



**Zeid Kootbally**  
University of Maryland  
College Park, MD

[zeidk@umd.edu](mailto:zeidk@umd.edu)

# Final Project

---

ENPM809Y : Fall 2019  
Due **Monday, May 4, 2020**

## Contents

<b>Introduction</b>	<b>2</b>
<b>Robots</b>	<b>2</b>
<b>Maze</b>	<b>5</b>
<b>Path Planning Algorithm</b>	<b>7</b>
<b>The Micromouse Simulator</b>	<b>8</b>
<b>The Main File</b>	<b>13</b>
<b>Pseudo-code</b>	<b>13</b>
<b>Documentation</b>	<b>14</b>
<b>Packaging The Project</b>	<b>14</b>

## Introduction

- **Scenario**

- A robot navigates through a maze to reach the center of the maze.

- **Objectives**

- Write a program that computes a path from the current position (S) of a robot in a maze to the center of the maze (G).
- Once a path is generated, the robot is tasked to follow the path.
- The micromouse simulator (<https://github.com/mackorone/mms>) will be used to visualize the result.
- Programmatically, the robot is from a C++-derived class.
- The program exits when either the robot reaches G or there is no solution from S to G.

## Robots

- Two types of mobile robotic arms are available.
- The first robot belongs to the C++ class **LandBasedWheeled** (see example of this type of robot in Figure 1a).
- The second robot belongs to the C++ class **LandBasedTracked** (see example of this type of robot in Figure 1b).
- Both classes derive from the C++ base class **LandBasedRobot**.

- **Class LandBasedRobot**

- Attributes:

- `std::string name_`: Name of the robot.
- `double speed_`: Driving speed of the robot.
- `double width_`: Width of the base of the robot.
- `double length_`: Length of the base of the robot.
- `double height_`: Height of the base of the robot.
- `double capacity_`: Payload of the arm.
- `int x_`: X coordinate of the robot in the maze.



(a) Husky



(b) LT2-F Tactical Robot

Figure 1: Robots used in the maze problem.

- `int y_`: Y coordinate of the robot in the maze.
- `char direction_`: Direction that the robot is facing in the maze. The different possibilities are 'N' (north), 'E' (east), 'W' (west), 'S' (south).
  - It is a good idea to store the different directions in a C++ structure. Take a look at C++ `struct`.
- Methods:
  - `char GetDirection()`: Get the direction of the robot in the maze.
  - `void MoveForward()`: Move the robot forward.
  - `void TurnLeft()`: Rotate the robot 90° counter-clockwise.
  - `void TurnRight()`: Rotate the robot 90° clockwise.
  - \* **Note**: `void TurnLeft()` and `void TurnRight()` only rotate the robot. To move the robot again, you will need to use `void MoveForward()`.
- Class **LandBasedWheeled**
  - Attributes:
    - `int wheel_number`: Number of wheels mounted on the robot.
- Class **LandBasedTracked**
  - Attributes:
    - `std::string track_type`: Type of track mounted on the robot.

- The instructions for the final project are a bit different from the ones provided for RWA-3. Some methods/attributes have been removed/renamed.

#### Program Instructions

- C++ classes
  - Using single `public` inheritance to create the following classes:
    - \* **LandBasedRobot** is an `abstract class`.
    - \* **LandBasedWheel** is a `concrete class` and derives from **LandBasedRobot**.
    - \* **LandBasedTrack** is a `concrete class` and derives from **LandBasedRobot**.
  - Attributes of the base class must be declared `protected`.
  - Attributes of the derived classes must be declared `private` or `protected`.
  - All methods in based and derived classes must be declared `private` or `protected`.
  - For each class:
    - \* Method prototypes go in the class definition (.h)
    - \* Method definitions go in the class implementation (.cpp)
  - **Namespace**: Wrap all your classes with the namespace `fp` (`fp` stands for Final Project).

- Constructors:
  - You can have overloaded constructors and you can create your own constructors.
  - Make sure to call the base class constructors from the derived class constructors to initialize the base class attributes.
  - Your constructors must take care of initializing all the attributes of your class.
- Accessors and mutators:
  - If you need them, write accessors (getters) and mutators (setters) directly in class definitions (.h)
- Copy constructor for each class
  - Write your own deep copy constructor for each relevant class if necessary.
- Dynamic polymorphism:
  - Dynamic binding (dynamic polymorphism) requires 3 elements:
    1. Inheritance.
    2. Pointer or reference object to base class: If you use pointers to the base class, they have to be **smart pointers**. You should figure out which type of smart pointers you need in this case (unique, shared, or weak).
    3. Virtual methods: Best practice is to include **virtual** and **override** keywords in derived classes.
  - \* **Note:** If you are using method prototypes (which you should be doing), **virtual** and **override** are placed in the method prototypes, not in the method definitions.

```
//--lanbasedtracked.h
class LandBasedTracked : public LandBasedRobot{
public:
    virtual void MyMethod() override;
};

//--landbasedtracked.cpp
void fp::LandBasedTracked::MyMethod()
{
    /*code*/
}
```

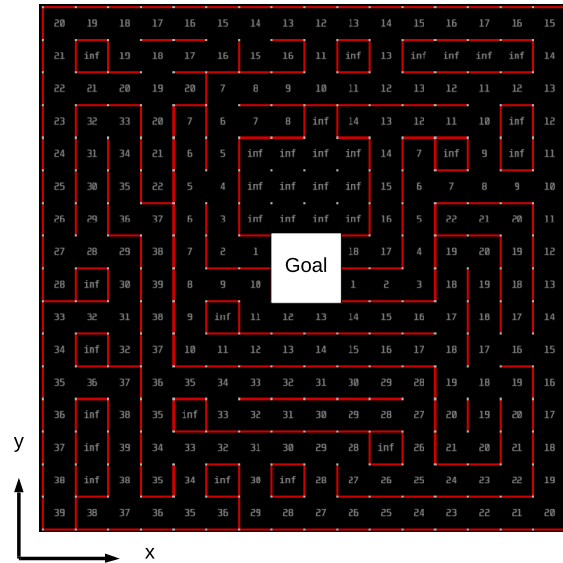


Figure 2: Goal and coordinate system for all the mazes.

## Maze

- All mazes have the same dimensions  $16 \times 16$  cells.
- All mazes have walls around their perimeter.
- The robot has no knowledge of walls (even the ones around the perimeter) beforehand.
- Walls are discovered only when the robot drives through cells that have walls.
  - You need to use methods from the class API to check for walls (see API class).
- The coordinate system for all the mazes is displayed in Figure 2.
  - $x \in [0, 15]$  and  $y \in [0, 15]$
- Any of the 4 cells that are in the center of the maze constitutes the goal (G) the robot needs to reach (see Figure 2). The coordinates of the goal cells are consistent among mazes: (7,7), (7,8), (8,7), and (8,8).
- When the simulation starts, the robot will always start at position (0,0) and will always face north.

### Maze Instructions

1. You will need to create a **Maze** class for this project.
2. You need to read the maze loaded by the micromouse simulator and store it in your program with a structure of your choice. Since all mazes have the same dimensions an array structure would be enough.
3. Some examples of methods for the **Maze** class are:
  - Get the dimension of the maze.
  - Check if there is a wall between two adjacent cells.
  - Set a wall between two adjacent cells.
4. One way to represent a maze is to use a specific value for adjacent cells that are not separated by a wall and a different value if there is a wall between them.
  - **Hint:** Since the robot has no knowledge of walls at the beginning, you should build an original maze ( $16 \times 16$ ) where adjacent cells are connected (i.e., no wall between these cells).
  - When you run your program and the robot encounters walls (left, right, or front), just update your maze representation to include walls between cells.

## Path Planning Algorithm

- You need to implement depth-first search (DFS) algorithm to generate a path from S to G.
- You have to explore the maze in the following order:
  1. down.
  2. right.
  3. up.
  4. left.

### Algorithm Class

- You have to create a C++ class **Algorithm** which contains the implementation for the DFS algorithm.
- Once a path is generated, this class will also be responsible for driving the robot through the maze using methods from the API class.
- **Note:** At the beginning of the simulation, all the walls are grayed out (i.e., unknown to the robot). When driving the robot through the maze from the generated path, use the method `setWall` from the **API** class (see next section) to change the color of the wall.

## The Micromouse Simulator

The final project uses an open source simulator (<https://github.com/mackorone/mms>) to visualize path planning algorithms.

The interaction between your program and the simulator can be performed using built-in methods from the API class. These methods can be found at <https://github.com/mackorone/mms> and the method prototypes can be found below.

### API Class Methods

```
int mazeWidth();/--return the width of the maze
int mazeHeight();/--return the height of the maze
bool wallFront();/--true if there is a wall in front of the robot, else false
bool wallRight();/--true if there is a wall to the right of the robot, else false
bool wallLeft();/--true if there is a wall to the left of the robot, else false
void moveForward();/--move the robot forward by one cell
void turnRight();/--turn the robot 90 degrees to the right
void turnLeft();/--turn the robot 90 degrees to the left
void setWall(int x, int y, char direction);/--display a wall in maze
void clearWall(int x, int y, char direction);/--clear a wall from maze
void setColor(int x, int y, char color);/--set the color of a cell
void clearColor(int x, int y);/--clear the color of a cell
void clearAllColor();/--clear the color of all cells
void setText(int x, int y, const std::string& text);/--set the text of a cell
void clearText(int x, int y);/--clear the text of a cell
void clearAllText();/--clear the text of all cells
bool wasReset();/--true if the reset button was pressed, else false
void ackReset();/--reset simulation
```

- Create (or copy from mms-cpp project) an API class (`api.h`) for these methods. An API is a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.
- The API class must be wrapped in the `fp` namespace.

### api.h

```
#pragma once
#include <string>

namespace fp{
    class API{
    public:
        static int mazeWidth();
        static int mazeHeight();
        static bool wallFront();
```



```

    static bool wallRight();
    static bool wallLeft();
    static void moveForward();
    static void turnRight();
    static void turnLeft();
    static void setWall(int x, int y, char direction);
    static void clearWall(int x, int y, char direction);
    static void setColor(int x, int y, char color);
    static void clearColor(int x, int y);
    static void clearAllColor();
    static void setText(int x, int y, const std::string& text);
    static void clearText(int x, int y);
    static void clearAllText();
    static bool wasReset();
    static void ackReset();
}; //--class API
} //--namespace fp

```

- The method definitions for the API class are provided below and will be included in api.cpp

api.h

```

#include "api.h"

#include <cstdlib>
#include <iostream>

int fp::API::mazeWidth() {
    std::cout << "mazeWidth" << std::endl;
    std::string response;
    std::cin >> response;
    return atoi(response.c_str());
}

int fp::API::mazeHeight() {
    std::cout << "mazeHeight" << std::endl;
    std::string response;
    std::cin >> response;
    return atoi(response.c_str());
}

bool fp::API::wallFront() {
    std::cout << "wallFront" << std::endl;
    std::string response;
    std::cin >> response;
    return response == "true";
}

```

```
bool fp::API::wallRight() {
    std::cout << "wallRight" << std::endl;
    std::string response;
    std::cin >> response;
    return response == "true";
}

bool fp::API::wallLeft() {
    std::cout << "wallLeft" << std::endl;
    std::string response;
    std::cin >> response;
    return response == "true";
}

void fp::API::moveForward() {
    std::cout << "moveForward" << std::endl;
    std::string response;
    std::cin >> response;
    if (response != "ack") {
        std::cerr << response << std::endl;
        throw;
    }
}

void fp::API::turnRight() {
    std::cout << "turnRight" << std::endl;
    std::string ack;
    std::cin >> ack;
}

void fp::API::turnLeft() {
    std::cout << "turnLeft" << std::endl;
    std::string ack;
    std::cin >> ack;
}

void fp::API::setWall(int x, int y, char direction) {
    std::cout << "setWall " << x << " " << y << " " << direction << std::endl;
}

void fp::API::clearWall(int x, int y, char direction) {
    std::cout << "clearWall " << x << " " << y << " " << direction << std::endl;
}

void fp::API::setColor(int x, int y, char color) {
    std::cout << "setColor " << x << " " << y << " " << color << std::endl;
}
```

```

void fp::API::clearColor(int x, int y) {
    std::cout << "clearColor " << x << " " << y << std::endl;
}

void fp::API::clearAllColor() {
    std::cout << "clearAllColor" << std::endl;
}

void fp::API::setText(int x, int y, const std::string& text) {
    std::cout << "setText " << x << " " << y << " " << text << std::endl;
}

void fp::API::clearText(int x, int y) {
    std::cout << "clearText " << x << " " << y << std::endl;
}

void fp::API::clearAllText() {
    std::cout << "clearAllText" << std::endl;
}

bool fp::API::wasReset() {
    std::cout << "wasReset" << std::endl;
    std::string response;
    std::cin >> response;
    return response == "true";
}

void fp::API::ackReset() {
    std::cout << "ackReset" << std::endl;
    std::string ack;
    std::cin >> ack;
}

```

- To use these methods in your own code, take a look at how it can be done below (excerpt of `landbasedtracked.cpp`).
- Since all the methods in the class **API** are public, you can directly call those methods in your code by preceding them with `fp::API::`:

`landbasedtracked.cpp`

```

void fp::LandBasedTracked::MoveForward(){
    std::cout << "LandBasedTracked::MoveForward is called\n";
    fp::API::moveForward();
}

```

- Figure 3 shows the fields that you need to fill for the simulator.

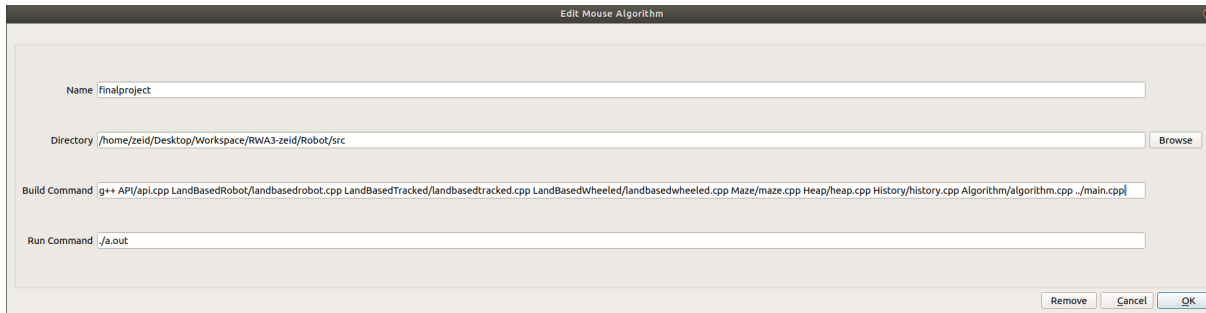


Figure 3: Fields to compile and run your program.

- **Name:** This is a user defined name that is for your algorithm, this can be any name.
- **Directory:** This path points to your **src** directory, which contains subdirectories for your classes.
- **Build Command:** Use the **g++** compiler followed by the relative paths (relative to the **src** directory) of all the **.cpp** files in your project.
- **Run Command:** **./a.out** is the default executable on Linux systems.
- **Important:** The simulator requires relative paths for included files.  
For instance:
  - If **main.cpp** needs to access the class **LandBasedTracked**, you will need to include it with `#include "LandBasedTracked/landbasedtracked.h"`.
  - If **landbasedwheeled.cpp** needs to access the class **LandBasedTracked**, you will need to include it with `#include "../LandBasedTracked/landbasedtracked.h"`.
- Finally, an example of project computing a path from a start position to a goal position can be found here <https://github.com/mackorone/mackalgo>. You can study this project to get an idea on how to accomplish the different tasks needed for the final project.

## The Main File

- An example of main file to run the program:

### Main File Example

```
int main(){
    std::shared_ptr<fp::LandBasedRobot> wheeled = std::make_shared<fp::LandBasedWheeled>("Husky");
    fp::Algorithm algo;
    algo.Solve(wheeled);

    return 0;
}
```

- You will need to pass a pointer to the base class to the methods that solve and drive the robot in the maze.

## Pseudo-code

- A pseudo-code for the final project is given below. The provided pseudo-code is just an example and obviously you can have a different pseudo-code.

```

initialize the maze (set walls around the perimeter);
color the center of the maze (4 goal cells);
initialize the robot (position and direction);
pick one of the 4 goals to reach;
while true do
    clear all tile color, except the goal cells;
    read all the walls around the robot (left, right, and front);
    get the current cell of the robot;
    generate a path from current cell to destination (DFS);
    if no path from current position to destination then
        unsolvable maze;
        false;
    end
    draw path in the maze with API::setColor(int,int,char);
    move the robot along the path with TurnLeft(), TurnLeft() and MoveForward();
    if robot reaches goal then
        success;
        false;
    else
        set new walls around robot;
    end
end

```

**Algorithm 1:** An example of pseudo-code for the final project.

## Documentation

Your project must be documented using Doxygen documentation. Document what makes sense (classes, struct, methods, etc). You will need to provide your Doxyfile when packaging your project before submission.

## Packaging The Project

### Instructions to Package the Project

1. Create a project named `Final-Project-Group#` (where `#` is your group number).
2. Create a folder for each class. Each one of these folders must be created inside the `src` directory. The structure should be:
  - `src/LandBasedRobot`
  - `src/LandBasedWheeled`
  - `src/LandBasedTracked`

- etc
3. Besides the C++ classes mentioned in this document, you are free to create any other classes, struct, methods, etc, as you see fit.
  4. Create a class in each directory. For example, the class **LandBasedRobot** will be created in the **src/LandBasedRobot** directory.
  5. Document your classes, methods, and attributes using Doxygen documentation.
  6. In your project, create a directory named **doc** and place your Doxyfile in it.
    - Example: For group 1, your Doxyfile will be placed in **Final-Project-Group1/doc/**
  7. Compress your project and correctly rename it if necessary
    - For instance, the compressed final project for Group 1 will be **Final-Project-Group1.zip**
  8. Upload it on Canvas.
  9. Good luck.