Programming Assignment 2: GridWorld

Due: **Due**: TBD
(late policy TBD)

Objectives:

1. Design and implementation of *all* data structures for a given specification (including runtime requirements).
2. More practice with linked lists (and arrays/vectors).

Summary: you are given the specifications for an Abstract Data Type along with runtime requirements. Your job: design and implement a class called **GridWorld** which meets the requirements.

Welcome to GridWorld where everybody is a number and lives in a square (unless of course they are dead).

The World: The world is kind of boring. It is an RxC grid (rows and columns). Each entry on the grid is called a *district* and is referred to by its row *r* and column *c* where $r \in \{0..R-1\}$ and $c \in \{0..C-1\}$. We'll sometimes refer to such a district as $D_{r,c}$.

The People: Then there are the people; ***each person is uniquely identified by an integer ID*** (like a social security number). Person IDs start at zero. Each living person resides in one (and only one) of the districts.

Day Zero: When a world is created, just two things determine its initial configuration:

- R: the number of rows
- C: the number of columns

When a world is created, there are no people at all: every district is a wasteland with no population. However, once created there are a number of operations that can be performed on the world (you should be thinking "Abstract Data Type" about now).

Operations: once a world is created, there are various operations that can be performed. These operations are laid out in the table starting on the next page.

C++ Stuff: You have been given a bare-bones file GridWorld.h as a starting point. There are no data members specified -- that is part of your job!

All of the required functions (and constructors) appear as empty "stubs". This is a pretty "blank-slate" assignment!

Your complete implementation GridWorld.h will be the primary deliverable for the assignment.

Next page: GridWorld operations

The Operations:

The table below lists each operation that must be supported including runtime and behavioral requirements.

Operation: Creation of a new world.

Corresponding Constructor:

```
GridWorld(unsigned nrows, unsigned ncols)
```

Description: initializes a world to have the dimensions specified by the parameters and zero population.

Operation: Creating a new person

Corresponding Member Function:

```
bool birth(int row, int col, int &id);
```

Description: by calling this function we are asking for a new person to be created and to place that person in district (row, col).

If the given row/col corresponds to a valid district, this function does the following:

- creates a new person
- assigns a unique integer ID to the person and
- places that person in district (row, col).
- communicates the assigned ID to the caller via the reference parameter id.
- Returns true (for "success").

If parameters row/col do not refer to a valid district in the world, no person is created and false is returned.

Runtime requirement:

O(1). There is a caveat here: this bound will be technically "amortized" because you will be allowed to use the `vector` class from the Standard Template Library as part of your solution.

(An instance of the `vector` class behaves like an array which automatically resizes itself when necessary. It follows essentially the same policy we used in our Stack class case-study -- the resizing operation constructs a new underlying array twice as large as the previous capacity.)

(birth operation cont.)

Rules For Assigning IDs (IMPORTANT): There are two scenarios here:

1) **First Priority - Recycling a previously used ID**: if there is one or more ID that was previously used for a person (who is now dead), you must use such an ID for the person being created. If there are multiple such IDs, you must assign the ID which has been "retired"/unused for the longest time (maybe this helps prevent fraud?)
2) **If case-1 does not apply (no IDs to recycle):** you must assign the smallest unused integer to the person being created.

Operation: "killing" an existing person

Corresponding Member Function:

```
bool death(int id);
```

Description: if alive, person represented by **id** is removed from its current district and the entire world. Data structures updated accordingly.

returns: true if operation succeeds; false if fails (no such living person).

Runtime requirement: O(1)

Tip: think about the ID assignment policy for the birth function; how might it affect your implementation of the death function?

Operation: moving/relocating a person to a target district

Description: if given person is alive, and specified target-row and column are valid, person is moved to specified district and data structures updated accordingly.

return: indicates success/failure

runtime: O(1)

comment/note: the specified person becomes the 'newest' member of target district (least seniority) -- see requirements of members().
This is true even if the target district happens to be the same as the person's current district (basically resets their seniority).

```
bool move(int personID, int targetRow, int targetCol);
```

Operation: querying for the current population of the entire world.

Corresponding Member function:

```
int population()const;
```

Description: simply returns the total number of (living) people in the entire world/grid.

Runtime: O(1)

Operation: querying for the current population of a specific district.

Corresponding Member function:

```
int population(int row, int col)const;
```

Description: simply returns the total number of (living) people in district specified by (row,col). If row/col does not correspond to a valid district, zero is returned.

runtime: O(1)

Operation: determine the district where a particular person lives.

Corresponding Member Function:

```
bool whereis(int personID, int &row, int &col)const;
```

Description: if personID represents a currently living person, the row and column where that person currently lives is reported via reference parameters row and col and true is returned.

If personID does *not* correspond to a currently living person, false is returned.

Runtime: O(1)

Operation: exporting the current members of a particular district.

Corresponding Member Function:

```
std::vector<int> * members(int row, int col)const;
```

Description: creates and populates an integer vector with a snapshot of the current residents of district specified by (row, col). The vector is returned as a pointer.

If there is no such district (row,col), a vector is still created and returned, but it is empty.

Requirement: the members of the district must be ordered in *descending order of seniority:* The person who has lived in the district the longest must be first and so-on.

Runtime: $O(N_{r,c})$ where $N_{r,c}$ is the population of the district in question.

Tip: Think about the ordering requirement above and how functions birth, death and move impact this rule.

Operations: queries for dimensions of world.

Corresponding Member Functions:

```
int num_rows()const;
int num_cols()const;
```

Description: Pretty self-explanatory - just returns the number of row (or columns) in the world. (Note that once a world is created, the dimensions are fixed).

Runtime: O(1)

Comment: these should be one-liners.

Operation: deallocating a world and doing associated cleanup.

Corresponding Destructor:

```
~GridWorld()
```

Description: deallocates all dynamically allocated objects associated with the world -- typical destructor stuff.

Resources:

You have been given the following:

GridWorld.h	This is a file of function "stubs" for the GridWorld class you are implementing. Your job: complete the GridWorld class by fleshing out this file:
driver.cpp	This is a simple interactive program you are welcome to use and modify to help in developing and testing your implementation. It basically creates a GridWorld object and the user enters commands mapping directly to the member functions. The commands are parsed and issued. Results are reported.

Additional Rules and Hints:

RULE: You may use the vector class from the STL (although you don't *have* to). All other external data structure classes (not written by you) are forbidden.

RULE: None of your functions should be printing anything to the terminal!!! If you inject print statements for debugging purposes, be sure to remove them before final submission! Expect penalties for extraneous output!

HINTS/SUGGESTIONS:

On pencil and paper, start with a straightforward implementation that does not necessarily meet the runtime requirements (but meets the behavioral requirements). Analyse the runtime of the various operations. Now start thinking of alternative ways to organize the data to meet runtime requirements.)

Hint: doubly-linked lists?

Come to class! We will be brainstorming strategies!

WARNING: If you come up with a proposed solution that utilizes ***only* arrays and/or vectors**, you almost certainly **will NOT meet all of the runtime requirements** (unless you are essentially simulating linked lists with said vectors/arrays.)

Testing

You need to test your own implementation! Be creative and try to break it!

We will release a test suite which will be a subset of the suite used for grading as the due date approaches. Do not wait for it to do your testing! You want the given tests to merely confirm behavior you have already tested.

Submission

Your primary deliverable is GridWorld.h. Submission will be via blackboard.