ASSIGNMENT ONE: CAR BOARD

ADVANCED PROGRAMMING TECHNIQUES – SEMESTER 1, 2018

SUMMARY

In this assignment, you will use your C programming skills to build an interesting game called **Car Board**. The rules for the game are simple: a car can move around inside a board; the player can move the car by entering a set of specific commands; and the car must stay within the board boundaries and must not hit the roadblocks.

In the following sections of this document, the details of your assignment are explained. Read them thoroughly and ask your questions from the teaching team. You are expected to understand every requirement explained in this document and implement all of them accordingly.

A start-up code is provided along with this document. *You **must** use this start-up in your assignment.* You are required to submit the exact files provided in the start-up code after completing the according to your tasks.

SUBMISSION

Total mark: 20% of the final mark

Assignment demo: During tutorial / lab sessions of week 4 and 5 *before mid-semester break* (22-28 Mar 2018)

Final submission: End of week 6 – Saturday, 14 April 2018 – **10:00 PM**

PLAGIARISM NOTICE

Plagiarism is a very serious offence. The minimum first time penalty for plagiarised assignments is zero marks for that assignment. Please keep in mind that RMIT University uses plagiarism detection software to detect plagiarism and that all assignments will be tested using this software. More information about plagiarism can be found here: http://www1.rmit.edu.au/academicintegrity/.

HOW TO COMPILE THE PROJECT

It is very important to pay attention to how this project needs to be compiled. There are multiple files that must be compiled together to form the executable of your assignment. Note the following issues in respect to compilation.

Your program must compile and work on the Unix machines of the school. You will not receive any marks for your submission otherwise. Demonstration on your laptops or different operating systems is not acceptable.

You must use the following command-line arguments to compile your project, and there should be no errors and warnings during the compilation:

```
gcc -ansi -pedantic -Wall -o CarBoard board.c carboard.c game.c helpers.c
player.c
```

For more information about `–ansi`, `-pedantic`, `-Wall` and `–o` parameters, refer to the lecture and tutorial material.

Do not use the `#include` pre-processor directive to include any `.c` file in your project. The `#include` directive should only be used for `.h` files.

Note that if you add a new `.c` file to your project, you will need to add the name of that `.c` file to the list of files in the above command.

You should separate your assignment files from other files in the system by creating a dedicated folder. In that case, you will be able to simplify the above command to the following, which compiles *all of the C files in the current directory (i.e. `*.c`) and build a single executable (i.e. `CarBoard`)*.

```
gcc -ansi -pedantic -Wall -o CarBoard *.c
```

DESCRIPTION OF THE PROGRAM

CORE REQUIREMENTS

REQ1: MAIN MENU

When the program starts, the following menu should be displayed.

```
./carboard

Welcome to Car Board
--------------------
1. Play game
2. Show student's information
3. Quit

Please enter your choice:
```

Your program should reject any invalid data. If invalid input is entered (for example: 10, c …), **you should simply ignore the input and show the prompt again**.

If user enters number 2, your program should print your name, your student number as below. Note that you should replace *<your full name>*, *<your student number>* and *<your email address>* sections with your real name, student number and student email address.

```
Welcome to Car Board
--------------------
1. Play game
2. Show student's information
3. Quit

Please enter your choice: 2

---------------------------------
Name: <your full name>
No: <your student number>
Email: <your email address>
---------------------------------

Please enter your choice:
```

If user selects menu number 3, your program should exit without crashing. This is specified in section 7 below.

If user selects menu number 1, the following list of commands will be displayed.

```
Welcome to Car Board
--------------------
1. Play game
2. Show student's information
3. Quit

Please enter your choice: 1

You can use the following commands to play the game:
load <g>
   g: number of the game board to load
init <x>,<y>,<direction>
   x: horizontal position of the car on the board (between 0 & 9)
   y: vertical position of the car on the board (between 0 & 9)
   direction: direction of the car's movement (north, east, south, west)
forward (or f)
turn_left (or l)
turn_right (or r)
quit
```

The *load* command is responsible to initialise the game board. More details about this command is available in section 3 below.

The *init* command is responsible to set the initialise values of the car's position and movement. More detail about this command is available in section 4 below.

The *forward* command is responsible to move the car one position forward according to the current direction of the car. The *turn_left (or l)* and *turn_right (or r)* commands are responsible to change the direction of the car's movement. See the details in section 5 below.

The *quit* command will exit the current game and return to the *main menu*. This requirement is defined in section 8 below.

It is important to note that the board (containing the blocks, the position and the direction of the car) must be displayed on the screen after it is *loaded*, *initialised*, moved *forward* and turned *left* or *right*. See the requirement for printing the board in section 2 below.

Note that you should print a statement that at each stage of the program you must also display the list of *currently acceptable commands* in the output. The acceptable commands are mentioned in each section below.

REQ2: DISPLAYING THE BOARD

This game has a 10x10 board. When the game starts (right after the **Play Game** command and before loading any board), the following board should be displayed on the screen.

```
|  |0|1|2|3|4|5|6|7|8|9|
|0|  |  |  |  |  |  |  |  |  |
|1|  |  |  |  |  |  |  |  |  |
|2|  |  |  |  |  |  |  |  |  |
|3|  |  |  |  |  |  |  |  |  |
|4|  |  |  |  |  |  |  |  |  |
|5|  |  |  |  |  |  |  |  |  |
|6|  |  |  |  |  |  |  |  |  |
|7|  |  |  |  |  |  |  |  |  |
|8|  |  |  |  |  |  |  |  |  |
|9|  |  |  |  |  |  |  |  |  |
```

Note that the board is completely empty. This is exactly how it should look like before any particular board is loaded in the next section.

At this stage of the program, only two commands are acceptable:

- ***load <g>***
- ***quit***

If user enters any other command, or an incorrect command, an error message (e.g. "Invalid Input.") must be displayed and the menu should be displayed again.

REQ3: Loading Game Boards and Data Structure Initialisation

The application comes with two different predefined boards. The data structure associated with each board is available in the start-up code. The user can load either of these boards as follows.

When user starts a game using board number 1:

```
load 1
|  |0|1|2|3|4|5|6|7|8|9|
|0|*|  |  |  |  |  |  |  |  |
|1|  |*|  |  |  |  |*|  |  |
|2|  |  |*|  |  |  |  |  |  |
|3|  |  |  |  |  |  |*|  |  |
|4|  |  |  |  |*|  |  |  |  |
|5|  |  |*|  |  |*|  |*|  |  |
|6|  |  |  |  |  |  |*|  |  |
|7|  |  |*|  |  |  |  |  |  |
|8|  |  |  |  |  |  |  |  |  |
|9|  |  |*|  |  |  |  |  |*|
```

This is another example where the user starts a game using board number 2:

```
load 2
|  |0|1|2|3|4|5|6|7|8|9|
|0|*|*|*|  |  |  |  |  |  |
|1|  |*|*|  |  |  |  |  |  |
|2|  |*|*|  |  |  |  |  |  |
|3|  |  |  |  |*|  |  |  |  |  |
|4|  |  |  |  |*|  |  |  |  |  |
|5|  |*|*|  |  |  |  |  |  |
|6|  |  |  |  |*|  |  |  |  |  |
|7|  |*|*|  |  |  |  |  |  |
|8|  |*|*|  |  |  |  |  |  |
|9|  |*|*|  |  |  |  |  |  |
```

At this stage you are required to initialise the board with the predefined data values which are available in the start-up code.

The cells in the board which contain a star sign (*) are blocked. The car cannot move to those cells. If the user attempts to move the car to one of those cells by the *forward* command, **an error will be shown** as discussed in section 5 below.

At this stage of the program, only three commands are acceptable:

- ***load <g>***
- ***init <x>,<y>,<direction>***
- ***quit***

If user enters any other command, or an incorrect command, an error message (e.g. `Invalid Input.`) must be displayed and the menu should be displayed again.

REQ4: Initialise Game

When the user loads a game board, then she should specify the initial position and direction of the car in the game board. This can be done through the *init <x>,<y>,<direction>* command. The meaning of the arguments of this command are as follows.

- <x>: an integer between 0-9 that specifies the horizontal location of the car
- <y>: an integer between 0-9 that specifies the vertical location of the car
- <direction>: any one of the values *north, east, south* or *west*. These specify the direction of the car's moving forward. For example, if *north* is specified and then *forward* command is issued, the car's current position will change to one cell towards the north (up).

Below is an example of initialising the game after loading game board 1:

```
init 5,3,north
| |0|1|2|3|4|5|6|7|8|9|
|0|*| | | | | | | | |
|1| |*| | | | | |*| | |
|2| | |*| | | | | | | |
|3| | | | | | |↑| |*| | |
|4| | | | |*| | | | | |
|5| | |*| | |*| |*| | |
|6| | | | | | |*| | | |
|7| | |*| | | | | | | |
|8| | | | | | | | | | |
|9| | |*| | | | | | |*|
```

You can see that the user specified arguments *5,3,north* and accordingly the car is positioned in the cell x=5 and y=3. The car should be shown in the grid with an arrow sign (↑, →, ↓, or ← according to the current direction of the car).

At this stage of the program, only four commands are acceptable:

- ***forward***
- ***turn_left (or l)***
- ***turn_right (or r)***
- ***quit***

If user enters any other command, or an incorrect command, an error message (e.g. "Invalid Input.") must be displayed and the menu should be displayed again.

REQ5: PLAY GAME

When the game is initialised, as explained in section 4 above, the user can move the car in the grid by using the *forward* command. This command will move the car one cell towards up, right, down or left. **The direction of the move is decided according to the current direction of the car.**

For example, if the current direction of the car is *north*, moving forward will increase the vertical location of the car by one cell. If the current direction is *left*, moving forward will decrease the current vertical position of the car by one cell.

Below is an example of how *forward* command would work on the board shown in section 4. Remember that in section 4 the board was initialised with x=5, y=3 and direction=north. Here a *forward* command is taking the car one cell towards the north (up). The new position of the car is x=5 and y=2, and the direction is not affected (remains *north*).

```
forward
| |0|1|2|3|4|5|6|7|8|9|
|0|*| | | | | | | | |
|1| |*| | | | | |*| | |
|2| | |*| | |↑| | | | |
|3| | | | | | | |*| | |
|4| | | | |*| | | | | |
|5| | |*| | |*| |*| | |
|6| | | | | | |*| | | |
|7| | |*| | | | | | | |
|8| | | | | | | | | | |
|9| | |*| | | | | | |*|
```

The *turn_right* command will change the direction of the car in the following sequence.

- North → East
- East → South
- South → West
- West → North

The *turn_left (or l)* command will affect the direction of the car in the opposite sequence:

- North → West
- West → South
- South → East
- East → North

Assuming that the car is currently located in [x=4, y=5] and pointing towards *north* (↑), below is an example of how it will be affected by a *turn_right (or r)* command followed by a *forward* command.

```
turn_right
forward
Error: cannot move forward because the road is blocked.

| |0|1|2|3|4|5|6|7|8|9|
|0|*| | | | | | | | |
|1| |*| | | | |*| | |
|2| | |*| | | | | | |
|3| | | | | | | |*| | |
|4| | | | |*| | | | | |
|5| | |*| |→|*| |*| | |
|6| | | | | | |*| | | |
|7| | |*| | | | | | | |
|8| | | | | | | | | | |
|9| | |*| | | | | | |*|

The car is at the edge of the board and cannot move further in that
direction
```

Note that in the example above the user first issued a *turn_right (or r)* command, which changed the direction of the car from *north* (↑) to *east* (→).

Given the new direction, when the user used the *forward* command, the program attempted to move the car towards the east. However, there is a roadblock on the right side of the current position (* on x=5, y=5) and the car cannot be moved forward. The error message indicates this problem. **This is the way you are expected to handle hitting the roadblocks.**

A further example of moving the car is shown below.

```
turn_right
forward
forward
| |0|1|2|3|4|5|6|7|8|9|
|0|*| | | | | | | | |
|1| |*| | | | | |*| | |
|2| | |*| | | | | | | |
|3| | | | | | | |*| | |
|4| | | | |*| | | | | |
|5| | |*| | |*| |*| | |
|6| | | | | | |*| | | |
|7| | |*| |↓| | | | | |
|8| | | | | | | | | | |
|9| | |*| | | | | | |*|
```

The car shown in the previous example has turned right again, which changed its direction from *east* (→) to *south* (↓). It then moved forward twice.

NOTE: The command *turn_left* should be interchangeable with the command *l*, and the command *turn_right* should be interchangeable with the command *r*. That is, the command *l* should behave exactly similar to *turn_left*, and the command *r* should behave exactly similar to *turn_right*.

REQ6: Stopping at the Edges of the Board and Before the Road Blocks

The car must not move beyond the edges of the board. For example, a car in a position with x=9 should not be able to move towards the *east* any more. Similarly, a car in a position with y=0 should not be able to move towards the *north* any more.

If the user attempts to move the car beyond the edges of the board, you should display the error message `The car is at the edge of the board and cannot move further in that direction`.

REQ7: Quit the Main Menu

This option should terminate your program without any crash for any reasons (i.e. exit from function *main()* of your C code).

REQ8: Return to the Main Menu

When the *quit* command is entered in the middle of the game, the game should terminate and the control should be returned to the main menu.

Additionally, the total number of successful *forward* moves in the game should be displayed before exiting the game. For example, if the player attempts to move the car forward four times and only three of these attempts were successful (e.g. one move hit a road block), the following message should be displayed when exiting the current game: `Total player moves: 3`

REQ9: Demonstration in Lab

A demonstration is required for this assignment in order to show your progress. This will occur in your scheduled lab classes. Demonstrations will be very brief, and you must be ready to demonstrate your work in a two-minute period when it is your turn.

As part of the assignment demo, the tutor may ask you random questions about your source code. You may be asked to open your source files and explain the operation of a randomly picked section. In the event that you will not be able to answer the questions, we will have to make sure that you are the genuine author of the program.

Buffer handling and input validation will not be assessed during demonstrations, so you are advised to incorporate these only after you have implemented the demonstration requirements. Coding conventions and practices too will not be scrutinised, but it is recommended that you adhere to such requirements at all times.

During the demonstration you will be marked for the following:

- Ability to compile/execute your program from scratch.
- Requirements 1, 2, 3 and 4 should be completed and functional. Your tutor will mark your work accordingly.

GENERAL REQUIREMENTS

GR1: BUFFER HANDLING

This requirement relates to how well your programs handle "extra input".

To aid you with this requirement, we want you to use the *readRestOfLine()* function provided in the start-up code. Use of this function with *fgets()* is demonstrated in sample source codes and tutorial exercises.

Marks will be deducted for the following buffer-related issues:

- Prompts being printed multiple times or your program "skipping" over prompts because left over input was accepted automatically. These problems are a symptom of not using buffer overflow code often enough.
- Program halting at unexpected times and waiting for the user to hit enter a second time. Calling your buffer clearing code after every input routine without first checking if buffer overflow has occurred causes this.
- Using *gets()* or *scanf()* for scanning string input. These functions are not safe because they do not check for the amount of input being accepted.
- Using long character arrays as a sole method of handling buffer overflow. We want you to use the *readRestOfLine()* function.
- Other buffer related problems.

For example, what happens in your program when you try to enter a string of 40 characters when the limit is 20 characters?

GR2: INPUT VALIDATION

For functionality that we ask you to implement that relies upon the user entering input, you will need to check the length, range and type of all inputs where applicable.

For example, you will be expected to check the length of strings (e.g. acceptable number of characters?), the ranges of numeric inputs (e.g. acceptable value in an integer?) and the type of data (e.g. is this input numeric?).

For any detected invalid inputs, you are asked to re-prompt for this input. You should not truncate extra data or abort the function.

GR3: CODING CONVENTIONS AND PRACTICES

Marks are awarded for good coding conventions/practices such as:

- Completing the header comment sections in each source file included in the submission.
- Avoiding global variables. If you still do not know what global variables are, do not assume. Ask the teaching team about it.
- Avoiding *goto* statements.
- Consistent use of spaces or tabs for indentation. Either consistently use tabs or three spaces for indentation. Be careful not to mix tabs and spaces. Each "block" of code should be indented one level.
- Keeping line lengths of code to a reasonable maximum such that they fit into 80 columns.
- Appropriate commenting to explain non-trivial sections of the code.
- Writing header descriptions for all functions.
- Appropriate identifier names.
- Avoiding magic numbers (remember, it is not only about numbers).
- Avoiding platform specific code with *system()*.
- General code readability.

GR4: FUNCTIONAL ABSTRACTION

We encourage the use of functional abstraction throughout your code. This is considered to be a good practice when developing software with multiple benefits. Abstracting code into separate functions reduces the possibility of bugs in your project, simplifies programming logic and make the debugging less complicated. We will look for some evidence of functional abstraction throughout your code.

As a rule, if you notice that you have the same or similar block of code in two different locations of your source, you can abstract this into a separate function. Another easy rule is that you should not have functions with more than 50 lines of code. If you have noticeably longer functions, you are expected to break that function into multiple logical parts and create new functions accordingly.

START-UP CODE

We provide you with a start-up code that you must use in implementing your program. The start-up code includes a number of files which you must carefully read, understand and complete. Refer to the following sections for further details.

STRUCTURE AND FILES

The start-up code includes the following 10 files:

- carboard.h and carboard.c: This is the main header and source files. The main method should be implemented in .c file.

- board.h and board.c: These files include the declaration and body of the functions related to the board.
- game.h and game.c: These files contain the body of the logic of the game. In addition to the provided function in the header file, you will need to use functional abstraction to break down the logic of the game into smaller functions.
- player.h and player.c: These files contain functions related to specific actions of the player.
- helpers.h and helpers.c: These files are used to declare the variables and functions which are used throughout the system and are shared among multiple other modules. A very good example is the *readRestOfLine()* function which must be used for reading input from the player.

DESCRIPTION ABOUT THE TYPEDEFS

The following enumeration specifies the various states of each cell on the board.

```
typedef enum cell
{
      EMPTY,
      BLOCKED,
      PLAYER
} Cell;
```

The *playerMove* enumeration specifies the possible outcome from the next move of the user.

```
typedef enum playerMove
{
    PLAYER_MOVED,
    CELL_BLOCKED,
    OUTSIDE_BOUNDS
} PlayerMove;
```

The following enumeration is the definition of boolean data type, which is not available in standard C.

```
typedef enum boolean
{
    FALSE = 0,
    TRUE
} Boolean;
```

The *direction* enumeration specifies the possible directions that the car can take.

```
typedef enum direction
{
    NORTH,
    EAST,
    SOUTH,
    WEST
} Direction;
```

The *turnDirection* specifies the possible turns that a car can make.

```
typedef enum turnDirection
{
    TURN_LEFT,
    TURN_RIGHT
} TurnDirection;
```

The *position* struct represents the position of the car on the board.

```
typedef struct position
{
      int x;
      int y;
} Position;
```

The *player* struct represents the data that should be stored for a particular player's car: the position of the car, the direction of the car, and the number of valid moves of the car.

```
typedef struct player
{
      Position position;
      Direction direction;
      unsigned moves;
} Player;
```

MARKING GUIDE

The following table contains the detailed breakdown of marks allocated for each requirement.

Requirement	**Mark**
REQ1: Main Menu	10
REQ2: Displaying the Board	10
REQ3: Loading Game Boards and Data Structure Initialisation	5
REQ4: Initialise Game	5
REQ5: Play Game	25
REQ6: Stopping at the Edges of the Board and before the Road Blocks	5
REQ7: Quit the Main Menu	5
REQ8: Return to the Main Menu	5
REQ9: Demonstration in Lab	10
GR1: Buffer Handling	5
GR2: Input Validation	5
GR3: Coding Conventions and Practices	5
GR4: Functional Abstraction	5
Total	**100**

PENALTIES

Marks will be deducted for the following:

- Compile errors and warnings.
- Fatal run-time errors such as segmentation faults, bus errors, infinite loops, etc.
- Missing files (affected components get zero marks).
- Files incorrectly named, or whole directories submitted.

- Not using start-up code or not filling-in the readme file.

Programs with compile errors that cannot be easily corrected by the marker will result in a maximum possible score of 40% of the total available marks for the assignment.

Any sections of the assignment that cause the program to terminate unexpectedly (i.e., segmentation fault, bus error, infinite loop, etc.) will result in a maximum possible score of 40% of the total available marks for those sections. Any sections that cannot be tested as a result of problems in other sections will also receive a maximum possible score of 40%.

It is not possible to resubmit your assignment after the deadline due to mistakes.

SUBMISSION INFORMATION

Create a zip archive using the following command on *saturn*, *jupiter* or *titan*:

```
zip ${USER}-a1 \
board.c board.h \
carboard.c carboard.h \
game.c game.h \
helpers.c helpers.h \
player.c player.h
```

This will create a .zip file with your username on the server. For example, if your student number is 1234567 then the file will be named **s1234567-a1.zip**. If you are using your staff account, you must rename the file to your student number before the submission.

You may use other methods to create a zip file for your submission. Note that **you must test your archive file before submission** to make sure that it is a valid archive, contains the necessary files, and can be extracted without any errors.

Submit the zip file before the submission deadline.

LATE SUBMISSION PENALTY

Late submissions attract a marking deduction of 10% of the maximum mark attainable per day for the first 5 days, including weekdays and weekends. After this time, a 100% deduction is applied.

We recommend that you avoid submitting late where possible because it is difficult to make up the marks lost due to late submissions.