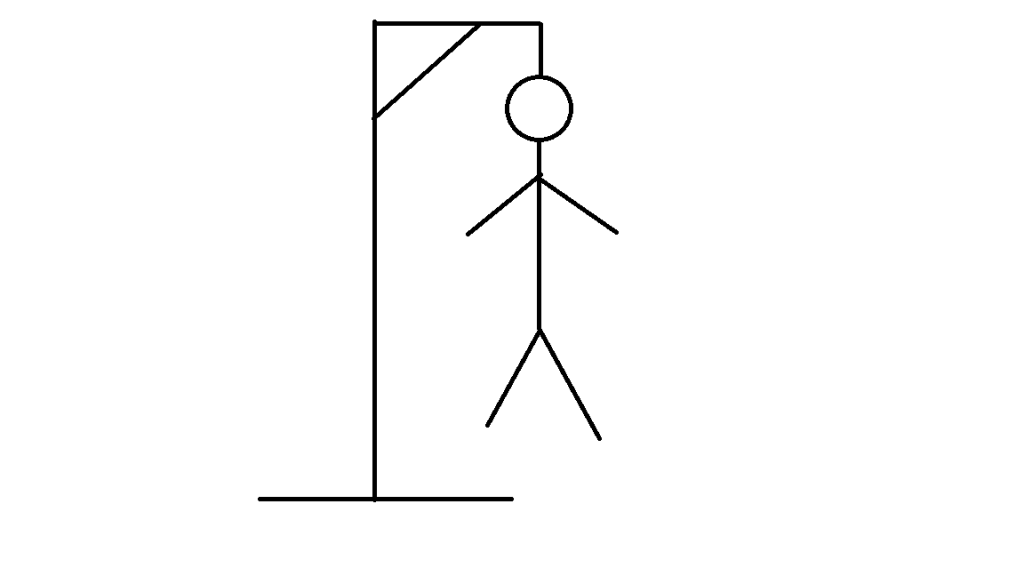


## Problem 1

The hangman game is a guessing game where the user guesses letters until they figure out what the secret word is. In most versions of the game, the user has 6 chances to guess the correct word. For each incorrect guess, a part of the hangman is drawn. If the user fails 6 times, the hangman is complete, and the user loses. If the user knows the secret word, they keep guessing the correct letters until the word is complete, which results in a win.

The goal of this problem is to simulate the hangman game. Because we don’t want to draw the hangman, we will just use textual messages to communicate the user of their status in the game. Follow the items below to complete your game step by step.



In your hw3\_p1.c file, you will create the following:

1. (2 points) A global constant array named FRUITS that contains the following elements (single-word fruits from [Wikipedia](https://simple.wikipedia.org/wiki/List_of_fruits)):

Abiu, Acerola, Akebi, Ackee, Apple, Apricot, Araza, Avocado, Banana, Bilberry, Blackberry, Blackcurrant, Blueberry, Boysenberry, Breadfruit, Canistel, Cashew, Cempedak, Cherimoya, Cherry, Cloudberry, Coconut, Cranberry, Currant, Damson, Date, Dragonfruit, Pitaya, Durian, Elderberry, Feijoa, Fig, Gooseberry, Grape, Raisin, Grapefruit, Guava, Honeyberry, Huckleberry, Jabuticaba, Plinia, Jackfruit, Jambul, Jostaberry, Jujube, Kiwano, Kiwifruit, Kumquat, Lemon, Lime, Loganberry, Longan, Loquat, Lulo, Lychee, Mango, Mangosteen, Marionberry, Melon, Cantaloupe, Galia, Melon, Honeydew, Watermelon, Mulberry, Nance, Nectarine, Orange, Clementine, Mandarine, Tangerine, Papaya, Passionfruit, Pawpaw, Peach, Pear, Persimmon, Plantain, Plum, Prune, Pineapple, Pineberry, Plumcot, Pluot, Pomegranate, Pomelo, Quince, Raspberry, Salmonberry, Rambutan, Redcurrant, Salak, Sapodilla, Sapote, Satsuma, Soursop, Strawberry, Tamarillo, Tamarind, Tangelo, Tayberry, Ximenia, and Yuzu.

1. (5 points) Function getSecretWord()

/\*

Function: getSecretWord()

Process: randomly generates an index between zero and the

length of the array of FRUITS. Then, selects the

appropriate secret word (based on the index) and

attribute the fruit name to the output parameter

(secret\_word)

Input data (parameter): none

Output data (return): size – indicates the length of the

selected secret word. For example, if the

selected word is yuzu, the function returns 4; if the selected word is grapefruit, the function returns 10.

Output data (parameter): secret\_word

Dependencies: srand, rand, strlen

\*/

1. (5 points) Function buildOutputArray()

/\*

Function: buildOutputArray()

Process: creates an output array with blank spaces that are

shown to the user as the first tip. The output array

must be the same size as the secret word and contain

one underscore character for each position. For

example, if the secret word is “yuzu”, the output

array will be = {‘\_’, ‘\_’, ‘\_’, ‘\_’}.

Input data (parameter): size – the length of the secret\_word

Output data (return): void

Output data (parameter): output\_array – contains the array

filled with the correct number of underscores.

Dependencies: none

\*/

1. (5 points) Function buildAttemptsArray()

/\*

Function: buildAttemptArray()

Process: creates an array with 6 positions to store the

user’s incorrect guesses.

Input data (parameter): none

Output data (return): void

Output data (parameter): attempts – array of characters of

length 6, filled with a star (‘\*’)character in each

position. The star represents that the no guesses

were made yet.

Dependencies: none

\*/

1. (5 points) Function checkGuess()

/\*

Function: checkGuess()

Process: given a letter and a secret word, returns TRUE (1) if the letter is in the secret word and FALSE (0) otherwise.

Input data (parameter): guess – a character guessed by the

user

secret\_word – array of characters that represent the secret word

Output data (return): TRUE/FALSE depending on whether the

letter is in the secret word

Output data (parameter): none

Dependencies: strlen

\*/

1. (5 points) Function fillOutputArray()

/\*

Function: fillOutputArray()

Process: for each letter in the secret word, verifies whether

the letter is the user guess. If it is, replace the

underscore `\_` in the output array with the letter.

For example, if the user guesses ‘u’ and the secret

word is `yuzu`, then the output array should become

{‘\_’,’u’,’\_’,’u’}.

Input data (parameter): letter – the user’s correct guess

secret\_word – array of characters that represent

the secret word

output\_array – array with the underscores.

Output data (return): TRUE (1) if any underscore is replaced

with the guessed letter. FALSE (0) if the guessed

letter was already filled (repeated guess).

Output data (parameter): output\_array – array with underscores replaced with the guessed letter in the correct indexes.

Dependencies: strlen

\*/

1. (5 points) Function fillAttemptsArray()

/\*

Function: fillAttemptsArray()

Process: replaces the next star (\*) in the attempt array with

a letter that represents a wrong guess from the

user. For example, if the secret word is “yuzu” and

the first user guess ia ‘a’, the attempts array

should become {‘a’, ‘\*’, ‘\*’, ‘\*’}.

Input data (parameter): letter – the user’s incorrect guess

Output data (return): TRUE (1) if a \* is replaced with the

guessed letter. FALSE (0) if the guessed letter is

already in the attempt array (repeated guess).

Output data (parameter): attempts – attempt array with a star

replaced with the guessed letter.

Dependencies: none

\*/

1. (5 points) Function checkVictory()

/\*

Function: checkVictory()

Process: check whether the output array is completely filled

with letters (no underscores).

Input data (parameter): output\_array – the array filled with

underscores or the correct guesses

size – the length of the output\_array

Output data (return): TRUE (1) if the output array is

completely filled with letters (no underscore).

FALSE (0) if there is at least one underscore in the

output array.

Output data (parameter): none

Dependencies: none

\*/

1. (5 points) Function checkDefeat()

/\*

Function: checkDefeat()

Process: check whether the attempts array is completely

filled with letters (no stars).

Input data (parameter): attempts – the array filled with

stars (‘\*’) or the incorrect guesses

Output data (return): TRUE (1) if the output array is

completely filled with letters (no stars).

FALSE (0) if there is at least one star (‘\*’)in the

output array.

Output data (parameter): none

Dependencies: none

\*/

1. (5 points) Function play()

/\*

Function: play()

Process: controls the game. Generates a secret word, creates

the array of attempts, creates the output array, and

asks for the user guesses. For each guess, the

function decides whether the letter is in the secret

word or not. If the letter is in the secret word, it

fills the output array, print it out, and check if

the user won. If the letter is an incorrect guess,

it fills the attempts array, prints it out and check

if the user lost. If the user lost, print out the

secret word. The user must be notified in case

of repeated guesses (correct or incorrect) and

prompted for another guess.

Input data (parameter): none

Output data (return): TRUE (1) if the user wins. FALSE (0) if

the user looses.

Output data (parameter): none

Dependencies: getSecretWord, buildOutputArray,

buildAttemptsArray, checkGuess, fillOutputArray,

fillAttemptsArray, checkVictory, checkDefeat,

printf, scanf

\*/

1. (3 points) Function main(): It’s game time!! In the main function, you will print out a title to the program and allow the user to play as many time as they want. Assume that they always want to play the first time. You will call the play() function and, once it is done, print out a message telling the user whether they won or lost (based on the return of play()). Ask the user if they want to play again and start over if they do. If they don’t, ends the game with a clear message.

## Problem 2

**IMPORTANT NOTE:** for this problem, I will NOT provide the function comments. You must create the comments based on the functions description. Adding the appropriate comments will count toward style rubrics.

--

We are going to simulate a battleship game. To do so, in your hw3\_p2.c file, you will create the following:

1. (5 points) a function called printMatrix() that, given a two-dimensional array informed as a parameter (matrix), prints the array in a nice tabular format. For example, the array matrix[3][3] = {{'1', '2', '3'}, {'1', '2', '3'}, {'1', '2', '3'}}; the output would be

1 2 3  
1 2 3  
1 2 3

1. (5 points) Create a function called buildBoard() that, given a two-dimensional array of char (10x10) named board and a single character c informed as a parameter, returns the board matrix filled with the character c in every position. For example, if we call the function as buildBoard(board, '0'), the resulting matrix would be:

'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'  
'0' '0' '0' '0' '0' '0' '0' '0' '0' '0'

1. (5 points) a function isWater() that, given a board (which is a matrix created with the buildBoard() function) and the values of a row and a column (i and j, respectively), returns a value that informs whether the position in the board contains water (true if contains water, false otherwise). Water is represented by the character '0' and true/false values are the integers 1 and 0 respectively. The function also returns false if the position i, j doesn’t exist in the board.
2. (5 points) a function getHorizontal() that, given a board (which is a matrix created with the buildBoard() function), an integer array of length 2 (positions) and a ship’s length, changes the array position to contain a position in the board where it is possible to add a ship in the horizontal direction (row). The position array will store the following: position[0] will indicate the row in the board where the ship will go, and position[1] will indicate the column in the board where the ship begins. See the example:

**NOTES:**

* Consider that a ship can be inserted in a particular position if there are enough cells with water ('0') to fit the ship that doesn’t go out of the board’s limits.
* The values for row (position[0]) and column (position[1]) must be randomly generated.

For example, considering the fake board below where '0' means that position contains water and 'S' means that position contains ship (unkown length). Calling the function getHorizontal(fake\_board, positions, 3) will try to add a ship in one of the *rows* that will fill three positions in the board.

fake\_board = {{'0', '0', 'S', '0', '0', '0', '0'},  
 {'0', '0', 'S', '0', '0', '0', '0'},  
 {'0', '0', 'S', '0', '0', '0', '0'}}  
  
//Possible returns  
//position[0]=0, position[1]=3  
//position[0]=0, position[1]=4  
//position[0]=1, position[1]=3  
//position[0]=1, position[1]=4  
//position[0]=2, position[1]=3  
//position[0]=2, position[1]=4

I cannot add a ship in any row with position[1]=0 because a ship of length 3 would hit the ship ('S') in column 2. In the same way, I cannot add a ship in the any row starting in column 5 on (position[1]=5 or position[1]=6) because a ship of length 6 would go outside the board’s limits. Notice that our real boards will have dimension 10x10! this is an example only.

1. (5 points) a function getVertical() that, given a board (which is a matrix created with the buildBoard() function), an integer array of length 2 (positions) and a ship’s length, changes the array position to contain a position in the board where it is possible to add a ship in the horizontal direction (row). The position array will store the following: position[0] will indicate the row in the board where the ship will go, and position[1] will indicate the column in the board where the ship begins. This function follows the same specification as item 4 (function getHorizontal()), the only difference is the orientation of the ship (vertical).

For example, considering the fake board below where '0' means that position contains water and 'S' means that position contains ship (unknown length). Calling the function getVertical(fake\_board, positions, 3) will try to add a ship in one of the *columns* that will fill three positions in the board.

fake\_board = {{'0', '0', 'S', '0', '0', '0', '0'},  
 {'0', '0', 'S', '0', '0', '0', '0'},  
 {'0', '0', 'S', '0', '0', '0', '0'}}  
  
//Possible returns  
//position[0]=0, position[1]=0  
//position[0]=0, position[1]=1  
//position[0]=0, position[1]=3  
//position[0]=0, position[1]=4  
//position[0]=0, position[1]=5  
//position[0]=0, position[1]=6

As we can see, the only row we can start adding ships is row 0 because this fake board has only 3 rows, which is the length of the ship! Thus, for any index greater than 0, the ship won’t fit the board. Additionally, the ship fit any columns other than 2, where we already have a ship. Notice that our real boards will have dimension 10x10! this is an example only.

1. (5 points) a function placeShipHorizontal() that, given a board, the ship’s length and the initial position (generated by the getHorizontal() function), places a ship in the board in the horizontal orientation. The ship starts at the initial position and occupies length places in the given row. A ship is represented by the character that indicates it’s length (**as character! not as integer!** make sure you cast the length into char before you add to the board!). For example, considering the following test code:

buildBoard(board, '0')  
placeShipHorizontal(board, 5, 0, 0) //length 5, row 0, column 0  
placeShipHorizontal(board, 3, 5, 4) //length 3, row 5, column 4  
printMatrix(board)

The output would be:

5 5 5 5 5 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 3 3 3 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0

1. (5 points) a function placeShipVertical() that, given the board, the ship’s length and the initial position (generated by the getVertical() function), places a ship in the board in the vertical orientation. The ship starts at the initial position and occupies length places in the given column. A ship is represented by the character that indicates it’s length (**as character! not as integer!** make sure you cast the length into char before you add to the board!). For example, considering the following test code:

tab1 = buildBoard('0')  
placeShipHorizontal(tab1, 5, 0, 0) //length 5, row 0, column 0  
placeShipHorizontal(tab1, 3, 5, 4) //length 3, row 5, column 4  
printMatriz(tab1)

The output would be:

5 0 0 0 0 0 0 0 0 0  
5 0 0 0 0 0 0 0 0 0  
5 0 0 0 0 0 0 0 0 0  
5 0 0 0 0 0 0 0 0 0  
5 0 0 0 0 0 0 0 0 0  
0 0 0 0 3 0 0 0 0 0  
0 0 0 0 3 0 0 0 0 0  
0 0 0 0 3 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0

1. (5 points) a function fillBoard() that randomly fill the board with ships. The function must add the following ships to the board:

* 1 aircraft carriers that occupies five positions in the board(length=5)
* 2 tankers, each of them occupying four positions in the board (length=4)
* 3 destroyers, each of them occupying three positions in the board (length=3)
* 4 submarines, each of them occupying two positions in the board (length=2)

To complete this function, **follow the steps below**:

1. create an array containing all the lengths of ships to be placed:

length = {5, 4, 4, 3, 3, 3, 2, 2, 2, 2}

1. For each ship in the array of lengths:

* decide whether the ship will be placed in the horizontal or vertical orientation. To do so, randomly generate a number 0 or 1, where:
  + 0 indicates that you have to place the ship in the horizontal orientation;
  + 1 indicates that you have to place the ship in the vertical orientation.
* get valid initial positions (i, j) for the ship using the getHorizontal() or getVertical() functions.
* place the ship in the [i][j] position using the appropriate function (placeShipHorizontal() ou placeShipVertical())

1. (5 points) a function shoot() that receives the following data as parameters: two boards (solution and game) and a position in the boards (i, j) representing the position in which the player wants to shoot.

The function needs to verify the solution board to determine whether the specified position (i, j) has water (‘0’) or a ship (a value between ‘2’-‘5’). Based on that, the function will change the game board following these rules:

* game[i][j] receives ‘W’ if the solution board contains water (‘0’);
* game[i][j] receives ‘S’ if the solution board contains a ship(‘2’, ‘3’, ‘4’, or ‘5’).

1. (5 points) a function checkVictory() that, given a solution and a game board, returns 1 (true) if the player has shot all the positions containing ships; that means, for every possition in the solution that contains a number > 0, there is a ‘S’ in the corresponding position of the game board. Return 0 (false) if the game board has any missing ship positions.
2. **(Extra-credit: 5 points)** It’s game time!! Write a main() function to assemble all the function calls and make the game work!

*Note: in this version of the battleship game, the user (human) will shoot the ships in the computer’s board, but the computer doesn’t fight back (the user doesn’t have a board to place their own ships).*

Follow these steps to complete your main() program:

**Part I - Setup**

1. use the function buildBoard() to create the solution board with water ('0') in every position;
2. use the function buildBoard() to create the game board with an 'X' in every position (the user didn’t shoot any positions yet)
3. use the function printMatrix() to print the game board, so that the user can chose where to shoot
4. use the function fillBoard() to fill the solution board with the ships in random positions

**Part II - the game**

1. While the user didn’t win and did not give up, do the following:
   * ask for the number of the row (i) where the user wants to shoot (0-9)
   * ask for the number of the column (j) where the user wants to shoot (0-9)
   * use the function shoot() to shoot at the position [i][j] given by the user
   * use the function printMatrix() to print the game board, to show the user the outcome of the shooting
   * check if the user won and print a congratulations message.

After that, Part II repeats until the user win or give up. The user gives up if they type -1 at any input. If the user gives up, print the solution board, so that they know where the ships they did not shoot were.

## Have fun!