
Assignment 1 Car Board

Assessment Type: Individual Assessment.

Weight: 35% of the final course mark

Due Date: 23:59, Monday 10 April 2023 (Week 7)

Silence Policy: From 5:00pm, Friday 7 April 2023 (Week 6)

Submission: Online via Canvas. Submission instructions are provided on Canvas.

1 Overview

In this assignment you will 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 this assignment you will:

- Practice the programming skills such as:
 - Pointers
 - Dynamic Memory Management
 - Arrays/Vectors
- Implement a medium size C++ program using predefined classes
- Use a prescribed set of C++11/14 language features

This assignment is on 4 milestones and marked according to three criteria, as given on the Marking Rubric on Canvas:

- Milestone 1: Demonstration in lab.
- Milestone 2-4: Implementation of the Car Board.
- Style & Code Description: Producing well formatted, and well documented code.

1.1 Relevant Workshop/Lab Material

To complete this assignment, you will require skills and knowledge from workshop and lab material for Weeks 2 to 6. 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. You will need to learn how to use C++ Standard Library class template `vector` much sooner so that you can build your game board. You are encouraged to learn new C++ features through C++ reference documentation such as <https://cplusplus.com/>.

1.2 Learning Outcomes

This assessment relates to all of the learning outcomes of the course which are:

- Analyse and Solve computing problems; Design and Develop suitable algorithmic solutions using software concepts and skills both (a) introduced in this course, and (b) taught in pre-requisite courses; Implement and Code the algorithmic solutions in the C++ programming language.
- Discuss and Analyse software design and development strategies; Make and Justify choices in software design and development; Explore underpinning concepts as related to both theoretical and practical applications of software design and development using advanced programming techniques.
- Discuss, Analyse, and Use appropriate strategies to develop error-free software including static code analysis, modern debugging skills and practices, and C++ debugging tools.
- Implement small to medium software programs of varying complexity; Demonstrate and Adhere to good programming style, and modern standards and practices; Appropriately Use typical features of the C++ language include basic language constructs, abstract data types, encapsulation and polymorphism, dynamic memory management, dynamic data structures, file management, and managing large projects containing multiple source files; Adhere to the C++14 ISO language features.
- 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.

2 Description of the Program

The development of *Car Board* program can be broken down into following smaller requirements (REQs).

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 also specified in REQ7 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 REQ3 below.

The *init* command is responsible to set the initialised values of the car's position and movement. More detail about this command is available in REQ4 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 REQ5 below.

The *quit* command will exit the current game and return to the *main menu*. This requirement is defined in REQ8 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 REQ2 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 REQ5 below.

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

- *load* $\langle g \rangle$
- *init* $\langle x \rangle, \langle y \rangle, \langle direction \rangle$
- *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 he/she should specify the initial position and direction of the car in the game board. This can be done through the *init* $\langle x \rangle, \langle y \rangle, \langle direction \rangle$ command. The meaning of the arguments of this command are as follows.

- $\langle x \rangle$: an integer between 0-9 that specifies the horizontal location of the car
- $\langle y \rangle$: an integer between 0-9 that specifies the vertical location of the car
- $\langle direction \rangle$: 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 ($\uparrow, \rightarrow, \downarrow$, or \leftarrow 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 the previous section on REQ4, 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 REQ4. Remember that in REQ4 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 (\rightarrow) to south (\downarrow). 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 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 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: 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.

3 Start-up Code

To help you understand and practice object-oriented programming principles, we are providing you with start-up code that you must use for the assignment. The start-up code consists of nine files that break down the game logic into smaller, more manageable pieces.

The `main()` function of your program is in the `carboard.cpp` file, and the remaining files are pairs of header and source files, each containing the declaration and implementation for a specific class. Here's a brief overview of each class:

- Game class (in `game.h`) - contains the game's logic broken down into smaller pieces. The `start()` method should be invoked by the `main()` method in `carboard.cpp` to start the game. This method should call other methods in the Game class to load the game board, initialize the player on the board, and start the game to move the player around, respectively. You'll need to implement all these methods, as well as those in the other classes when implementing the logic of the game.
- Player class (in `player.h`) - contains attributes of the player, such as its current position and direction on the board, as well as methods related to the player, such as `initialisePlayer()`, `turnDirection()`, `getNextForwardPosition()`, `updatePosition()`, and `displayDirection()`.
- Board class (in `board.h`) - contains a two-dimensional vector holding the configuration of the game board, as well as relevant methods such as `load()` to load a board, `placePlayer()` to place a player on the board, `movePlayerForward()` to move the player forward by one cell, and `display()` to display the player as an arrow on the board. Two pre-configured boards are provided in `board.cpp` for your assignment.
- Helper class (in `helper.h`) - provides several utility functions that are commonly used for input validation. The implementations of these functions are provided in the `helper.cpp` file. You should think about how to use them for your own implementation.

Additionally, the provided files contain predefined data types and strings that should be used to avoid hard-coding numbers or strings, thereby enhancing the program's readability. You should not modify these existing data types or method prototypes. However, you are allowed to add your own new member variables and methods to support your implementation. Note that we have provided some explanation on what each method is expected to do in the header files (in the comments before each function prototype). We suggest you to read these descriptions first to get a better understanding of how to use these methods to accomplish a desired functionality.

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

```
g++ -Wall -Werror -std=c++14 -O -o assign1 board.cpp carboard.cpp game.cpp helper.cpp player.cpp
```

4 Milestone Implementations

You may assume that the environment is a fixed size of 10x10, except for Milestone 4. This assignment has four Milestones. To receive a PA/CR 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 should be completed in sequence. For example, if you attempt Milestone 3, but your Milestone 2 is buggy, you won't get any marks for Milestone 3.

Milestone 1: Demonstration in Lab

A demonstration is required for this assignment in order to show your progress. **This will occur in your scheduled lab classes in Week 4.** Demonstrations will be very brief, and you must be ready to demonstrate your work in a 3-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.

Input validation will be also assessed during demonstrations. An executable of the program is provided so that

you can see how the program is supposed to behave, hence providing you some ideas on how you could test your own program with respect to input validation (following the principle of *defensive programming*). You can access and run the executable file directly on the school teaching server (e.g., via PuTTY on Windows) at the following location: `~/KDrive/SEH/SCSIT/Students/Courses/COSC1076/apt`. Once logged in with your RMIT credential, you can just `cd` into this folder, and find the executable file `car`. Just run it at the commandline. If you are new to Unix and not sure how to do this, please refer to the [Unix Survival Guide](#) on Canvas.

Coding conventions and practices will not be scrutinised at this milestone, however 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 (ie., REQs) 1, 2, 3, 4 and 9 should be completed and functional. Your tutor will mark your work accordingly.

Milestone 2: Implementation of Play Game (REQ5)

A good piece of software requires careful design and uses suitable data structures and classes. In Assignment 1, you will implement our design, that is, our provided start-up code including newly defined data structures and classes. These classes and the prototypes of their associated methods are provided. You will need to use the start-up code as the starting point to implement the methods of these classes. Read Section 3 on **Start-up Code** for details about these classes, which should provide you some clues on how to start on this milestone. You are encouraged to develop some sort of pseudo-code to capture the game logic before starting to implement your code. It is also important to follow a *structured programming* view (i.e., “single point of entry and single point of exit”) for developing your program, as it encourages better modularity of your code.

Milestone 3: Stopping at the Edges of the Board (REQ6), REQ7 and REQ8

Subsequent to Milestone 2, you will need to handle scenarios when the car hit the edge of the game board. The car is meant to stay within the board at all time, hence appropriate methods need to be called to do this checking, and an error message should be also displayed to prompt the user what has happened. Please refer to REQ6. In addition, if the user wishes to quit the game, then the program should be returned to the main menu (REQ8). Furthermore, the user can also exit the program at the main menu (REQ7). You need to continue to ensure input validation is appropriately carried out in this milestone.

Milestone 4: Further Extension to the Program

Attempt this milestone only if you have successfully completed the Milestones 1, 2 and 3, since this milestone is considered more challenging. In Milestones 1 - 3, we assume that the game board is always of a fixed size (10x10). This means the board size is fixed and cannot be changed during a program run.

For Milestone 4, you are required to modify your implementation to accommodate two important changes:

- Dynamically resize the board size according to an user input, e.g., any size ranging from 10 up to 20, rather than using a fixed size at compile time. Please note you must do this with a re-sizable array. It is not enough to just use a larger fixed size array to achieve this functionality.
- Randomly allocate road blocks on the board, according to a percentage value, e.g., if a user enters 0.1, then it means 10% of the cells on the board will be randomly allocated with road blocks, while the remaining cells are empty.

Below are expected behaviours of Milestone 4 program:

After main menu, if the user chooses 1 (i.e., Play Game), the following list of commands should be displayed.

```

Welcome to Car Board
-----
1. Play Game
2. Show student information
3. Quit

Please enter your choice: 1

You can use the following commands to play the game:

generate <d>,<p>
    d: the dimension of the game board to be generated
    p: the probability of the blocks on board to be generated randomly
init <x>,<y>,<direction>
    x: horizontal position of the car on the board
    y: vertical position of the car on the board
    direction: direction of the car's movement (north, east, south, west)
forward (or f)
turn_left (or l)
turn_right (or r)
quit

```

You would notice that now you have an extra command:

- *generate* <d>,<p>
- *d*: the dimension of the game board to be generated
- *p*: the probability of the blocks on board to be generated randomly

This command allows the user to specify the size (or dimension) of the game board, and the percentage of the cells to be allocated randomly with road blocks. If the user enters *generate 15,0.3*, then following is what the display would look like, i.e., a 15 x 15 game board with 30% of its cells randomly allocated with blocks.

```

generate 15,0.3

```

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

The rest of the M4 program should behave the same way as your M1-M3 implementation, including placing the car, and moving the car in different directions, etc.

To carry out your implementation on Milestone 4, you will need to extend appropriate classes to accommodate new methods for implementing the above two functionalities. We suggest that you have two separate working folders:

- M123 for your codes developed for Milestones 1, 2 and 3.
- M4 for Milestone 4.

It is important to make sure that your M4 code can be compiled and run independently in this folder, completely separate from the M123 folder.

5 Documentation, Style and Code Description

Making sure your code is 100% correct is very important. Making sure your code is understandable is equally important. Your code should follow the Course Style Guide, as given on Canvas (including not using any banned elements), and should be well documented with clear comments.

Finally, you need to provide a short 1-paragraph description (at the top of your main file) to:

- Describe (briefly) the approach you have taken in your implementation.
- Describe (briefly) any issues you encountered.
- Describe (briefly) potential improvements of the design of the given start-up code.

6 Submission

Follow the detailed instructions on Canvas to complete your submission for Assignment 1.

Assessment declaration: When you submit work electronically, you agree to the *assessment declaration* on the following page: <https://www.rmit.edu.au/students/my-course/assessment-results/assessment>.

It is recommended you test your program prior to submission on the RMIT servers using the program “valgrind”. There will be a demonstration video on how to do this closer to the due date.

6.1 Silence Period

A silence policy will take effect from 5.00pm, Friday 7 April 2023. 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.

6.2 Late Submissions & Extensions

A penalty of 10% per day is applied to late submissions up to 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.

7 Marking Guidelines

The marks are divided into three categories:

- Lab Demo: 5/35 (about 14%)
- Software Implementation: 20/35 (about 57%)
- Code Style, Documentation & Code Description: 10/35 (about 28%)

The detailed breakdown of this marking guidelines is provided on the rubric linked on Canvas. Please take note that the rubric is structured with three “brackets”:

- 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 your mark will be a DI, getting a DI in all rubric categories
- If you 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.

8 Academic integrity and plagiarism (standard warning)

! *Plagiarism is a very serious offence.*

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 following: <https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity>.