` represents the file name to load the nodes from. You should first open `blizzard_grid.txt` for reading. If the file cannot be opened, print an error message. Hereafter, you should always print an error message when a file cannot be opened. Then use two nested FOR loops to load (i.e., read) the first `ROW_COUNT * COLUMN_COUNT` integers from the file into the `World` array.

    - **Hint:** Read in the world nodes using formatted I/O (>> notation).

    - **Optional:** Close the input file when you are done. If you do not, it will automatically be closed when the `ifstream` variable goes out of scope, which is normally at the end of the function.

13. Add code to the function named `worldLoadAll` that takes a `World` and a `string` as parameters. The `string` represents the game name. Declare a string variable for the name of the node data file. Calculate a value for this file name by adding `"_grid.txt"` to the game name. Then declare another string variable for the name of the text data file. Calculate a value for this second file name by adding `"_text.txt"` to the game name. For example, if the game name is `"blizzard"` then the two file names should be `"blizzard_grid.txt"` and `"blizzard_text.txt"`. At this stage, use the first of these filename to call the `worldLoadNodes` function. In Part C, you will use the second of these filenames to call the `worldLoadDescriptions` function.

    - **Hint:** You can use "+" to add two strings, i.e., `"Fred "` + `"Flintstone"` gives `"Fred Flintstone"`.

    - **Hint:** As debugging, print your filenames to make sure they are exactly right.

- **Note:** For Assignment 1, the game name passed into this function as the game name will be "`blizzard`". This will be done in part D. In later assignments, the game name will be changed.

14. Test your `World` module using the `TestWorld1B.cpp` program again. At this point, it should compile, run without crashing, and award you full marks.

Part C: Descriptions [20% test program, 7% of this also depends on code]

In Part C, you will add functions to handle descriptions. Put the function prototypes in your `World.h` file and the function implementations in your `World.cpp` file.

By the end of Part C, your `World` type will have associated functions with the following prototypes:

- `void worldClear (World world);`
- `void worldLoadAll (World world, string game_name);`
- `void worldLoadNodes (World world, string filename);`
- **`void worldLoadDescriptions (World world, string filename);`**
- `void worldDebugPrint (const World world);`

-
  ```
  bool worldIsValid (const World world,
                     int row, int column);
  ```

-
  ```
  NodeValue worldGetAt (World world,
                        int row, int column);
  ```

-
  ```
  bool worldCanGoNorth (const World world,
                        int row, int column);
  ```

-
  ```
  bool worldCanGoSouth (const World world,
                        int row, int column);
  ```

-
  ```
  bool worldCanGoEast  (const World world,
                        int row, int column);
  ```

-
  ```
  bool worldCanGoWest  (const World world,
                        int row, int column);
  ```

-
  ```
  bool worldIsDeath   (const World world,
                       int row, int column);
  ```

-
  ```
  bool worldIsVictory (const World world,
                       int row, int column);
  ```

-
  ```
  void worldSetAt (World world,
                   int row, int column,
                   NodeValue new_value);
  ```

-
  ```
  void worldFindValue (const World world,
                       int& row, int& column
                       NodeValue value_to_find);
  ```

-
  ```
  void worldPrintDescription (const World world,
                              int row, int column);
  ```
- **`void worldPrintStartMessage (const World world);`**
- **`void worldPrintEndMessage (const World world);`**

Perform the following steps:

1. In `World.h`, copy in the new (**bold**) function prototypes.

2. In `World.h`, define `DESCRIPTION_COUNT` as a constant with a value of `26`. Choose a suitable data type for integer values that are always zero or positive.

3. Copy the new function prototypes into `World.cpp`. For each of these functions, provide a stand-in implementation. None of these functions return any values (return type `void`), so they should all have empty function bodies.

4. Compile and link your `World` module with the `TestWorld1C.cpp` program provided. You will need the `TestHelper.h` and `TestHelper.cpp` files again. The test program should give you a mark of `7 / 20`.

5. In `World.cpp`, create a global array of `DESCRIPTION_COUNT` strings.

    - **Note:** A **global variable** is one that is declared outside of any function. Put the declaration before any of the functions in the .cpp file.

5A. Add code to the `WorldLoadAll` function to call the `worldLoadDescriptions` function such that the first parameter is a `World` and the second parameter is a string that ends with `"_text.txt"`.

6. Add code to the function named `worldLoadDescriptions` that takes a `World` and a `string` as parameters. The `string` represents the file name for the descriptions of the world nodes. Open `blizzard_text.txt` file and load the descriptions into the global description array.

    - **Hint:** If you are working in replit, you will need to create the file named `blizzard_text.txt` inside replit with no content. Then copy/paste the contents of the file from the website. It will not work to upload the file from the website on a Windows machine because of differences between Windows and Linux (replit runs on Linux).

    - **Hint:** Read in the lines from the description file one at a time using `getline`. If the line you just read is not blank, add it to the end of the current description, followed by a newline. Otherwise, if you have not read enough descriptions yet, advance to the next description. Otherwise, you are done.

    - **Hint:** You will need two extra `getline`s to read the line with 26 and the blank line following it. These instructions should go before your loop that does the reading of the description messages.

    - **Hint:** This function is tricky to write. Add lots of cout statements (for debugging) so that you know what values your variables actually have and so you know which lines the program is actually executing. You may want to reduce the number of descriptions to (say) 3 while you get this function working.

7. Add code to the functions named `worldPrintStartMessage` and `worldPrintEndMessage`. Each of these functions should take a `World` as a parameter. The `worldPrintStartMessage` function should print the description with index `START_MESSAGE` (which is equivalent to 1) and the `worldPrintEndMessage` should print the description with index `END_MESSAGE` (which is equivalent to 2). These descriptions are stored in the global description array.

- **Example:** The `END_MESSAGE` constant has a value of `2`, so the `worldPrintEndMessage` function should print message number 2 from the global description array. This should print:

  ```
  Thank you for playing Blizzard Valley!
  ```

8. Add code to the function named `worldPrintDescription` that takes a `World`, a row, and a column as parameters. The function should determine the description number for that node and then print that description. The description is stored in the global description array. When printing the descriptions, use an array lookup instead of many `if` statements.

   - **Example:** Suppose you wanted to print the node with row `1` and column `2`. (This is the second row and third column because the array is indexed from `0`.) The description number for that node is `9`, so the program should look in position 9 of the global description array and it should print:

     ```
     You are on a mountain slope, lost in the raging blizzard.
     Everything is white.
     ```

9. Test your `World` module using the `TestWorld1C.cpp` program.

Part D: The **`main`** Function [30% = 15% test scripts + 15% test scripts]

In Part D, you will make a simple game using the `World` module. The game will consist of a `main` function in its own file named `main.cpp`.

1. Put `#include`s for the `<string>` and `<iostream>` libraries in `Main.cpp` as well as for your `World.h` header file.

   - **Reminder:** Whenever you `#include` standard libraries such as `<iostream>`, you will also need the line:
     `using namespace std;`

   - **Reminder:** Do not put `#pragma once` in any source (`.cpp`) file.

   - **Reminder:** Never `#include` any source (`.cpp`) file into any other file.

2. Add a `main` function for your program.

3. Add appropriate variables to the `main` function. You will need a variable of the `World` type to represent the game world and row and column variables to represent the player location.

4. After the variables are declared, the `main` function should load the world and initialize the player location to be at the start node. Then it should print the starting message, a blank line, and a description for the node where the player is current located.

   - **Hint:** Use your functions from Parts B and C to do these things.

   - **Hint:** If you are using Visual Studio, the program window usually closes as soon as the program ends. To prevent this, at the end of `main` you can print a message saying `"Press [ENTER] to continue... "` without a newline. Then read in a line of input with `getline`. These instructions should be the last instructions before the `return` statement at the end of `main`.

5. After the starting messages are printed, add a ***main loop***. This is a loop in an interactive program that handles user input, controls the rest of the program, and runs until the program quits.
    - **Note:** We will fill in the contents of the main loop below.
    - **Hint:** One way to control when the main loop stops is with a `bool` variable.
6. After the main loop, the program should print a blank line, followed by the end game message. Then is should terminate.
    - **Note:** If you added a final `getline` command (as described in Part 4), it should be after the messages added in this step.

The remainder of the steps refer to the contents of the main loop:

7. At the start of the main loop, print "`Next? `" without a newline.
8. Then read in the player input. Only the first character matters, but you should read all the characters on the line so they are gone before you go through the main loop again.
    - **Hint:** Use the `getline` command.
9. If the input contains no characters, do nothing or print "`Invalid command!`".
10. If the first character is a `'#'`, this line is a comment. Do nothing.
    - **Hint:** To access the first character of a string, you should use `[0]` or `.at(0)`
11. If the first character is a `'q'`, this is a quit command. Print "`Are you sure you want to quit?  `" without a newline and then read in another line of input. If the first character in the new input is `'y'` the main loop should end. Otherwise, nothing should happen.
    - **Hint:** You can stop the main loop by changing the `bool` variable that controls it.
12. If the first character in the original line is a `'n'`, `'s'`, `'e'`, or `'w'`, the player should attempt to move north, south, east, or west, respectively. Use the appropriate `World` function to test if the move is possible (e.g., use the `worldCanGoNorth` function for north). If the move is possible, move the player in that direction (e.g. decrease the row variable by `1` for north) and print the description for the new node. If the move is not possible, print "`There is no way to go in that direction`".
13. If the player enters any other command, print "`Invalid command!`".
14. At the end of the main loop, check whether the player is at a death node or a victory node. If so, stop the loop.
    - **Hint:** Use your functions from Part B to check whether the player is at a death or victory node.

15. Test your program with the five test cases provided: `testcase1A.txt`, `testcase1B.txt`, `testcase1C.txt`, `testcase1D.txt`, and `testcase1E.txt`. Each test case is a series of commands you should enter to your program in order. The easiest way to do this is to select everything in the file (`[CTRL]+[A]` on Windows) and copy-paste it into the console after starting your program.

- Write a comment at the top of `main.cpp` that states your name and student number.

Formatting [−10% if not done]

1. Neatly indent your program using a consistent indentation scheme.
2. Put spaces around your arithmetic operators:
   `x = x + 3;`
3. Use symbolic constants, such as `INACCESSIBLE`, when appropriate.
4. Write a comment at the top of `main.cpp` that states your name and student number. You do not need to put comments beside almost every line even if you were asked to do so in CS 110.
5. Format your program so that it is easily readable. Things that make a program hard to read include:
   - **Very many blank lines**. If more than half your lines are blank, you probably have too many. The correct use of blank lines is to separate logically distinct sections of your program.
   - **Multiple commands on the same line.** In general, don't do this. You can do it if it makes the program clearer than if the same commands were on separate lines.
   - **Uninformative variable names.** For a local variable that is only used for a few lines, the name of the variable doesn't matter much. But a variable that is used over a larger area (including all global and member variables) should have a name that documents its purpose. Similarly, parameters should have self-documenting names because the function will be called from elsewhere in the program.
   - **No variable names in function prototypes**. Function parameters should have the same name in the prototype as in the implementation. This makes calling the function much less confusing. No marks deducted for this situation in Assignment 1.

Submission

- Submit a complete copy of your source code. You should have the following files with exactly these names:
  1. **`World.h`**
  2. **`World.cpp`**
  3. **`Main.cpp`**
  - **Note:** A Visual Studio `.sln` file does NOT contain the source code; it is just a text file. You do not need to submit it. Make sure you submit the `.cpp` files and `.h` files.
  - **Note:** You do not need to submit the test programs. The marker has those already.
- If possible, convert all your files to a single archive (`.zip` file) before handing them in
- Do NOT submit a compiled version

- Do NOT submit intermediate files, such as:
    - `*.o` files
    - `Debug` folder
    - `Release` folder
    - `ipch` folder
    - `*.ncb`, `*.sdf`, or `*.db` files
- Do NOT submit a screenshot