ASSIGNMENT ONE: HUNT THE WUMPUS

ADVANCED PROGRAMMING TECHNIQUES – SEMESTER 2, 2018

SUMMARY

In this assignment, you will use your C programming skills to build a simple game called **Hunt the Wumpus**. This is a simplified variant of a game by the same name created in the early 1970's by Gregory Yob. The rules for the game are simple: a player can move around inside a board where the cells represent adjoining caverns; the caverns may contain bottomless pits, bats that can carry the player away to another empty cavern and a deadly creature called the 'wumpus'. The player can use warning clues indicating if bats, pits or the wumpus are nearby. The player can also fire arrows, which can potentially kill the wumpus.

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. Just producing code to implement the requirements is not enough. *You **must** use the start-up code in your assignment, and you are **not** permitted to change the given function prototypes.* The prototypes have been written in such a way to make you demonstrate your ability to use certain concepts taught in this course. You are required to submit the exact files provided in the start-up code after completing the specified tasks.

SUBMISSION

Total mark: 20% of the final mark

Assignment demo: During tutorial / lab sessions of week 4 and 5.

Final submission: End of week 6 – Friday, 24 August 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: https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity.

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 wumpus board.c huntWump.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 <u>not</u> 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. `wumpus`)*.

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

DESCRIPTION OF THE PROGRAM

CORE REQUIREMENTS

REQ1: MAIN MENU

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

```
./wumpus
Welcome to Hunt the Wumpus
--------------------------
1. Play game
2. Show student 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 print the error message `Invalid input.` and show the menu 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 Hunt the Wumpus
--------------------------
1. Play game
2. Show student's information
3. Quit

Please enter your choice: 2

---------------------------------
Name: <your full name>
Student ID: <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 8 below.

If user selects menu number 1, the following list of commands will be displayed, down to `Press return to continue...` . After the user press's the enter key, the remainder of the list is presented.

```
Welcome to Hunt the Wumpus
--------------------------
1. Play game
2. Show student 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 (either 1 or 2)
init <x>,<y>
 x: horizontal position of the player on the board (between 0 & 4)
 y: vertical position of the player on the board (between 0 & 4)
north (or n)
south (or s)
east (or e)
west (or w)
quit

Press enter to continue...
At this stage of the program only two commands are acceptable:
load <g>
quit
```

The *load* command is responsible to initialise the game board. More details about this command is available in section 2 below.

The *init* command is responsible to set the initialise values of the player's position. More detail about this command is available in section 3 below.

The *north/south/east/west* commands are responsible to move the player one position in the relevant direction from the player's current position. 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 9 below.

If invalid input is entered, the program should print the error message `Invalid input.` and prompt for commands *load* or *quit* until valid input is entered.

REQ2: Loading the Game Board

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 by specifying the command *load* followed by either *1* or *2*.

An example where the user uses board number 1:

```
load 1

Board successfully loaded
```

At this stage you are required to set the the board to the predefined data values which are available in the start-up code and the message `Board successfully loaded` printed once loading has been performed.

REQ3: Initialise Player Position

After the user loads a game board, they can specify the initial position of the player in the game board. This can be done through the *init <x>,<y>* command. The meaning of the arguments of this command are as follows.

- <x>: an integer between 0-4 that specifies the horizontal location of the player
- <y>: an integer between 0-4 that specifies the vertical location of the player

The board (containing the cavern cells, the position and any previous positions of the player) and any warnings must be displayed on the screen after it is *initialised*. See the requirement for printing the board in section 4 below.

Below is an example of initialising the game after loading game board 1 using *init 0,1* which places the player at coordinates 0,1, indicated by ##

```
init 0,1

   0  1  2  3  4
  ----------------
0 |  |  |  |  |  |
  ----------------
1 |##|  |  |  |  |
  ----------------
2 |  |  |  |  |  |
  ----------------
3 |  |  |  |  |  |
  ----------------
4 |  |  |  |  |  |
  ----------------
You hear flapping!
```

After initialisation of the player has been performed, the following options are presented:

```
At this stage of the program, only three commands are acceptable:
 <direction>
 shoot <direction>
 quit
Where <direction> is one of: north,n,south,s,east,e,west,w

Please enter your choice:
```

If invalid input is entered, the program should print the error message `Invalid input.` and prompt for the commands *<direction>, shoot <direction>* or *quit* until valid input is entered.

REQ4: DISPLAYING THE BOARD AND WARNINGS

The game has a 5x5 board. The board (containing the cavern cells, the position and any previous positions of the player), together with any warnings must be displayed on the screen after it is *initialised*, after the player attempts to move, and after the player attempts to fire an arrow.

The position of the player is indicated by ##, and any positions previously occupied by the player are indicated by **. For example assume the initial position of the player was 0,0 and then the player moved south to position 0,1 on board 1. The displayed board would be:

```
   0  1  2  3  4
  ----------------
0 |**|  |  |  |  |
  ----------------
1 |##|  |  |  |  |
  ----------------
2 |  |  |  |  |  |
  ----------------
3 |  |  |  |  |  |
  ----------------
4 |  |  |  |  |  |
  ----------------
```

Associated with each displaying of the board is the displaying of any warnings. In the above example, the warnings would be `You hear flapping!`, which would be printed under the board.

To determine warnings, board positions immediately adjacent to the player are checked for hazards. For example, if the player was at position (2,3), then board positions (1,2), (2,2), (3,2), (1,3), (3,3), (1,4), (2,4), (3,4) would be checked for hazards.

Hazards are reported as follows:

- If any number of positions containing a pit are found, a single warning `You feel a breeze!` is displayed.
- If any number of positions containing bats are found, a single warning `You hear flapping!` is displayed.
- If a wumpus is found, the warning `You smell a wumpus!` is displayed.

REQ5: MOVING THE PLAYER

When the game is initialised, as explained in section 3 above, the user can move the player in the board by specifying a direction. This will move the player one cell towards up, right, down or left from the current position of the player.

For example, if the current position of the player is coordinates 0,1 then moving *north* will move the player up to coordinates 0,0. The player must not move beyond the edges of the board. For example, a player in a position with x=0 should not be able to move towards the *west* any more. Similarly, a player in a position with y=0 should not be able to move towards the *north* any more.

If the user attempts to move the player beyond the edges of the board, the program should display the error message `Unable to move - outside bounds`, and the move/shoot/quit options should be presented again.

REQ6: HANDLING PLAYER HAZARDS – PITS, BATS, AND WUMPUS

If a player moves into a board cell containing a pit or the wumpus, the player is killed and the message `Player killed!` Printed. The user should then be returned to the main menu.

If a player moves into a board cell containing bats, the player is moved to another board location as follows: The function *rand* from *stdlib.h* should be used to generate new 'pseudo random' coordinates for the player until a new board location is found which does not contain a bat, pit

or wumpus. The random number generator should have been initialised at the start of the game by calling *srand(0)*.

REQ7: SHOOTING AN ARROW

The player initially has 5 arrows. The user can use the *shoot* command, specifying a direction, to shoot an arrow one cell into an adjacent position, resulting in one of the following actions:

- If the number of remaining arrows is 0, the message `You don't have any arrows to fire.` is displayed.
- If the player has arrows but the specified direction would take the arrow out of the board, the message `Unable to fire arrow in that direction.` is displayed and the number of arrows is unchanged.
- If the player had arrows but the specified direction of shooting did not target the position of the wumpus, the message `Missed. You now have n arrows` is displayed, where *n* is the number of remaining arrows. The number of remaining arrows is reduced by 1.
- If the player had arrows and the specified direction of shooting did target the position of the wumpus, the message `You killed the Wumpus!` is displayed and the main menu is presented.

REQ8: 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).

REQ9: 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.

REQ10: 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, 4 and 8 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 very 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:

- huntWump.h and huntWump.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
{
      board_EMPTY,
      board_TRAVERSED,
      board_BATS,
      board_PIT,
      board_WUMPUS,
      board_PLAYER
} Cell;
```

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

```
typedef enum playerMove
{
    board_PLAYER_MOVED,
    board_PLAYER_KILLED,
    board_BAT_CELL,
    baord_OUTSIDE_BOUNDS
} PlayerMove;
```

The *arrowHit* enumeration specifies the possible outcome from firing an arrow.

```
typedef enum arrowHit
{
    board_ARROW_MISSED,
    board_WUMPUS_KILLED,
    baord_ARROW_OUTSIDE_BOUNDS
} ArrowHit;
```

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 player can take.

```
typedef enum direction
{
    player_NORTH,
    player_EAST,
    player_SOUTH,
    player_WEST
} Direction;
```

The *position* struct represents the position of the player 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 : the player's position of the car, the direction of the car, and the number of valid moves of the car.

```
typedef struct player
{
      Position position;
      unsigned numArrows;
} Player;
```

MARKING GUIDE

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

Requirement	**Mark**
REQ1: Main Menu	10
REQ2: Loading the game board	5
REQ3: Initialise player position	5
REQ4: Displaying the board and warnings	10
REQ5: Moving the player	10
REQ6: Handling player hazards – Pits, bats and wumpus	10
REQ7: Shooting an arrow	10
REQ8: Quit the main menu	5
REQ9: Return to the main menu	5
REQ10: 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 \
huntWump.c huntWump.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.