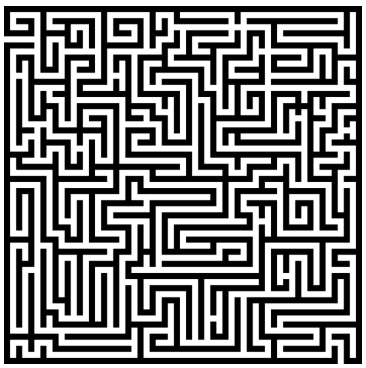
# Project 4: `MazeSolver`



For this project you will build a **`MazeSolver`**.
Your maze solver will read the maze from an **input file**, find a path that leads to the exit and print the solution to the **standard output** (solution = maze with highlighted path from start position to exit)

## Maze representation:

In **the input file** the maze will be represented as a string of characters separated by spaces (`' '`).

The first two characters in the input file will be integers representing the number of **rows** and **columns**. The remaining characters will be valid maze characters representing a maze.

If you think of a maze in terms of hallways and walls, the **maze characters** are as follows:

**_** represents a hallway (input and output character)

***** represents a wall (input and output character)

**$** represents an exit (input and output character)

**>** represents a **PATH** towards the exit (output character only)

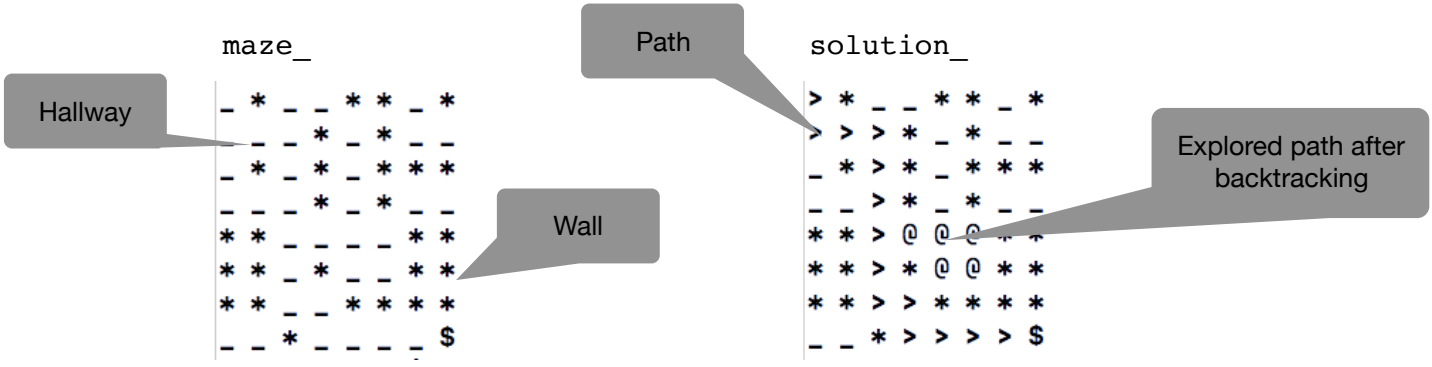
**@** represents a location that has been visited and then **BACKTRACKED** out of on the solution (output character only)

**X** represents a location that has been **VISITED** on the maze ("working" intermediary character on maze)

`MazeSolver` will represent the maze and a solution as 2-dimensional arrays containing maze characters.

**Simplifications:** start position is always [0][0] and MazeSolver will search for a solution moving in **two directions only** (EAST and SOUTH) – unless it is **backtracking**.

**Here are a sample maze and its solution:**



## Finding a solution:

To find a solution, `MazeSolver` will explore the maze starting at position [0][0] and proceeding by moving either SOUTH or EAST whenever it encounters a hallway character ( _ ), if instead it encounters a wall character ( * ) it must either explore another direction or **backtrack** if it cannot move any further.
You will implement the backtracking mechanism by using a **stack**.
So finding a path to the exit will proceed as follows:

```
Push on the stack all positions reachable by moving one step SOUTH and
one step EAST from the point on which you are standing ([0][0]).

While the stack is not empty
   If your current position is an exit ($)
      Congratulations!
   Else if you are at an intersection
      Push  on the stack all positions reachable by moving one step
      SOUTH and one step EAST from the point on which you are standing.
      Mark you current position as PATH ( > ) on the solution
      Move forward by setting the current position to the one at the
      top of the stack.
  Else if you cannot move forward
      Mark your current position as VISITED (X) on the maze
      Mark your current position as BACKTRACKED (@) on the solution
      Pop the stack //BACKTRACK STEP
      If the stack is not empty
            Move forward by setting the current position to the one at
            the top of the stack.
      Otherwise
            Print “This maze has no solution.”
```

## Review – reading the input:

In C++ to read input from a file you need a file stream

```
#include <fstream>
```

Since we are only reading input you can use an `ifstream` object. Once you have an input file stream object (say you call it `in_stream`) you use it as you would `cin`:

```
in_stream >> my_variable;
```

Don't forget to:

- Open the stream before reading.
- Check that opening the stream did not fail before reading, and output (`cout`) an error message if it does fail.
- Close the stream after reading.

## Review – dynamically allocating 2-dimensional arrays:

**https://www.codeproject.com/Articles/21909/Introduction-to-dynamic-two-dimensional-arrays-in**

## The task: `MazeSolver` class

For this project you will implement the class `MazeSolver`.

This class will have **5 public members:**

- **A constructor**

```
MazeSolver(std::string input_file);
```

- **A destructor**

```
~MazeSolver();
```

- **A method that returns a boolean indicating whether the maze has been initialized**

```
bool mazeIsReady();
```

- **A method that finds a solution to the maze**

```
bool solveMaze();
```

- **A method that prints the solution to standard output (`cout`)**

```
void printSolution();
```

`MazeSolver` **private data members** are:

- **The number of rows in the maze**

```
int maze_rows_;
```

- **The number of columns in the maze**

```
int maze_columns_;
```

- **A boolean indicating whether the maze has been properly initialized**

```
bool maze_ready;
```

- **A 2D array representing the maze**

```
char** maze_;
```

- **A 2D array representing the solution**

```
char** solution_;
```

- **A stack used for backtracking** (you will use the STL stack here, you will use its public members `backtrack_stack_.push()`, `backtrack_stack_.pop()` and `backtrack_stack_.top()` ).

```
std::stack<Position> backtrack_stack_;
```

`MazeSolver` has several helper functions that will be discussed below along with the public method they help (the **helper functions are private**).

`MazeSolver` will make use of a struct `Position` to represents a position on the maze (namely a row and a column), and, to make your code more readable, an enum to represent the directions in which the solver can move on the maze (SOUTH, EAST).

```
enum direction {SOUTH, EAST};

struct Position
{
    int row;
    int column;
};
```

You should **break down the work** (and thoroughly test and debug **incrementally**) as follows:

1. Implement (debug and test) the constructor (and its helpers).
2. Implement (debug and test) the destructor.
3. Implement `mazeIsReady`
4. Implement (debug and test) `solveMaze` (and its helpers).
5. Implement (debug and test) `printSolution`.

**Let's look at these one by one:**

## 1. Implement (debug and test) the constructor (and its helpers).

MazeSolver has only one parameterized constructor which takes the name of the input file as its parameter. This constructor is responsible for several things:

- If the input cannot be read, it will output "Cannot read from *input_file_name*".

OTHERWISE:

- Read from the input the number of rows and columns and initialized the corresponding data members
- Initialize the maze matrix to the correct number of rows and columns
- Fill the maze with characters read from the input
- Initialize the solution matrix and copy characters from maze to it
- Set the maze_ready flag to true
- Close the input stream

To do this, the constructor will rely on the following helper functions (**make sure you implement and test these incrementally**)

```
void initializeMaze(int rows, int columns); //called by constructor
void fillMaze(std::ifstream& input_stream); //called by constructor
void initializeSolution(); //called by constructor
void copyMazetoSolution(); //called by initializeSolution

(pre and post conditions for these functions specified on the
interface)
```

## 2. Implement (debug and test) the destructor.

The destructor is responsible for deallocating the memory that has been dynamically allocated to the two matrices (`maze_` and `solution_` ).

Keep in mind that there could be instances in which the `maze_` and `solution_` were never allocated (e.g. if the input cannot be read), so the destructor needs to check that.

## 3. Implement `mazeIsReady`

trivial

## 4. Implement (debug and test) `solveMaze` (and its helpers).

solveMaze will implement the pseudocode at the top of this document (repeated here with a little more detail):

```
While the stack is not empty
   If your current position is an exit ($)
      Print "Found exit!!!" and return true
   Else if the current position is extensible
      Push all positions reachable by moving one step SOUTH and one
      step EAST from the point on which you are standing on the stack.
      Mark you current position as PATH ( > ) on the solution
      Move forward by setting the current position to the one at the
      top of the stack.
  Else (if you cannot move forward)
      Mark your current position as VISITED (X) on the maze
      Mark your current position as BACKTRACKED (@) on the solution
      Pop the stack //BACKTRACK STEP
      If the stack is not empty
            Move forward by setting the current position to the one at
            the top of the stack.
        Else (if the stack is empty)
            Print "This maze has no solution." and return false
```

`solveMaze` will make use of the helper function `bool extendPath(Position current_position);`
This method will take the current_position and push on the stack all positions that are extensible (that it can "move into") from the current position.
The pseudocode for `extendPath` is:

```
   If your current position is extensible in direction SOUTH
      Push the position extending SOUTH onto the stack
       extended is true
   If your current position is extensible in direction EAST
      Push the position extending EAST onto the stack
       extended is true
   Return extended
```

Keep in mind that, although it would still be correct to invert the order here, for this project **YOU MUST EXTEND FIRST SOUTH THEN EAST** to get full credit.

`extendPath` will use helper functions:

```
Position getNewPosition(Position old_position, direction dir);
bool isExtensible(Position current_position, direction dir);
```

`(again, refer to the interface for pre and post conditions)`

A position **is extensible** if from the current position you can move either SOUTH or EAST by moving into a position marked as ' _ ' or ' $ ' on the `maze_`. Make sure you keep in mind the boundaries of the matrices here and don't try to access indices outside of the `maze_`

### 5. Implement (debug and test) `printSolution`.

Print (`cout`) "The solution to this maze is:" followed by the solution on a new line.
To output the solution simply loop through `solution_` and print every character separated by a space ( ' ' )
Print each row on a new line.

## Usage:

You can test your program with the following `main` function. Your submission will be tested on multiple input files, not only the ones provided on blackboard.

```
#include <iostream>
#include "MazeSolver.h"

int main() {

    MazeSolver solver("input.txt");

    if(solver.mazeIsReady())
    {
        solver.solveMaze();

        solver.printSolution();
    }

    return 0;
}
```

## Submission:

For this project you will **submit `MazeSolver.cpp` only (1 file)**.
You can find the interface (`MazeSolver.h`) and some test input files on blackboard under CourseMaterials/Project4.
**Your project must be submitted on Gradescope. The due date is Tuesday November 6 by NOON (12pm). No late submissions will be accepted.**

I strongly encourage you to **START EARLY!!!!!**
Read this description and the interface (MazeSolver.h), over and over and over, until you have a good understanding of the problem, how it has been translated (the representation), the technique of finding a solution by backtracking via stack. Once you have a clear understanding of the design, proceed to incremental implementation and testing.
You will not be able to complete this project successfully if you leave it for the last week (not to mention days).

## Have Fun!!!!!