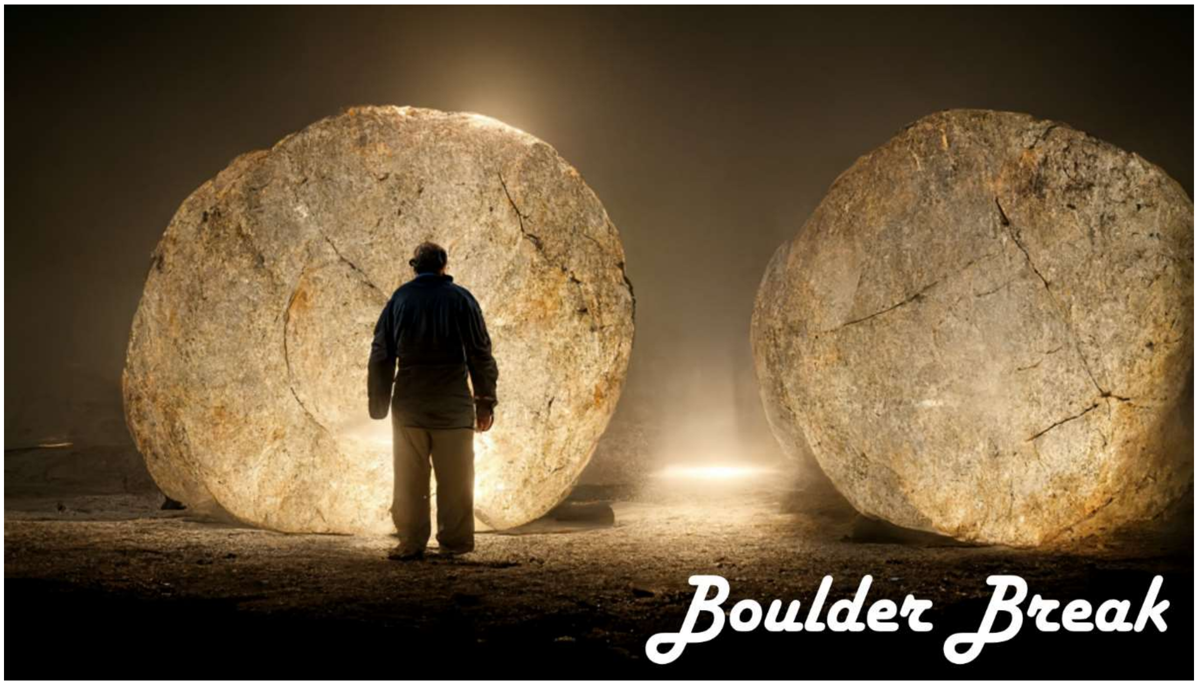
ENGGEN 131 – Semester Two – 2022

C Programming Project



Deadline: 11:59pm, Saturday 22nd October
Worth: 10% of your final grade

An important note before we begin…

Welcome to the C Programming project for ENGGEN131 2022!

This project is organized around an interactive text-based puzzle game that can be implemented as a series of related tasks. When combined, the tasks will complete the implementation for the game. For each task there is a problem description, and you must implement *one function* to solve that problem. In addition to the required function, you may define *other functions* which the required function calls upon (i.e. so-called *helper* functions). Using helper functions is often a useful way to organize your code, as several simpler functions can be easier to read and understand than one complex function.

Do your very best, but don't worry if you cannot complete every task. You will get credit for each task that you do solve.

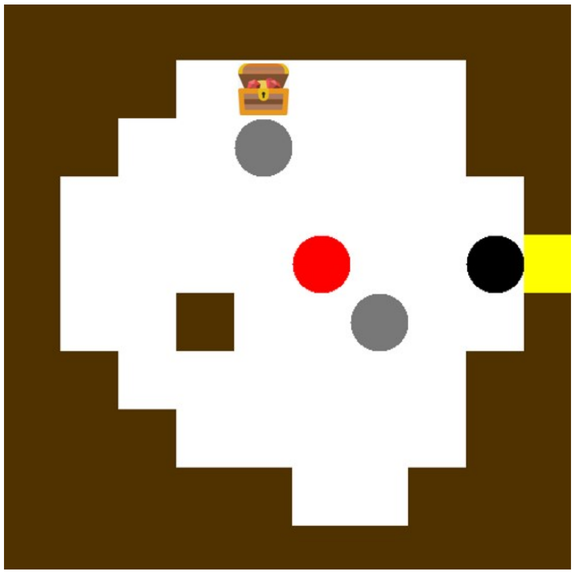
IMPORTANT - Read carefully
This project is an assessment for the ENGGEN131 course. It is an **individual** project. You do not need to complete all of the tasks, but the tasks you do complete should be an accurate reflection of your capability and effort. You may discuss ideas in general with other students, but writing code must be done by yourself. *No exceptions.* You must not give any other student a copy of your code in any form – and you must not receive code from any other student in any form. There are absolutely NO EXCEPTIONS to this rule.

Please follow this advice while working on the project – the penalties for academic misconduct (which include your name being recorded on the misconduct register for the duration of your degree, and/or a period of suspension from Engineering) are simply not worth the risk.

Acceptable	***Unacceptable***
• Describing problems you are having to someone else, either in person or on Piazza, without revealing *any* code you have written • Asking for advice on how to solve a problem, where the advice received is general in nature and does not include *any* code • Discussing with a friend, away from a computer, ideas or general approaches for the algorithms that you plan to implement (but *not* working on the code together) • Drawing diagrams that are illustrative of the approach you are planning to take to solve a particular problem (but *not* writing source code with someone else)	• Working at a computer with another student • Writing code on paper or at a computer, and sharing that code in any way with anyone else • Giving or receiving any amount of code from anyone else in any form • In short, there are no exceptions to the rule: you cannot share code with others.

OK, now, on with the project…

A brief background



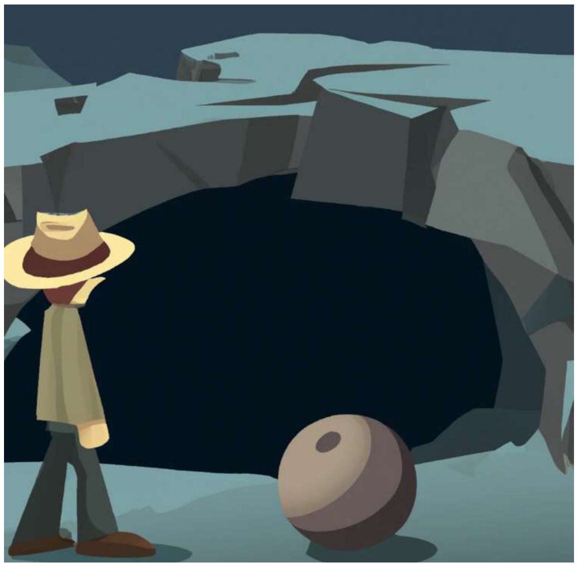
Boulder break is a 2-dimensional puzzle game where the goal is to help an adventurer collect treasure and escape from a cave by pushing boulders into holes that may block the exit. In the diagram shown on the right, the adventurer (trying to escape the cave) is represented by the red circle. The gray coloured circles are the boulders that the adventurer can push. The black circle is the hole which is currently blocking the yellow exit, and into which the boulders can be pushed. The small chest containing jewels is the treasure.

To collect treasure, the adventurer simply needs to move onto its location.

If the adventurer steps onto a hole, they will fall in and meet an untimely end. That will be the end of the game.

To push a boulder, the adventurer must position themselves next to the boulder and can only push the boulder in the direction of their movement. When the adventurer pushes a boulder, the boulder will continue to roll in a straight line until it hits an obstacle (either part of the cave, a treasure chest or another boulder). The position of the adventurer does not change when they push a boulder. After the boulder has moved, the adventurer can move.

To escape the cave, the adventurer must move onto the yellow exit. If the exit is blocked by a hole, they must push a boulder into the hole which will fill it in, allowing them to access the yellow exit.



Task One: "What's the story?" (10 marks)

Every good game has an intriguing backstory!

For this task, you should write a story that provides some background to the game. Why is the adventurer stuck in the cave? How did the treasure get there? What are the boulders and holes doing in the cave? These are just ideas – you can write this background in any way that you like – try to be creative!

You must write a function called `InitialiseStory()`:

```
void InitialiseStory(char* paragraph)
```

which takes a pointer to a character as input. This pointer will refer to an array of characters into which your story should be written.

The only requirements for your story are the following:

- The story must be *longer* than 100 printable characters (i.e. the letters that appear when printed, excluding new lines)
- The underlying array which is passed to the function will be of length 2000. This means that the *maximum* length your story can be is 1999 characters (as there must be space for the null terminating character)
- The story must be broken into lines (using the new line character). The maximum length of any line in your story is 60 printable characters (i.e. the letters that appear when printed, excluding new lines)

To illustrate how this function should work, consider the following `main()` function for a short program that calls the `InitialiseStory()` function:

```
int main(void)
{
      char story[2000];
      InitialiseStory(story);
      printf("%s\n", story);

      return 0;
}
```

Here is one possible output that might be generated by the program above:

```
A brave adventurer is stuck in a cave! There is treasure
and danger! Collect the treasure and then escape!
```

Notice that in this example the story satisfies the requirements because:

- It is longer than 100 printable characters (a total of 105 printable characters appear on the screen, 56 on the first line beginning "A brave…" and 49 on the second line beginning "and danger!...")
- The total number of characters that are stored in the array is less than 2000 (in this case, a total of 108 characters are stored in the array: the 105 printable characters, one new line character at the end of each of the two lines, and the null terminating character)
- The maximum length of any line is 60 printable characters (in this case, the first line is 56 printable characters and the second line is 49 printable characters)

However, this example story is not very interesting! You can surely do better...

Task Two: "A bird's eye view" (10 marks)

The goal of this task is to produce a printable representation of a cave. You will need to define a function called `PrintCave()`:

```
void PrintCave(int cave[CAVE_SIZE][CAVE_SIZE])
```

The cave is represented by a square 2-dimensional grid. The size of the grid is given by one constant: `CAVE_SIZE`. As the name of this constant suggests, it indicates how many *rows* and *columns* the grid contains. Initially, this constant is set to the following value:

```
#define CAVE_SIZE 10
```

To begin with, every element of the array will be one of two special values: 0 or 1.

The value 0 represents 'empty space' and should be displayed using the space character. The value 1 represents a solid wall and should be displayed using the '#' character.

For example, consider the two variables below that each store a 2-dimensional array representing a cave:

```
      int caveA[CAVE_SIZE][CAVE_SIZE] = {
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
      };

      int caveB[CAVE_SIZE][CAVE_SIZE] = {
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
      };
```

The first cave, `caveA`, is just one big solid wall! Not much fun for exploring. The second cave, `caveB`, consists just of an outer boundary of a solid wall with empty space in the middle.

Consider the following code which prints each of these two caves by calling the `PrintCave()` function:

```
  printf("Printing Cave A:\n");
  PrintCave(caveA);

  printf("Printing Cave B:\n");
  PrintCave(caveB);
```

The output of this code should be as follows:

```
Printing Cave A:
##########
##########
##########
##########
##########
##########
##########
##########
##########
##########
Printing Cave B:
##########
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
##########
```

Notice that every line of output produced by the `PrintCave()` function should end with a single new line character.

Task Three: "Landscaping" (10 marks)

Now that you can print a text-based representation of a cave from a 2-dimensional array, it is time to find a more convenient way of initializing the array.

We will do this using a string. The string will only contain three different characters: '0', '1' and 'E'. Here is an example of such a string stored in a variable called `layout` (it is printed over two lines below, for space reasons, but is really just one long string on a single line):

```
      char layout[200] = "1111111111111000001111000000111000000001100000
                         000E1001000001110000001111100000111111100111111111111";
```

You will need to define a function called `InitialiseCave()` that takes two inputs: the 2-dimensional array that is being initialized and the string that contains the layout information:

```
  void InitialiseCave(int cave[CAVE_SIZE][CAVE_SIZE], char *layout)
```

This function should initialize each element of the array using the values in the string. The elements of the array should be initialized from left to right, and top to bottom. The values in the string define the layout of the cave. The value '0' represents 'empty space', the value '1' represents a solid wall and the value 'E' represents an 'exit'.

When the cave is printed, the 'exit' should be displayed using the space character (this is identical to the 'empty space', so the 'exit' will look just like an 'empty space' when printed). You will need to modify your `PrintCave()` function to ensure the exit is printed correctly.

For example, consider the code below which initializes the array and then prints the cave:

```
      char layout[200] = "1111111111111000001111000000111000000001100000
                         000E1001000001110000001111100000111111100111111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);
      PrintCave(cave);
```

The output produced by this code should be as follows:

```
##########
###     ##
##      ##
#        #
#
#  #     #
##      ##
###     ##
#####  ###
##########
```

In this case, the 'exit' position (which was represented in the layout string by the character 'E') appears to look like an 'empty space' in the middle of the right hand wall.

You can assume that the input string will only contain the characters '0', '1' and 'E'. You can also assume that the length of the string will perfectly match the size of the array (that is, the number of characters in the string will equal the number of elements in the 2-dimensional array).

Task Four: “No more gaps” (10 marks)

Now that we can initialize the array easily, we must check that the cave we have created is valid. In order to play the game, the cave must have the following properties:

- There can be no gaps or ‘empty space’ in the *outside border* of the cave (i.e. the outer rows and columns). In other words, each element of the outside border of the cave must be a solid wall or be an ‘exit’ position.
- There must be *exactly one* exit position in the entire cave.
- The exit position must be on the border of the cave (i.e. on one of the outer rows or columns) and it must not be on a corner (i.e. not in the top left, bottom left, top right or bottom right corners).

The `InitialiseCave()` function does not check for these things, and so we must check them now. We will check for each of these three conditions separately.

Start by writing a function called `IsBorderComplete()`:

```
  int IsBorderComplete(int cave[CAVE_SIZE][CAVE_SIZE])
```

This function returns true if every element of the outer border of the maze consists only of solid walls or ‘exit’ positions – in other words, no element on the outer border is ‘empty space’. For example, consider the following code that creates and prints a cave and then calls this function:

```
      char layout[200] = "1111111111E000000001100000000110000000001
              1000000001100000000110000000011000000001100000000111111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);
      PrintCave(cave);
      printf("Complete? %d\n", IsBorderComplete(cave));
```

The output should be as follows:

```
##########
         #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
##########
Complete? 1
```

In this case, the gap that appears near the top left of the cave is an ‘exit’ position, so this is valid.

In contrast, now consider this example:

```
      char layout[200] = "1111111111000000000110000000011000000001
                100000000110000000011000000001100000000110000000011111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);
      PrintCave(cave);
      printf("Complete? %d\n", IsBorderComplete(cave));
```

This time, the ‘exit’ position has been replaced with an ‘empty space’. The output now is:

```
##########
         #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
##########
Complete? 0
```

Even though the printed cave *looks* the same as before, this time the gap near the top left corner is actually an ‘empty space’ (not an ‘exit’) and so the cave is invalid.

Task Five: "The one and only" (10 marks)

Next, a valid cave must have *only one* exit. Define a function called `IsExitUnique()` which checks that there is a single 'exit' position in the 2-dimensional array:

```
int IsExitUnique(int cave[CAVE_SIZE][CAVE_SIZE])
```

If there is *exactly* one 'exit' position, then the function should return true, otherwise it should return false. Consider the following example:

```
      char layout[200] = "1111111111100000000110000000011000000001
                1EEEEEEEE1100000000110000000011000000001100000000111111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};
      InitialiseCave(cave, layout);
      PrintCave(cave);
      printf("Unique? %d\n", IsExitUnique(cave));
```

The output of this code is:

```
##########
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
##########
Unique? 0
```

This is because there are many 'exit' positions on row 4 of the cave (see the 'E' characters in the string). In contrast, consider the following example:

```
      char layout[200] = "1111111111100000000110000000011000000001
                1000E0000110000000011000000001100000000110000000011111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};
      InitialiseCave(cave, layout);
      PrintCave(cave);
      printf("Unique? %d\n", IsExitUnique(cave));
```

This time, there is only one 'exit' position in the entire 2-dimensional array. Even though the output of the cave *looks* exactly the same as before (due to the way that 'exit' positions appear), this time the exit is unique and so the output is as shown on the right:

```
##########
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
##########
Unique? 1
```

Task Six: "Exit this way please" (10 marks)

Finally, the cave for the game is only valid if the single 'exit' position is positioned on the outside border of the cave. It cannot be internal to the cave. In addition, the 'exit' position must be accessible to the player who moves horizontally and vertically, and so it cannot be positioned in one of the four corners.

Define a function called `IsExitAccessible()`:

```
int IsExitAccessible(int cave[CAVE_SIZE][CAVE_SIZE])
```

which checks the location of the 'exit'. If the 'exit' is on the outside border and is not on a corner, then the function should return true, otherwise it should return false. For example, consider the following code:

```
      char layout[200] = "1111111111100000000110000000011000000001
                10000000011000000001100000000110000000011000000001111111111E";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);
      PrintCave(cave);
      printf("Accessible? %d\n", IsExitAccessible(cave));
```

In this case, the 'exit' position is in the bottom right corner and so is not accessible:

```
##########
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#########
Accessible? 0
```

In contrast, consider the following example where the ‘exit’ position is on the lower border and so is accessible:

```
char layout[200] = "1111111111100000000110000000011000000001
          1000000001100000000110000000011000000001100000000111111111E1";
int cave[CAVE_SIZE][CAVE_SIZE] = {0};

InitialiseCave(cave, layout);
PrintCave(cave);
printf("Accessible? %d\n", IsExitAccessible(cave));
```

In this case, the output would be as follows:

```
##########
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
######## #
Accessible? 1
```

Task Seven: "Treasure and danger" (10 marks)

Now that we have a valid cave, with a solid border, and one accessible 'exit' positioned on the border, we can add the remaining elements to the cave.

There are four types of elements that can be added:

- The **player** – this is the adventurer that the user will control with the keyboard and is displayed as 'X'. There must be *exactly one* player in the cave
- A **hole** – this is a danger, and if the player steps onto the hole the game will be over. A hole is shown as '*' and there can be zero or more holes in the cave
- A **treasure** – this is a reward, and if the player steps onto the treasure they will collect it (meaning the treasure will disappear). There can be zero or more treasures in the cave.
- A **boulder** – this can be pushed by the player, and if a boulder falls into a hole then the hole (and the boulder) disappears. There can be zero or more boulders in the cave.

Define a function called `AddItem()` which adds one of the above listed items into the cave at a specified location:

```
void AddItem(int cave[CAVE_SIZE][CAVE_SIZE], int row, int col, char *element)
```

The first input to the function is the cave, the next two inputs are the *row* and *column* location of where the item should be placed, and the fourth input is a string representing the type of item. This string will be one of the following:

- "player"
- "hole"
- "treasure"
- "boulder"

A new item can only be placed into the cave is there is 'empty space' at that location. If the row and column position is invalid – for example, if it is outside the bounds of the 2-dimensional array or if there is already an item or solid wall at the location, then nothing should happen and the cave should not be modified.

In addition, there can only be one player in the cave. So, if the `AddItem()` function is attempting to add a player, and yet there is already a player at a different location in the cave, then the cave should not be modified.

Consider the following example, which adds several items to the cave and then prints it:

```
char layout[200] = "1111111111100000000110000000011000000001
          100000000E10000000011000000001100000000110000000011111111111";
int cave[CAVE_SIZE][CAVE_SIZE] = {0};

InitialiseCave(cave, layout);

AddItem(cave, 7, 3, "player");

AddItem(cave, 2, 2, "boulder");
AddItem(cave, 2, 3, "boulder");
AddItem(cave, 4, 6, "boulder");
AddItem(cave, 5, 7, "boulder");
AddItem(cave, 6, 8, "boulder");

AddItem(cave, 7, 1, "hole");
AddItem(cave, 8, 3, "hole");

AddItem(cave, 6, 4, "treasure");
AddItem(cave, 3, 5, "treasure");
AddItem(cave, 1, 1, "treasure");

PrintCave(cave);
```

The output from this code would be:

```
##########
#+       #
# OO     #
#    +   #
#     O
#      O #
#   +   O#
#* X     #
#  *     #
##########
```

Consider the following example where some invalid inputs are provided to the `AddItem()` function:

```
      char layout[200] = "1111111111100000000110000000011000000001
                100000000E10000000011000000001100000000110000000011111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);

      AddItem(cave, 0, 0, "player");
      AddItem(cave, 5, 5, "player");
      AddItem(cave, 6, 6, "player");

      AddItem(cave, 2, 2, "boulder");
      AddItem(cave, 2, 2, "boulder");
      AddItem(cave, 2, 2, "boulder");
      AddItem(cave, 2, 2, "boulder");
      AddItem(cave, 1000, 1000, "boulder");

      PrintCave(cave);
```

In this case, the player ends up at position (5, 5). This is because (0, 0) is invalid (these is a solid wall at that location) and (6, 6) is also invalid because the player has already been added at position (5, 5). Similarly, only one boulder appears in the cave because the other positions are invalid (either repeated or out of bounds).

```
##########
#        #
# O      #
#        #
#
#    X   #
#        #
#        #
#        #
##########
```

Task Eight: "Around and around" (10 marks)

It may be useful to view the cave from a different perspective!

Define a function called `RotateCave()` that modifies the cave by rotating it 90 degrees in a clockwise direction. All of the contents in the cave should also rotate correctly by 90 degrees:

```
void RotateCave(int cave[CAVE_SIZE][CAVE_SIZE])
```

You may find it helpful to declare a temporary array in the `RotateCave()` function to help with the rotation. You could use a temporary variable such as:

```
int temp[CAVE_SIZE][CAVE_SIZE];
```

If the cave is rotated four times using this function, that represents a 360 degree rotation and the cave will return to its original position. Consider the following example which creates a cave and then rotates it four times, printing it after each rotation:

```
      char layout[200] = "1111111111100000000110000000011000000001
                100000000E100000000110000000011000000001100000000111111111";
      int cave[CAVE_SIZE][CAVE_SIZE] = {0};

      InitialiseCave(cave, layout);

      AddItem(cave, 5, 5, "boulder");
      AddItem(cave, 5, 6, "boulder");
      AddItem(cave, 5, 7, "boulder");
      AddItem(cave, 2, 2, "player");

      printf("Original position:\n");
      PrintCave(cave);

      printf("First rotation:\n");
      RotateCave(cave);
      PrintCave(cave);

      printf("Second rotation:\n");
      RotateCave(cave);
      PrintCave(cave);

      printf("Third rotation:\n");
      RotateCave(cave);
      PrintCave(cave);

      printf("Fourth rotation:\n");
      RotateCave(cave);
      PrintCave(cave);
```

The output from this code is shown on the following page:

```
Original position:
##########
#        #
# X      #
#        #
#
#    OOO #
#        #
#        #
#        #
##########
First rotation:
##########
#        #
#      X #
#        #
#        #
#   O    #
#   O    #
#   O    #
#        #
##### ####
Second rotation:
##########
#        #
#        #
#        #
# OOO    #
         #
#        #
#      X #
#        #
##########
Third rotation:
#### #####
#        #
#    O   #
#    O   #
#    O   #
#        #
#        #
# X      #
#        #
##########
Fourth rotation:
##########
#        #
# X      #
#        #
#
#    OOO #
#        #
#        #
#        #
##########
```

Task Nine: “Time to get moving”	(10 marks)

The last task is to implement a function to move the player. Define a function called `MakeMove()`:

```
int MakeMove(int cave[CAVE_SIZE][CAVE_SIZE], char move)
```

This function is passed two inputs: the current configuration of the cave and a character which represents the direction in which the adventurer wants to move. This character will either by ‘w’ (up), ‘a’ (left), ‘s’ (down) or ‘d’ (right).

The player should move in the direction specified. The basic rules are as follows:

- The player can move up, left, right or down by one position at a time.
- The player cannot move into or through the walls of the cave.
- To push a boulder, the player must be adjacent to the boulder and must move in the direction of the boulder. In order for the boulder to move, the boulder must be able to roll, in the same direction, from the square that it was in (i.e. if the player moves “up” towards a square with a boulder, then the boulder must have at least some ‘empty space’ above it in which to roll). The location of the player does not change when they push a boulder.
- When a boulder rolls into a hole, both the hole and the boulder disappear (i.e. ‘empty space’ appears at the previous location of the hole)
- A boulder, once pushed, will continue to move in a straight line until it hits an obstacle (either a hole, a treasure, another boulder or a cave wall)
- You can assume that the boulder will never be pushed through the exit (in other words, you don’t need to handle the case that a boulder moves onto the ‘exit’ location).

You must implement the `MakeMove()` function such that for any input array, and any given directional move, the array will be updated correctly to reflect that move.

You can assume that the cave will be valid (that is, it will meet the rules for validity described in Tasks 4, 5 and 6). In other words, the border of the cave will be a solid wall with the exception of a single accessible ‘exit’ position.

Consider the examples below which illustrate how the cave should change following the specified actions.

Starting array (i.e. input)	*Move*	*Ending array (i.e. output)*
`##########`		`##########`
`### +   ##`		`### +   ##`
`##  O   ##`		`##  O   ##`
`#        #`		`#        #`
`#    X  * `	‘s’ = down	`#       * `
`#  #  O  #`		`#  # XO  #`
`##      ##`		`##      ##`
`###     ##`		`###     ##`
`#####  ###`		`#####  ###`
`##########`		`##########`
`##########`		`##########`
`### +   ##`		`### + O ##`
`##  O   ##`		`##  O   ##`
`#        #`		`#        #`
`#       * `	‘w’ = up	`#       * `
`#  #  O  #`		`#  #     #`
`##    X ##`		`##    X ##`
`###     ##`		`###     ##`
`#####  ###`		`#####  ###`
`##########`		`##########`
`##########`		`##########`
`### +   ##`		`### +   ##`
`##  O   ##`		`##  O   ##`
`#        #`		`#        #`
`#       * `	‘a’ = left	`#       * `
`#  #  OX #`		`#  #O  X #`
`##      ##`		`##      ##`
`###     ##`		`###     ##`
`#####  ###`		`#####  ###`
`##########`		`##########`
`##########`		`##########`
`###     ##`		`###     ##`
`##      ##`		`##      ##`
`#        #`		`#        #`
`#  XO   * `	‘d’ = right	`#  X      `
`#  #O    #`		`#  #O    #`
`##      ##`		`##      ##`
`###     ##`		`###     ##`
`#####  ###`		`#####  ###`
`##########`		`##########`

The `MakeMove()` function must return an integer. This returned value must be either 0, 1 or 2. The following conditions determine what value should be returned.

Return value	*Situation*
2	The player exited the cave by moving onto the ‘exit’ position. When this happens, the cave is printed showing the player at the ‘exit’ position.
1	The player fell into a hole. When this happens, the cave is printed showing the hole only (the player disappears!).
0	The player did not exit the cave or fall into a hole on this move.

A program has been provided to you that you can use to test all of your functions for this project, including your `MakeMove()` function. This program allows you to play the game, and takes care of reading the input move from the user. If the game is over, for example the player moves onto the ‘exit’ position, then a message is printed:

```
Enter move: d
##########
###     ##
##      ##
#        #
#        X
#  #O    #
##      ##
###     ##
#####  ###
##########


CONGRATULATIONS!!
You escaped!
```

See the “**Resources**” section in this document, which describes the resource files provided to you including the source file `boulder_break.c` which you can use to play the game.

Task Ten: “A different size”	(10 marks)

Actually, you may already be finished!

You should have implemented all of the previous functions so that they work correctly for a cave of any size (as defined by the `#define` constant, `CAVE_SIZE`).

If you modify this constant definition, for example change it to 5:

```
#define CAVE_SIZE 5
```

then the game should still work exactly as described. For this final task, you should check that this is indeed the case. For example, consider the following constant definition:

```
#define CAVE_SIZE 5
```

and the code below which sets up the game on a 5x5 array:

```
	char layout[200] = "11111100011000E1000111111";
	int cave[CAVE_SIZE][CAVE_SIZE] = {0};

	InitialiseCave(cave, layout);

	AddItem(cave, 2, 3, "boulder");
	AddItem(cave, 3, 1, "treasure");
	AddItem(cave, 1, 2, "player");
```

Starting with this cave, a complete game may look like the following:

```
#####
# X #
#  O
#+  #
#####

Enter move: a
#####
#X  #
#  O
#+  #
#####

Enter move: s
#####
#   #
#X O
#+  #
#####
```

```
Enter move: s
#####
#   #
#  O
#X  #
#####

Enter move: d
#####
#   #
#  O
# X #
#####

Enter move: d
#####
#   #
#  O
#  X#
#####

Enter move: w
#####
#  O#
#
#  X#
#####

Enter move: w
#####
#  O#
#  X
#   #
#####

Enter move: d
#####
#  O#
#   X
#   #
#####


CONGRATULATIONS!!
You escaped!
```

Resource files

You should begin by downloading the resource file from Canvas: "**Project2Resources.zip**".

Inside this archive you will find two source files:

- `project2.c`
- `boulder_break.c`

You should begin with "`project2.c`". This program contains a very simple `main()` function which does the following:

```
char story[2000];
char layout[200] = "1111111111111000001111000000111000000001
          100000000E10010000001110000000111110000011111110011111111111";
int cave[CAVE_SIZE][CAVE_SIZE] = {0};

InitialiseStory(story);
InitialiseCave(cave, layout);
PrintCave(cave);
```

If you compile and run this program, you will see the following output:

```
C:\MyFiles\Project2Resources> project2
Define the InitialiseStory() function
Define the InitialiseCave() function
0 1
Define the PrintCave() function
0
```

If you examine the source file, you will see that the `InitialiseStory()`, `PrintCave()` and `InitialiseCave()` functions have been defined, but these definitions are *incorrect*. You should correctly implement these three functions, as well as the other required functions for this project. You should define your own tests in the `main()` function to test that your code is working.

When you have finished all of the tasks, you should place your function definitions into the "`boulder_break.c`" file. This program provides a more complete `main()` function, which implements the full game. Playing the game is a simple, robust and fun way to test your function definitions. Try to come up with some challenging caves!

And that's all there is to it - good luck and have fun coding!

ENGGEN131 C Project
Paul Denny
October 2022