Zeid Kootbally
University of Maryland
College Park, MD

zeidk@umd.edu

Final Project

ENPM809Y : Fall 2019
Due **Monday, December 2, 2019**

Contents

Introduction

- **Scenario**
  - A robot navigates through a maze to reach the center of the maze.
- **Objectives**
  - Write a program that generates a path from the current position (`S`) of a robot in a maze to the center of the maze (`G`).
  - The robot that navigates the maze will always be a robot from a derived C++class
  - 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 display the result on the screen.
  - 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 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.

(a) Husky

(b) LT2-F Tactical Robot

Figure 1: Robots used in the maze problem.

    - `int` `x_`: X coordinate of the robot in the maze.
    - `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.
  - Methods:
    - `void` `SpeedUp(int)`: The robot can increase its speed, which is translated by the number of cells in the maze that the robot can traverse in each step.
- 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, 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 `protected`
  - All methods in based and derived classes must be declared `public`
  - 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.
  - Make sure to call the base class constructors in the derived class constructors.
  - 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.
- 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 MoveForward() override;
};

//--landbasedtracked.cpp
void fp::LandBasedTracked::MoveForward()
{
   /*Code that moves the robot forward*/
}
```

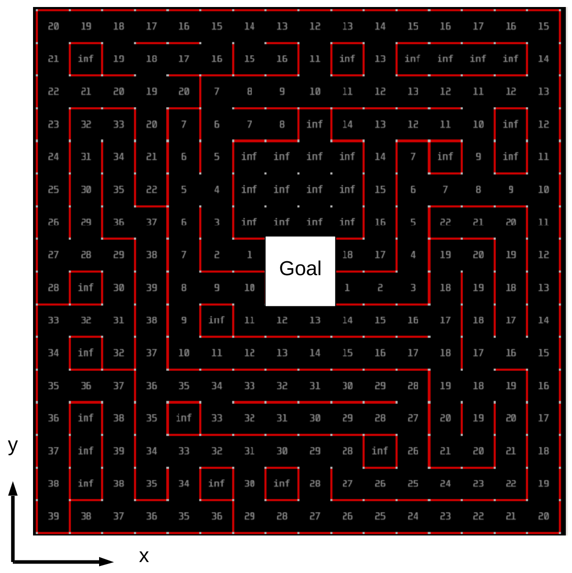


Figure 2: Goal and coordinate system for all the mazes.

Maze

- All mazes have the same dimensions 16×16 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.
- The coordinate system for all the mazes is displayed in Figure 2.
  - With $x \in [0, 15]$ 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)
  - (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 (arrays, `std::vector`, etc)
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
   - etc
4. The best 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 (16×16) 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 Algorithms

- You need to choose between 2 approaches to generate a path from `S` to `G`.
  - Deph-first search
  - Breadth-first search
- For both approaches, you need to explore the maze in the following order:
  1. down
  2. right
  3. up
  4. left

Deph-first Search (DFS)

- https://en.wikipedia.org/wiki/Depth-first_search
- Algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores **as far as possible** along each branch before backtracking.

Breadth-first Search (DFS)

- https://en.wikipedia.org/wiki/Breadth-first_search
- Algorithm for traversing or searching tree or graph data structures.
- It starts at the tree root (or some arbitrary node of a graph), and explores **all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level**.

Algorithm Class

- You will need to create a class **Algorithm** that contains either the code for BFS or DFS.
- Once a path is generated, this class will also be responsible for driving the robot through the maze.
- **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 another project which is open source. The micromouse simulator (https://github.com/mackorone/mms) is a small C++QT simulator that allows you to visualize path planning algorithms.
The interaction between your program and the simulator can be performed using built-in methods. These methods can be found at https://github.com/mackorone/mms and the method prototypes can be found below.

Micromouse Controller Methods

```
//--return the width of the maze
int mazeWidth();
//--return the height of the maze
int mazeHeight();
//--true if there is a wall in front of the robot, else false
bool wallFront();
//--true if there is a wall to the right of the robot, else false
bool wallRight();
//--true if there is a wall to the left of the robot, else false
bool wallLeft();

//--move the robot forward by one cell
void moveForward();
//--turn the robot ninty degrees to the right
void turnRight();
//--turn the robot ninty degrees to the left
void turnLeft();

//--display a wall at the given position
void setWall(int x, int y, char direction);
//--clear the wall at the given position
void clearWall(int x, int y, char direction);

//--set the color of the cell at the given position
void setColor(int x, int y, char color);
//--clear the color of the cell at the given position
void clearColor(int x, int y);
//--clear the color of all cells
void clearAllColor();

//--set the text of the cell at the given position
void setText(int x, int y, const std::string& text);
//--clear the text of the cell at the given position
void clearText(int x, int y);
//--clear the text of all cells
void clearAllText();

//--true if the reset button was pressed, else false
```

```
bool wasReset();
//--allow the robot to be moved back to the start of the maze
void ackReset();
```

- Create an API class that will only store 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.

api.h

```
#pragma once
#include <string>

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();
};
```

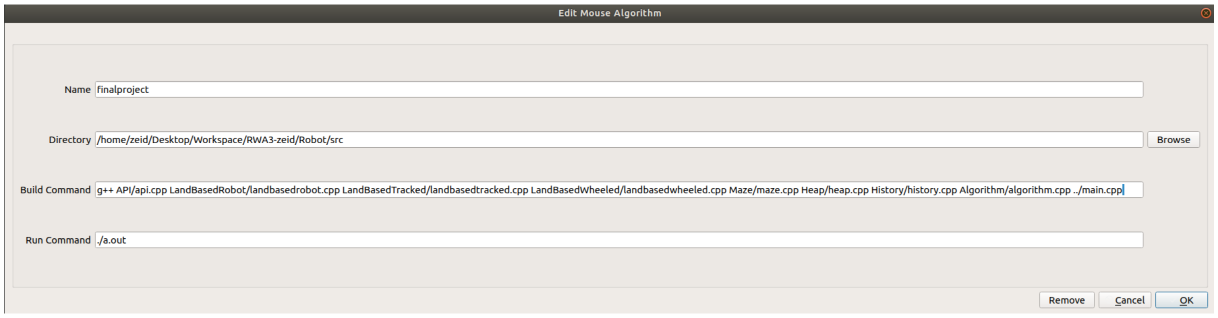
- The method definitions for the API class are provided below and will be included in api.cpp

api.h

```
#include "api.h"

#include <cstdlib>
#include <iostream>
```

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

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

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

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

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

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

void API::turnRight() {
   std::cout << "turnRight" << std::endl;
   std::string ack;
```

```
    std::cin >> ack;
}

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

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

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

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

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

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

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

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

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

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



Name: finalproject

Directory: /home/zeid/Desktop/Workspace/RWA3-zeid/Robot/src

Build Command: g++ API/api.cpp LandBasedRobot/landbasedrobot.cpp LandBasedTracked/landbasedtracked.cpp LandBasedWheeled/landbasedwheeled.cpp Maze/maze.cpp Heap/heap.cpp History/history.cpp Algorithm/algorithm.cpp ../main.cpp

Run Command: ./a.out

Figure 3: Fields to compile and run your program.

```
void 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 `API::`

landbasedtracked.cpp

```
#include "landbasedtracked.h"
#include "../API/api.h"
#include <iostream>

void fp::LandBasedTracked::MoveForward(){
   std::cout << "LandBasedTracked::MoveForward is called\n";
   API::moveForward();
   /*
   More code if needed
   */
}
```

- Figure 3 shows the fields that you need to fill for the simulator.
  - **Name**: This 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**: Unfortunately, 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 the shortest path using Dijkstra 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(){
   //--Replace the use of raw pointers with smart pointers
   fp::LandBasedRobot *wheeled = new fp::LandBasedWheeled("Husky");
   fp::Algorithm algo;
   algo.Solve(wheeled);

   delete 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;
initialize the robot (position and direction);
while true do
    clear all tile color;
    read all the walls;
    get the current cell of the robot;
    generate a path from current cell to destination (DFS or BFS);
    if no path from current position to destination then
        unsolvable maze;
        exit;
    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 destination then
        success;
        exit;
    else
        set new walls;
    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.