H5: The maze generator

This week, you will write a program that generates random mazes and finds a path in there from a beginning to an end position, and then prints the maze on the screen. To give you an idea, this is how the output of your program is supposed to look like:

```
./mazegen 5 6
+---+---+---+---+---+---+
|  .    .   . |         |
+---+---+   +---+---+   +
|       | .    .   . |  |
+   +---+---+---+   +   +
|               | . |   |
+   +---+---+   +   +   +
|   |   |       | . |   |
+   +   +   +---+   +   +
|       |         .    . |
+---+---+---+---+---+---+
```

Your program should perform the following major steps:

1. Read from the program's command line parameters either 2 or 3 integer values. The first one is the number of rows of the maze, the second one is the number of columns. The third parameter is optional. (But you must implement handling it.) If present, the third parameter must be used as seed value for the random generator. (So you can decide to always generate the same maze for testing purposes.) When called with two parameters only, the program must generate a different maze, each time it is run. Hint: you can convert a string from argv[] to an int using **atoi   (https://www.cplusplus.com/reference/cstdlib/atoi/)** .

2. Generate a random maze of the given size, using the algorithm called *Recursive backtracker* (see below). The core of this assignment is using classes in a meaningful way. Before you start implementing, first decide what are the "things" in your program, and create classes for them. (Example things are the maze, coordinates, cells in the maze, or even the command line parameters.) **Your program must implement at least two classes.** But more are likely helpful for achieving good structure.

3. Find a path inside your random maze, from the top-left corner -- we refer to as position (0,0) --, to the bottom right corner. For details, see below.

4. Print your maze with the path to cout. (Details, see below)

Maze generation

The maze consists of walls and cells. The maze must be closed, so there must be no openings in the outer walls. Initially, all cells are separated from each other by walls. Inside the maze, some walls must be removed to connect the cells to each other. All cells must be connected to each other; from any cell it must be possible to reach any other cell by following paths that have been created by removing some of the internal walls. Cycles inside the maze are forbidden. So, the number of removed walls must be equal to the number of cells, minus 1. Fortunately, there is an algorithm, called *Recursive backtracker*, that does this for us. You can watch **this video**

(https://www.youtube.com/watch?v=Y37-gB83HKE) with a very nice explanation. You only need the first 8 minutes where he explains the algorithm. Please do not get carried away with the code he describes. While I love his explanation of the algorithm, his code is using his own graphics library and too many C++ features that are as debatable as advanced. (And if you take *"too much inspiration"* from his code, you will get in trouble anyways...) Not needed, but a nice read is the description of maze generation on **wikipedia (https://en.wikipedia.org/wiki/Maze_generation_algorithm)** .

Path finding

Once you have generated your maze (and stored inside a beautiful, class-based data structure), it is time to find a path through your maze. By convention, the path must always start in the top left corner, and must end in the bottom right corner. The finally identified path must not include any detours. For path finding, not having cycles in the maze makes our lives much easier!

Path finding can be done by a (recursive) technique called backtracking. But all you need is implement the following pseudo code, using your own class data structures:

`Function findPath, given a maze M, and two coordinates, from and to, returns true if a path could be found, false otherwise:`

1. `M.at(from).visited <- true`
2. `if from equals to, return true`
3. `neighbours <- list of all direct neighbours of from that can be reached (that are not blocked by walls)`
4. `for all n in neighbours:`
    1. `if M.at(n).visited == false`
        1. `if findPath(n,to) == true, return true`
5. `M.at(from).visited <- false`
6. `return false`

Maze printing

Maze printing must be done very precisely, according to this description. Otherwise, the automated tests will fail. This means, no other characters, no extra characters, no extra spaces or newlines. Let's consider the following 2-by-2 example maze:

```
+---+---+
|  .    . |
+---+   +
|      . |
+---+---+
```

Each cell is 5 characters wide and 3 characters high. Walls and corners are shared among neighbouring cells. Each corner is denoted as `+`. A horizontal wall is denoted as 3 minus signs: `---` A vertical wall is denoted by `|`

The contents of a cell are either three spaces: `'   '` ,

or two spaces with a . in the middle: `' . '` ,

the latter if the cell is part of the path from the top left corner to the bottom right corner.

Must not do

Just to be sure, using the following C++ features is **not allowed**:

- arrays
- pointers
- iterators
- auto

This tool needs to be loaded in a new browser window

The session for this tool has expired. Please reload the page to access the tool again