RMIT UNIVERSITY

COSC1076 | Semester 1 2020
Advanced Programming Techniques

Assignment 1
Maze Solver

Weight: 15% of the final course mark.
Silence Policy: From 5.00pm, Friday 3 April 2020 (Week 5)
Due Date: 11.59pm, Sunday 5 April 2020 (Week 5)
Submission Instructions: See Canvas Instructions to submit through Canvas.
Learning Outcomes: This assignment contributes to CLOs: 1, 2, 3, 4, 6

Change Log

1.0
- Initial Release

1 Introduction

1.1 Summary

In this assignment you will implement a version of ***Trémaux's algorithm*** for solving simple mazes. The algorithm in this assignment has a slight variation, so make sure you check the details here!

In this assignment you will:

- Practice the programming skills covered throughout this course, such as:
  - Pointers
  - Dynamic Memory Management
  - Provided API
- Correctly Implement a pre-defined API
- Implement a medium size C++ program
- Use a prescribed set of C++11/14 language features

This assignment is marked on three criteria, as given on the Marking Rubric on Canvas:

- **Milestone 1:** Writing Tests
- **Milestone 2-4:** Implementation of the Maze Solving Task
- **Style & Code Description:** Producing well formatted, and well documented code.

1.2 Relevant Lecture/Lab Material

To complete this assignment, you will requires skills and knowledge from lecture and lab material for Weeks 2 to 4 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Thus, do the work you can initially, and continue to build in new features as you learn the relevant skills.

! For Assignment 1, you may only use C++ languages features and STL elements that are covered in class.

1.3 Academic Integrity and Plagiarism

! CLO 6 for this course is: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the RMIT Academic Integrity Website.

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software.

2 Background

A maze is a very typical puzzle. To solve a maze you must find a path that connect the entry (starting) and exit (ending) points of the maze.

In this assignment, we will represent a simple 2D maze as a grid of ASCII characters. For example:

```
==========
S........=
========.=
=......=.=
==.=.=...=
=..=..=.==
=.===....=
=..==.==.=
=====E====
```

Aspects of the maze are represented by different symbols:

Symbol	Meaning
= (equal)	Wall or Obstacle within the maze. The robot cannot pass obstacles
. (dot)	Empty/Open Space.
`S`	The starting point (entry) of the maze.
`E`	The ending point (exit) of the maze.
* (asterisks)	Used to show the solution of the maze, which is a trail of *breadcrumbs*

The solution to the above maze can be shown as:

```
==========
S********=
========*=
=...***=*=
==.=*=***=
=..=**=.==
=.===*...=
=..==*==.=
=====E====
```

Each location in the maze (including the maze edges) is indexed by a cartesian `(x,y)` co-ordinate. The top-left corner of the maze is always the co-ordinate `(0,0)`, the x-coordinate increases right-wards, and the y-coordinate increases down-wards. For the above maze, the four corners have the following co-ordinates:

```
(0,0) . . (10,0)
  .           .
  .           .
(0,10) . . (10,10)
```

Finally, we also define 4 cardinal directions, so we can talk in terms of moving/walking about the maze.

```
      north
        ^
west <- | ->  east
        v
      south
```

Mazes can be designed in different ways. For the purposes of this assignment we will use mazes where:

1. There is one starting and one ending point.
2. The maze is *always* surrounded by walls, except for the starting/ending points.
3. The maze *only* contains *junctions* and *corridors*.

In this assignment you will implement a *simplified version of Trémaux Algorithm* for these types of mazes.

> **!** **You MUST implement this algorithm.** While there are many ways to solve a maze, y must implement this algorithm. If you don't, you will receive a NN grade.

2.1 Simplified Trémaux Algorithm.

This algorithm let us simulate "walking" about the maze leaving a trail of *breadcrumbs* behind us as we go. Along the way we can encounter dead-ends, so we will have to backtrack. As we back-track we will make some of the breadcrumbs "*stale*". Once we reach the end (goal), our trail of "good" (fresh/non-stale) breadcrumbs will be the path for solving the maze.

> **!** While a maze might have multiple solutions, this algorithm will only find *one* solution. It might not be the best or optimal, but it will be the same solution each time.

The basic premise of the algorithm is:

1. We "walk" along a corridor, dropping a breadcrumb at each location
2. When we hit a junction, we head down a corridor that we haven't been down before (we know this from our breadcrumbs!)
3. If we reach a dead-end, then we backtrack along our trail of breadcrumbs, marking these previously laid breadcrumbs as "stale", until we get back to a junction.
4. Then we choose a different path to head down
5. This keep on going until we reach the end of the maze.

The following is a more explicit form of this algorithm in *pseudocode*:

Pseudocode for the Simple Maze Solving

```
1  Let M be the maze
2  Let T be the trail of breadcrumbs that we leave
3  Let S and E be the starting and ending points of the maze respectively
4  Let b be track our current place in the maze, and set b = S
5  repeat
6      if The trail has no bredcrumb for our current location, b then
7          Add a breadcrumb for b to the end of the trail, that is T.addback(b)
8      end
       // Look for a place to go next
9      In order of north, east, south, west from b find the first location l, such that:
10     if l is empty and we havent left a breadcrumb in the trail T then
11         Move from b to l. That is, b = l
12     else
           // We need to Backtrack, as there is no-where we can go
13         Mark b (current breadcrumb) as "stale". (Make sure to update this within T)
14         Find the last good breadcrumb in the Trail T, call it b_good.
15         Go to this breadcrumb, that is b = b_good.
16     end
17 until We reach the end, that is b == E
```

On Canvas, you will find a short video stepping through this algorithm. In particular, it goes over the importance of *maintaining the order of the breadcrumb trail*. This is crucial to the backtracking. If we keep the breadcrumb trail in the right order, then we can figure out how to backtrack, by searching *backwards* from the *end* of the trail. This is because the trail is the *exact order* in which we have explored the maze. So the *reverse* of the trail, is the order to go back the way we came.

3 Task

The task for this assignment is to write a full C++ program that:

1. Reads in a 20x20 maze from standard-input (`std::cin`).
2. Solves the maze (using the Simplified Trémaux Algorithm in Section 2.1).
3. Prints out the solution to the maze to standard output (`std::cout`).
4. (Optionally) prints out walking/navigation directions (that you could follow to walk through the maze).

For the purposes of this assignment (except for Milestone 4), ***you may assume*** that the maze is ***always*** a fixed size of 20x20.

This assignment is divided 4 Milestones. To receive a PA grade, you only need to complete Milestones 1 & 2. To receive higher grades, you will need to complete Milestones 3 & 4.

> **!** Take careful note of the Marking Rubric on Canvas. Milestones form a sequence. For example, if you attempt Milestone 3, but your Milestone 2 is buggy, you won't get any marks for Milestone 3.

3.1 Milestone 1: Writing Tests

Before starting out on implementation, it is good practice to write some tests, so that you can know if your implementation is 100% correct. We are going to use I/O-blackbox testing. That is, we will give our program a maze to solve (as Input), and then test that the program's Output is what we expect it to be.

This a test consists of two text-files:

1. `<testname>.maze` - The *input* maze for the program to solve
2. `<testname>.out` - The *expected output* which is the solution to the maze.

A test *passes* if the output of your program *matches* the expected output.

You should write enough tests so that, if your program passes all of your tests pass, you are confident that your program is 100% correct.

> **!** Recall that you can "run" a test, by I/O redirection, and *capturing* your programs output:
> `./assign1 <testname.maze >program.out`
> Then you can test if the *actual* and *expected* outputs match with the `diff` command:
> `diff program.out testname.out`

3.2 Milestone 2: Maze Solving

Even for a relatively small algorithm, it is important to have the right *design* for our programs, and use appropriate *data structures* and *classes*. To help you in Assignment 1, we done this design for you to implement[1]. We will implement 3 classes, plus a `main` file:

- `Breadcrumb` class - to represent a single breadcrumb in the trail.
- `Trail` class - to and update the trail of breadcrumbs as we explore the maze.
- `MazeSolver` class - to encapsulate the maze solving algorithm.
- The `main` file that uses these classes, and does any reading/writing to standard input/output.

You are given these classes as header files in the starter code. You may add any of your own code, but you **must not modify** the provided class methods and fields. If you implement the class, you will have a working Milestone 2.

3.2.1 `Breadcrumb` Class

The `Breadcrumb` class represents one breadcrumb in our trail. It is a tuple `(x,y,stale)`, which is the `x-y` location of the breadcrumb in the maze, and whether breadcrumb is fresh or stale. It has accessor methods for this information and one method to change if the breadcrumb is fresh or stale.

[1] This won't be the case for Assignment 2, where you will have to make these decisions for yourself.

```
// Constructor/Desctructor
Breadcrumb(int x, int y, bool stale);
~Breadcrumb();

// x-co-ordinate of the particle
int getX();

// y-co-ordinate of the particle
int getY();

// Get if the Breadcrumb is stale. (If false, it is good/fresh)
bool isStale();

// Mark this Breadcrumb as stale (or not)
void setStale(bool stale);
```

3.2.2 `Trail` Class

The `Trail` class stores a trail of breadcrumbs *in the correct order*, using an *array* of breadcrumb objects.

```
// Trail of breadcrumb objects
// You may assume a fixed size for M1 & M2
Breadcrumb* breadcrumbs[TRAIL_ARRAY_MAX_SIZE];

// Number of breadcrumbs currently in the trail
int length;
```

Remembers, since it's an array we also need to track the number of breadcrumbs in the trail.

> ! You *must* implement the `Trail` class using an *array*.

The constant `TRAIL_ARRAY_MAX_SIZE` is the maximum number of breadcrumbs that can be in any trail. For this milestone, we can assume no trail will every get longer than this maximum size. This constant is given in the `Types.h` header file.

```
#define MAZE_DIM                20
#define TRAIL_ARRAY_MAX_SIZE    (MAZE_DIM * MAZE_DIM)
```

A Trail has the following methods:

```
// Constructor/Desctructor.
Trail();
~Trail();

// Number of elements in the Trail
int size();

// Get a pointer to the i-th trail element in the array - useful for making breadcrumbs stale.
Breadcrumb* getPtr(int i);

// Add a COPY trail element to the BACK of the trail.
void addCopy(Breadcrumb* trail);

// Check if the trail contain a breadcrumb at the location
bool contains(int x, int y);
```

These methods let you add breadcrumbs to the trail, check if a location in the trail already has a breadcrumb, and get a pointer to a breadcrumb in the trail which is useful for making breadcrumbs stale. Be aware, that the Trail class has full control over all breadcrumb pointers that are stored in the array. Thus, if breadcrumbs are removed from the array you must remember to "delete" the `Breadcrumb` object.

3.2.3 `MazeSolver` Class

The `MazeSolver` class *encapsulates* the implementation of the algorithm to solve a maze. This will make use of the Trail and Breadcrumb classes. It has two main components:

1. Construct a solution to a maze
2. Get the solution to a maze

```
// Constructor/Destructor
MazeSolver();
~MazeSolver();

// Solve the maze
void solve(Maze maze);

// Get a COPY of the solution
Trail* getSolution();
```

This uses a custom data type `Maze`, which is given in the `Types.h`. It is a 2D array of characters that represents a maze using the format in Section 2. It is a fixed size, because we assume the size of the maze is known.

```
// A 2D array to represent the maze or observations
// REMEMBER: in a maze, the location (x,y) is found by maze[y][x]!
typedef char Maze[MAZE_DIM][MAZE_DIM];
```

It is very important to understand the `Maze` type. It is defined as a 2D array (given below). If you recall from lectures/labs, a 2D array is indexed by *rows* then *columns*. So if you want to look-up a position `(x,y)` in the maze, you find this by `maze[y][x]`, that is, first you look-up the `y` value, *then* you look-up the `x` value.

The `solve` is given a maze, and *finds* a solution using the algorithm in Section 2.1.

```
void solve(Maze maze);
```

This solution is the trail of breadcrumbs from the starting to the ending location and is stored in a private field:

```
// Trail of breadcrumbs from the start to end
Trail* solution;
```

Importantly, the `solve` method ***must not*** modify the maze it is given!

The second method of `MazeSolver` returns a ***copy*** of the exact trail of breadcrumbs from the starting to the ending point of the maze. Be aware that this is a *deep copy* of the trail. This means you must also copy each breadcrumb in the trail!

```
// Get a DEEP COPY of the solution
Trail* getSolution();
```

3.2.4 `main` file

The final step of Milestone 2 is to write a `main` file that:

1. Reads in a maze from standard input
2. Solves the maze
3. Gets the solution to the maze
4. Outputs the maze (with the solution) to standard output

The starter code gives you the outline of this program. It has two functions for you to implement that read in the maze and print out the solution.

```
// Read a maze from standard input.
void readMazeStdin(Maze maze);

// Print out a Maze to standard output.
void printMazeStdout(Maze maze, Trail* solution);
```

Some hints for this section are:

- You can read one character from standard input by:

  ```
  char c;
  std::cin >> c;
  ```

- If you use the above, it *ignores all whitespace* including newlines. This will work since we know the size of the maze.
- When printing the maze, you might find it easier to add update the maze with solution first, and then print out the whole maze.

3.3 Milestone 3: Walking Directions

This is an *extension* Milestone. You only need to attempt this if you seek a *DI* grade for this assignment.

It would be nice to generate a list of "walking" directions that you could follow if you were actually travelling through the maze. These are given as a series of cardinal directions.

For Milestone 3, after your program shows the solution to the maze, it should print out the walking directions. Each direction is printed in order one-at-a-time, each on a new line (as below).

Make sure *update* your tests cases from Milestone 1 and add in walking directions!

For example, if we use the maze and solution from Section 2, the walking directions would be:

```
east
east
east
east
east
east
east
east
south
south
south
west
west
north
west
west
south
south
east
south
south
south
```

3.4 Milestone 4: Dynamic Array Allocation

This is an *extension* Milestone. You only need to attempt this if you seek a *HD* grade for this assignment.

For Milestones 1 - 3, we assume that the maze is *always* of a fixed size (`20x20`). This means, that for the `Maze` data type and the `Trail` class we could define the size of the arrays before-hand.

For this Milestone, you must modify your implementation to use a Maze of *any rectangular* size. To do this, you will need to modify a number of aspects of your implementation to *dynamically* allocate memory for 1D and 2D arrays used in this assignment. You will need to consider the following modifications:

- Change the type of `Maze` to a generic 2D array:

  ```
  typedef char** Maze;
  ```

  The `milestone4.h` file in the starter code has a sample method to help you dynamically allocate memory for a 2D array.
- Change the type of the field `breadcrumbs` in the `Trail` class to a generic 1D array of pointers:

  ```
  Breadcrumb** breadcrumbs;
  ```
- Create memory as necessary for the maze and trail of breadcrumbs.
- When reading in the maze, you will need to be able to spot newline characters. You can't do this if you follow the suggestion for Milestone 2. Instead you will need the `get` method of `std::cin`:

  ```
  char c;
  std::cin.get(c);
  ```
- Be *efficient*. Avoid allocating large amounts of memory that won't be used. You *could* to this by just making the `MAZE_DIM` very big, but most mazes won't be huge and this is a waste.

For this milestone *only*:

- You cannot modify the given class methods. If you need to change the parameters, you can add/overload methods and constructors as you see fit.
- You may modify the private fields. In fact, as outlined above you will need to modify some of these.

3.5 Documentation, Style and Code Description

This applies for all Milestones 2, 3 and 4.

Making sure your code is 100% correct is very important. Making your sure code is understandable is equally important. The third criteria on which your assignment is marks determined if your code:

- Follows the Course Style Guide, as given on Canvas. This includes not using any banned elements.
- Is well documented, with clear comments.

Finally, you need to provide a ***short description*** (at the top of your `main` file) that

- Describe (briefly) the approach you have taken in your implementation
- Describe (briefly) any issues you encountered
- Justify choices you made in your software design and implementation
- Analyse (briefly) the efficiency and quality of your implementation, identifying both positive and negative aspects of your software

If you completed Milestones 3 or 4, this code description should include what you had to do for these milestones.

For Assignment 1, you may only use C++ languages features and STL elements that are covered in class. Using unapproved features will results in a reduced grade. For example, you must use an array of objects to implement the `Breadcrumb` class.

4 Getting Started

4.1 Starter Code

We have provided starter code to help you get underway. This includes header and code files for the classes described in the Task above, as well as the `Types.h` and `main` files. There is also a header file `milestone4.hpp` for Milestone 4. Ignore this if unless you attempt this milestone. Take your time to review these files.

To compile your program, you will need to use a command similar to the following:

```
g++ -Wall -Werror -std=c++14 -O -o assign1 Breadcrumb.cpp Trail.cpp MazeSolver.cpp main.cpp
```

4.2 Suggestions for starting Milestone 1

The starter code also contains a folder with one sample test case for Milestone 2. This should give you an idea of how your tests should be formatted.

Obviously this is not enough, however, it should give you a good starting point for testing your code.

4.3 Suggestions for starting Milestone 2

Part of the learning process of the skill of programming is devising how to solve problems. In this assignment, the problem solving is turning an *algorithm* and *pseudocode* into a *complete functioning program*.

This process involves completing small tasks one-at-a-time. For this assignment we recommend that you implement things in the following order:

1. In the `main` file, read in a maze from standard input and print out this maze (unmodified)
2. Implement the `Breadcrumb` class
3. Implement the `Trail` class
4. Implement the `MazeSolver` class
5. Update the `main` file to use the `MazeSolver`

Testing is also an important part of this process. The tests you need to write for Milestone 1 test *your whole program*. This has a problem, because this means you have to write the whole program first. However, you can write small programs to ***test your program as you go***.

> **!** This next bit is super important to helping you!

Have a careful look at the `main` file given in the Starter Code. It has a couple of examples of what you can do to test that your `Breadcrumb` and `Trail` class are working as you develop them. Of course, once you finish the assignment, you can delete this testing code. The important part is this ***lets you test small parts of your program as you go*** rather than waiting until the end and just hoping the whole thing works.

4.4 Running Milestone 1 Tests

As a reminder, you can run a test as below. Recall that is uses the `diff` program to compare the *actual* and *expected* output of your program.

```
./assign1 <testname.maze >actual.out
    diff actual.out testname.out
```

5 Submission Instructions & Marking

To submit, follow the instructions on Canvas for Assignment 1.

5.1 Notes on Marking

> **!** You must not modify the elements provided starter code (including the methods and fields) unless otherwise specified. If you incorrectly implement the assignment, even if it "passes the tests" you will received a reduced grade.

This assignment is designed with three "brackets" of competition:

- If you do a good job on Milestone 1 & 2, then your final mark will be a CR. This will mean you have a CR in all three rubric categories
- If you do a good job for Milestone 3, then you mark will be a DI, getting a DI in all rubric categories
- Similarly, if do a good job for Milestone 4, your mark will be a HD.

The purpose of this is for you to focus on successfully completing each Milestone *one-at-a-time*. You will also notice there are not many marks for "trying" or just "getting started". This is because this is an *advanced* course. You need to make *significant* progress on solving the task in this assignment before you get any marks.

5.2 Silence Period

A silence policy will take effect from **5.00pm, Friday 3 April 2020 (Week 5)** before the assignment is due. This means no questions about this assignment will be answered, whether they are asked on the discussion board, by email, or in person.

Make sure you ask your questions with plenty of time for them to be answered. There may not be time to answer questions which are asked too close to the silence deadline. Be aware, that as the lecture finishes 4.30pm on Friday, it is unlikely that there will be time to answer many questions after the end of the lecture.

5.3 Late Submissions & Extensions

A penalty of 10% per day is applied to late submissions up to business 5 days, after which you will lose ALL the assignment marks. Extensions will be given only in exceptional cases; refer to Special Consideration process.

Special Considerations given after grades and/or solutions have been released *will automatically result in an equivalent assessment in the form of a test*, assessing the same knowledge and skills of the assignment (location and time to be arranged by the course co-ordinator).

The due date is a *hard* deadline. Any late assignment will incur a penalty, no matter how close after the deadline it is submitted. Thus, we recommend that you *do not* leave submitting until the last possible moment.