Changelog

Rev.	Date	Description	
V1.3	2024/11/18	P.15 []	Fixed Typo of "Column" in vertical match message
		P.22 [6.3.2]	Fixed "FindAndRemoveMatch" function name typo
		P.15,16 [5.5.4] P.17 [5.6] P.20 [6.3.1]	Fixed Missing "=====" in some gameplay demo
v1.2	2024/11/11	P.10 [5.2 2(b)]	Fixed Printout typo of printGameBoard()
		P.13 [5.5.1.1/ 5.5.1.3]	Fixed the wrong error message output (extra \n)
		P.15 [5.5.4]	Fixed typo in "No Match found!" output
v1.1	2024/11/06	P.5 [4]	Fixed the wrong coloring of matching candies in first round.
		P.14 [5.5.2]	Added the checking of empty cells for *Target cell* at swap.
		P.14 [5.5.4]	Added clarification on how findAndRemoveMatch() checking order.
		P.22 [6.3.2]	Added clarification on how the cascade matching checking order.
		P.25 [8]	Elaborated the grading specification.

ENGG1110 Problem Solving by Programming

The Chinese University of Hong Kong

2024-2025 Term 1

Project

Due Date: 2024/12/04 (Wed) 23:59

1. Introduction

Candy Crush is a popular match-three puzzle game in the world, originally released in 2012 for Facebook and later adapted for mobile platforms.

In the basic gameplay, the gameboard consists of a grid filled with various types of candies, each represented by different shapes and colors. The player's goal is to create matches of three or more identical candies by swapping adjacent one. When a match is made, the matched candies are cleared from the gameboard, and new candies will fall down to fill the empty spaces. This sometimes leads to "chain reactions", where new matches are automatically created as the board refills.

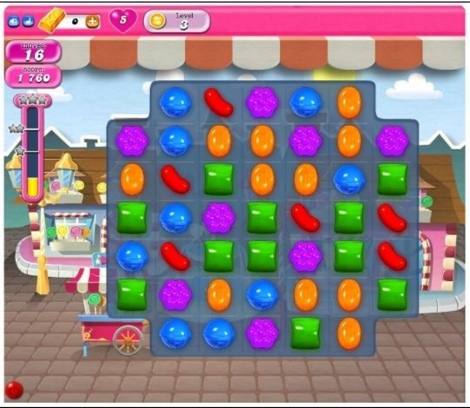


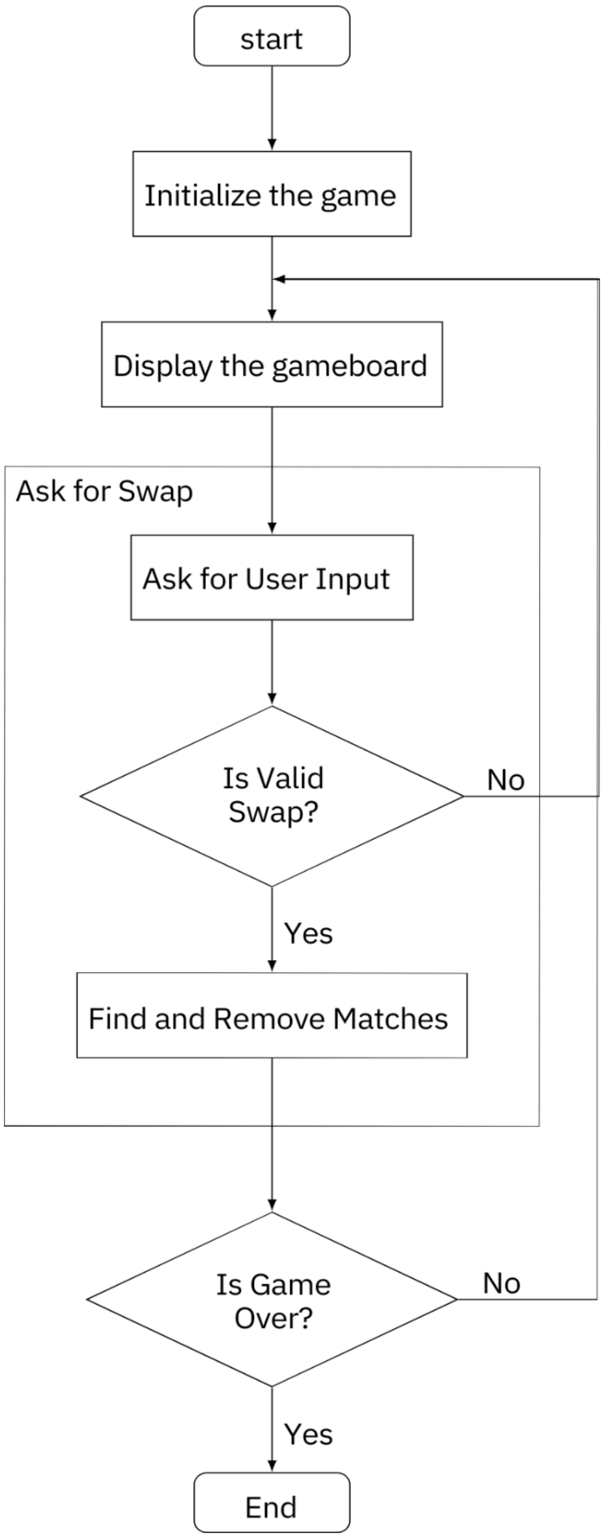
Figure 1. Sample Gameplay. Snapshots from https://www.flickr.com/photos/alper/10330168576

In this project, you will develop a Candy Crush game using the C programming language. The project is structured in two phases: Part I focuses on implementing the basic version of the game, while Part II builds upon the first part by adding advanced features and additional gameplay mechanics.

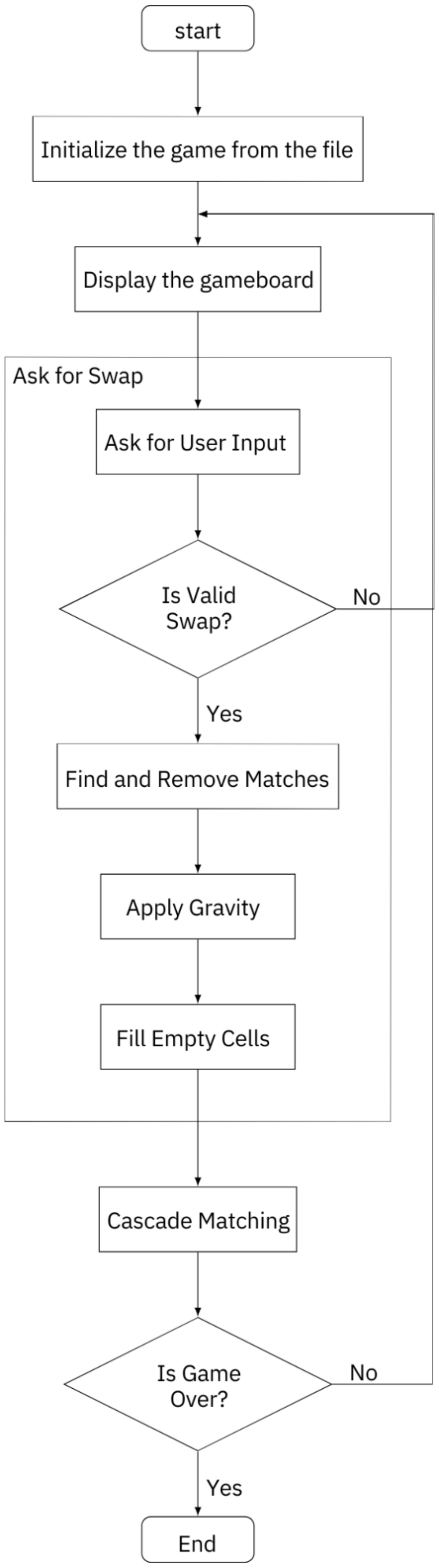
You are required to complete the given source code `main.c` without modifying any existing code (except otherwise specified) or introducing new libraries. Marks will be deducted from every modification.

2. Program Flow

Part I



Part II


start
Initialize the game from the file
Display the gameboard
Ask for Swap
Ask for User Input
Is Valid Swap?
No
Yes
Find and Remove Matches
Apply Gravity
Fill Empty Cells
Cascade Matching
Is Game Over?
No
Yes
End

3. Suggested Project Schedule

Part I (60%)	
Week 10	InitGameBoard(), printGameBoard()
Week 11	AskForSwap()- Input/Validation, swap()
Week 12	FindAndRemoveMatch(), isGameOver()

Part II (40%)	
Week 13/14	File I/O, applyGravity, fillEmpty(),Cascade
Week 14	Cascade

4. Sample Runs for Part I

The following shows several examples of inputs and the resulting candy clearings. User inputs are indicated by **bold, highlighted underlined** text. The matches found by the function are in red font.

```
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Enter the coordinate (row, column) of the candy:5 2
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):L
Vertical Match found at column 2!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
Enter the coordinate (row, column) of the candy:0 2
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):D
Horizontal Match found at row 1!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
Enter the coordinate (row, column) of the candy:0 0
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):L
Move Out of Bound.
Please try again.
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
Enter the coordinate (row, column) of the candy:1 2
Empty Cell Selected.
Please try again.
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
Enter the coordinate (row, column) of the candy:4 5
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):D
Horizontal Match found at row 4!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
Enter the coordinate (row, column) of the candy:2 3
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):R
Horizontal Match found at row 2!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # |   |   |   | @ | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # |   |   |   | @ | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
Enter the coordinate (row, column) of the candy:0 4
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):R
Horizontal Match found at row 0!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   |   |   | # |
 1|   |   |   | # | # | % |
 2| # |   |   |   | @ | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   |   |   | # |
 1|   |   |   | # | # | % |
 2| # |   |   |   | @ | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
Enter the coordinate (row, column) of the candy:1 5
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):U
Horizontal Match found at row 1!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   |   |   | % |
 1|   |   |   |   |   |   |
 2| # |   |   |   | @ | * |
 3| % | * |   | # | * | @ |
 4| # | * |   |   |   |   |
 5| % | # |   | * | % | @ |
Game Over! No more possible moves.
```

5. Detailed Program Design Flow – Part I

In this project, you are required to follow exactly the output format specified. Using other output formats will jeopardize your mark.

This program will

1. Read and initialize the game board from source
2. Display the game board
3. Allow the player to select a candy to swap
    a. The player can only swap two candies at a time
    b. The player will choose a candy by entering the coordinates (row, column)
    c. The player will also input a direction (U for Up, D for Down, L for Left, R for Right) to specify where to swap the selected candy.
4. Find and remove matches
    a. After each swap, check for matches of three or more identical candies in a row or column.
    b. If a match is found, display a message indicating the match's location, remove the matched candies, and update the game board.
5. Check for a game-over condition
    a. Check if no further matches are possible on the gameboard.
    b. If the board has no possible moves left, display the message and end the game.

5.1. Header files, Functions, and Variable Declarations

At the beginning of the `main.c` file, we include two headers, `<stdio.h>` and `<stdlib.h>`.

No other header files or library are allowed in the project.

We define several macros after the header line. Macros are identifiers defined by `#define`, which are replaced by their value before compilation. The first two are H and `W`.

```
#define H 6  // height
#define W 6  // width
```

H is the height of the game board, while `W` is its width. Therefore, our Candy Crush game board is a 6x6 square. In our test cases, we do not have any test cases with varying board size. You may try it if you are interested.

Each cell on the game board stores a candy, with different types of candies represented by numbers. To assist in converting these numbers into their corresponding candy representations, the following global variable is provided.

```
char candies[] = {'*', '#', '@', '%'};
```

No other global variables are permitted. All variables you define must be declared within functions and passed as parameters or return values as necessary. *Violations of this rule will result in a deduction of marks.*

Also, there are a number of helper functions declared in the project. You can add new functions to aid your work, but you **CANNOT modify** (function signatures – function names, return type, parameters), or **delete the given functions**.

5.2. Main Function

The main function is given to control the game flow.

There are some local variables already declared for you. You can declare your own local variables for your use.

Variables	**Usages**
`int board[H][W]`	Stores the candies on the game board.

The main function controls the gameplay as follows:

1. Call `initGameBoard()` to load the initial game board.
2. Enter the main game loop. The loop should continue until the game is over. Here are the tasks in the loop.
    a. The loop should start by printing five equal signs (=====) as a separator for each round, as follows:

    ```
    =====
    New Round:
    ```

    b. Then, call `printGameBoard()` to display the current game board.

    ```
    =====
      | 0 | 1 | 2 | 3 | 4 | 5 |
     0| # | % | @ | * | # | * |
     1| @ | @ | * | # | # | % |
     2| # | % | % | @ | % | * |
     3| % | * | @ | # | * | @ |
     4| # | * | @ | % | % | @ |
     5| % | @ | # | * | % | % |
    ```

    c. Call askForSwap() to execute the swapping logic. If the askForSwap() returns zero (0), indicating the swapping is unsuccessful. In this case, print the following error message:

    ```
    Please try again.\n
    ```

d. Call `isGameOver()` to check if any moves are possible. If no valid moves remain, display a game over message and exit the game loop. In this case, print the following error message:

```
Game Over! No more possible moves.\n
```

5.3. Initializing Game Board

The main() function will first call the `initGameBoard()` function, which initializes the `board[]` array for the later gameplay. In Part I, the content of the array should be read from another hard-coded arrays – `board_samples[]`.

5.4. Print Game Board

The given `printGameBoard()` is useful throughout the program to show the current game board status to the user. You are required to implement this function and print the game board on the screen based on the following format, where ␣ represents a space character that should appear as an actual space in the program output.

First, print a line of five equal signs (=====) as a separator. Then, display the game board with row and column numbers labeled. Each cell should be separated by spaces and vertical bars to clearly distinguish rows and columns.

```
=====
␣␣|␣0␣|␣1␣|␣2␣|␣3␣|␣4␣|␣5␣|
␣0|␣#␣|␣%␣|␣@␣|␣*␣|␣#␣|␣*␣|
␣1|␣@␣|␣@␣|␣*␣|␣#␣|␣#␣|␣%␣|
␣2|␣#␣|␣%␣|␣%␣|␣@␣|␣%␣|␣*␣|
␣3|␣%␣|␣*␣|␣@␣|␣#␣|␣*␣|␣@␣|
␣4|␣#␣|␣*␣|␣@␣|␣%␣|␣%␣|␣@␣|
␣5|␣%␣|␣@␣|␣#␣|␣*␣|␣%␣|␣%␣|
```

There are two types of elements displayed on the grid:

- **Candies**: Use the candies[] character array to convert the numerical values stored in board[][] into the corresponding candy symbols.
- **Empty Space**: Display the space character (' '), which is stored in `board[][]` as ASCII 32 (space).

It is crucial that your output format follows the example **exactly**, including the number of spaces and any other formatting details, as the program will be graded using an autograder. Any deviation in the format could result in a loss of marks.

5.5. Ask for input for swapping

The main function will call `askForSwap()` to handle the game action. The swapping logic will be implemented inside this function. This function will return an integer to indicate whether the swapping is successful or not. (Details of return value)

5.5.1. User Input

5.5.1.1. Coordinates

The function should first ask the users for input two integers for the coordinates first:

```
Enter the coordinate (row, column) of the candy:
```

After receiving the user input, validate the input based on numerical correctness. You can assume the user will only input integers. Ensure that the coordinates are within the valid game board range. If the coordinates input is out of bounds, output the following error message and return zero (0).

```
Coordinates Out of Bound.\n
```

5.5.1.2. Empty Cells

Then, the function should check whether the selected cell is empty or not. If the cell is empty, output the following error message and return zero (0).

```
Empty Cell Selected.\n
```

5.5.1.3. Direction to Swap

After the coordinates are valid, the game should ask for the direction:

```
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):
```

You can assume that the user will input one character only. This input should be validated to ensure it is correct – i.e. Upper case and should be one of the four possibilities (U,D,L,R). If the direction is incorrect, output the following error message and return zero (0).

```
Wrong Direction Input.\n
```

5.5.2. Valid Swapping Check

When attempting to move a candy, the program should validate that the move stays within the boundaries of the game board, For example, the candy at position (0, 0) cannot be swapped to the left, as it is already at the left boundary of the board

If the move is out of bounds, output the following error message and return zero (0).

```
Move Out of Bound.\n
```

In addition, the program should also check the *Target cell* (e.g. if swapping (1,3) to the left, *Target cell* is (1,2)) is empty. In this case, output the following error message and return zero (0).

```
Empty Cell Selected.\n
```

5.5.3. Execute Swap

If the input validations pass, the swap action is performed by calling the swap() function. You should implement the swap() to take the board, the starting coordinate, the target coordinate, and the direction as arguments. The function will then swap the selected candies in the specified direction. For example, if the direction is U (up), the candy at `(row,col)` will be swapped with the candy directly above it at (row-1, col). Similarly, if the direction is L (Left), the candy at `(row, col)` will be swapped with the candy to the left `(row,col-1)`. After the swap, you may use `printGameBoard()` to inspect the result, but remember to remove this call after completing your inspection.

5.5.4. Find and Remove Matches

After the swapping, the `findAndRemoveMatch()` function is called to locate any matches involving the candy at both the target position and the original position, as a match may form in either direction. The function checks the neighboring cells around each of these positions to detect any sequence of three or more identical candies horizontally or vertically.

To facilitate auto-grading and maintain consistency, you should check the matches in the following order.

1. **Order of Checks:** When checking for matches, start with the *Target Position (where swap has just placed a new candy)*. After completing all the checks and removal at the *Target Position*, move to the *Start Position* (the original cell where the candy was swapped from) and perform the same operations.
   For example, swapping (1,2) downwards: *Target Position* is (2,2); and the *Start Position* is (1,2).
2. **Horizontal Matches**: Starting for the given cell, if it is not empty (‘ ’), check to the left (if any) to see if there are three or more consecutive identical candies in the same row. Then check the right direction (if any).

```
      | 0 | 1 | 2 | 3 | 4 | 5 |
     0| # | % | * | * | # | * |
     1| @ | @ | @ | # | # | % |
```

3. **Vertical Matches**: From the given cell, if it is not empty (‘ ’), check upwards (if any) to see if there are three or more consecutive identical candies in the same column. Then check the bottom direction (if any).

```
      3| % | * | @ | # | * | @ |
      4| # | * | @ | % | % | @ |
      5| % | # | @ | * | % | % |
```

If a match is found in the horizontal/vertical direction, output the corresponding message on the screen and indicate the row and column where the match occurred.

```
Horizontal Match found at row <Row Index>!\n
e.g. Horizontal Match found at row 1!
```

```
Vertical Match found at column <Column Index>!\n
e.g. Vertical Match found at column 1!
```

After finding the match, the game should remove the matching candies from the game board. This is the intermediate status of the board after the removal.

```
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Enter the coordinate of the candy you want to swap:5 1
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):R
Vertical Match found at col 2!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
```

If the swap yields no matches, you should restore the game board to the previous status (before the swap), print the message “No Match found!\n”, and return zero (0). Here is the example:

```
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Enter the coordinate of the candy you want to swap:0 0
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):R
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| % | # | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
No Match found!
```

In this function, you should return one (1) to the caller – i.e. AskForSwap() function.

5.6. Game Over Checking

As the game progresses, there will be eventually no more matches in the game board. you should implement `isGameOver()` to check the whole board to see if there is at least one potential match or not.

You can try swapping each candy with its right neighbour (if any) and its bottom neighbour (if any). After the swap, matches may be formed in any four directions (Up, Down, Left, Right). Therefore, you will implement a helper function `isMatching()` to do the checking. This function will do a simple checking to see if there is at least one match-three that can be formed by the given cell in any of the directions. It shall return one (1) if there is at least one match; If there is no match found, return zero (0).

You may need to call it twice to check both the original position and swapped position for any potential matches.

After each check, immediately undo the swap to restore the board to its original state.

If a match is found during any swap, return one (1); If no matches are possible after all check, return zero (0). The main function will check this status and

The following input can be used to check the game over status:

```
5 1
R
1 2
U
4 5
D
2 0
```

```
 D
5 0
R
0 4
R
0 5
D
```

Below shows the last game round:

```
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   |   |   | # |
 1|   |   |   | # | # | % |
 2|   |   |   | @ | % | * |
 3|   | * |   | # | * | @ |
 4|   | * |   |   |   |   |
 5|   | % |   | * | % | @ |
Enter the coordinate (row, column) of the candy:0 5
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):D
Horizontal Match found at row 1!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   |   |   | % |
 1|   |   |   |   |   |   |
 2|   |   |   | @ | % | * |
 3|   | * |   | # | * | @ |
 4|   | * |   |   |   |   |
 5|   | % |   | * | % | @ |
Game Over! No more possible moves.
```

The `main()` function will handle this, print out the error message,

```
Game Over! No more possible moves.\n
```

and will eventually terminate the game loop.

**** End of Part I ****

6. Detailed Program Design Flow – Part II

In the part, we will extend the work of Part I to enhance the program functionality. **Make a copy of your source code of Part I.** You will need to submit another source code for Part II.

6.1. Main Function

In the main function, three more variables are introduced for the next section.

`int stacks[100]`	stores the extra candies, which are used to fill empty spaces after matched candies are removed. Maximum 100 candies can be stored.
`numCandies`	Represents the total number of candies available in the `stacks[]` array for refilling the board.
`current`	Acts as a pointer or counter that keeps track of the current candy being used from `stacks[]` to refill the board during gameplay. The current variable will be initialized to 0 and will be incremented as each candy is used during gameplay.

6.2. Read board and stacks from file

Previously in Part I the game board and stacks are read from hard-coded arrays for easier debugging. Now let's extend the function to file I/O and give us flexibility to replace the game board to a different initial status.

The main() function will first call the `initGameBoardFromFile()` function, which initializes these arrays. The contents of the arrays should be read from a file named **"board.txt"**. Finally the function should return the number of candies in the stack, which will be stored in the `numCandies` variable in the main function.

Below is an example of the expected format for **"board.txt"**:

```
6 6
1 3 2 0 1 0
2 2 0 1 1 3
1 3 3 2 3 0
3 0 2 1 0 2
1 0 2 3 3 2
3 2 1 0 3 3
100
1 2 3 3 2 1 1 0 1 3 1 0 3 0 2 1 1 3 3 1 1 2 0 3 2 2 0 2 0 2 0 0 2 1 2 1 2 2 2 1 1 0 1 3 0 2 0 0 2 0
3 0 0 3 2 3 1 2 2 2 1 1 1 3 3 2 0 2 1 2 1 2 1 1 0 0 2 1 1 1 1 0 2 2 1 0 3 3 3 2 3 0 2 2 1 0 2 2 3 3
```

- **First Line**: Contains two integers representing the game board dimensions – the number of rows r and columns c.
- **Next r Lines**: A grid of r x c integers representing the initial game board. Each integer corresponds to a candy type.
- **Following Line**: Contains a single integer n, which corresponds to numCandies and represents the total number of extra candies in the stacks[].
- **Final Line**: A list of n integers, each representing an extra candy that will be used to fill the board when matches are removed.

Please ensure that the file is read correctly, and handle file operations properly. If an error occurs (e.g., the file cannot be opened), output the following message and terminate the program with an exit code of -1.

```
Failed to open board.txt!\n
```

6.3. Applying Gravity

There will be empty cells after the candy's removal.

```
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
```

You should implement the gravity algorithm in the `applyGravity()` function for simulating "gravity". The main() function will call this function if the swapping is successful – i.e. the return value of `askForSwap`() is non-zero.

In this function, implement the gravity feature to shift candies downwards into empty spaces.

Starting from the **bottom** of the each column, the function should upward to find empty spaces (represented by ASCII 32 ' '). When such empty space is detected, shift any candies above it down by one row . Repeat this process until each column has no empty space remaining below any candies.

At the end of the `applyGravity()`, call printGameBoard() to output the game board.

In this example, the candies at the top of column 2 should fall down to fill the empty spaces below. and the result should look like this:

```
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   | * | # | * |
 1| @ | @ |   | # | # | % |
 2| # | % |   | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | * | % | % | @ |
 5| % | # | % | * | % | % |
```

At this point, you do not have to do the match checking.

6.3.1. Fill Empty Spaces

After candies are removed from the board, empty spaces will be left behind, which need to be refilled after applying the gravity. The main function will call `fillEmpty()` to fill these spaces using the candies stored in the `stacks[]` array after applying the gravity. You can track the current position in the array using the current variable, which indicates where the next candy should be retrieved from.

As you retrieve candies from the array, fill the empty spaces from <u>bottom-to-top, left-to --right</u> manner to simulate the real candy crush.

At last, call `printGameBoard()` to show the board status.

Here is the status after the candy removal, just before the gravity application and candy refill.

```
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * |   | # | * | @ |
 4| # | * |   | % | % | @ |
 5| % | # |   | * | % | % |
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % |   | * | # | * |
 1| @ | @ |   | # | # | % |
 2| # | % |   | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | * | % | % | @ |
 5| % | # | % | * | % | % |
```

After calling fillEmpty() in the main() function, you should see the following:

```
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | % | * | # | * |
 1| @ | @ | @ | # | # | % |
 2| # | % | # | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | * | % | % | @ |
 5| % | # | % | * | % | % |
```

Once the original candies have fallen to the bottom (highlighted in yellow above), any remaining empty spaces are refilled sequentially using candies from the `stacks`[] array. The current variable in the parameter keeps track of the position in stacks[] from where the next candy should be drawn. Each time after the candies is drawn, current is incremented to point to next candies.

Here is the example of how current variable works with the `stacks[]`. After the beginning of the game, the current variable is set to zero (0) to indicate the candy drawn next is the first item in the stack.

```
↓ current = 0
1 2 3 3 2 1 1 0
```

After one candy is drawn for refill, the `current` variable is incremented to one (1) such that next candy drawn is the second item in the `stacks[]`.

```
  ↓ current = 1
1 2 3 3 2 1 1 0
```

In the game, this is the array that helps you to convert the number to the candy representation.

```
char candies[] = {'*', '#', '@', '%'};
```

With this array, you can observe that the empty cells are filled with “#”, “@” and “%” from bottom to top, which is 1,2,3 in the stacks array respectively.

The process will continue until all the candies in the stack are exhausted. When filling the candy, if the number of candies remaining in the stacks[] is less than the number of empty spaces on the board, the game will be unable to fill all spaces. In this case, the game should display an error message and terminate the program with an exit code of -1.

```
No more candies available.
```

This check will naturally handle both cases: if stacks[] is exhausted during normal game play, or if stack[] is empty from the very beginning, as both conditions would trigger the same error and exit the program at this point.

If multiple columns contain empty spaces, fill these spaces in a bottom-to-top order within each column, moving from the leftmost column to the rightmost column. Here is the example of the case of filling in multiple column spaces. The number in red italic and brackets indicates the filling sequence.

```
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0|(2)|(4)|(6)| * | # | * |
 1|(1)|(3)|(5)| # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
```

6.3.2. Cascade Matching

When the candies are refilled, new matches may form on the board. To mimic real gameplay, the game should continuously check for new matches as the board refills, repeating the process until the board is stable with no further matches.

The main function will call `cascade`() function right after the matching process. Within the function, you should call both the `applyGravity`() and `fillEmpty`() for the cascade process. As a result, any previous calls to these functions in the main function should be removed, as they are now called within cascade().

In the cascade function, you are required to check if there are any matches for each cell. For this, you will call `findAndRemoveMatch`() function (developed in part I) to find all matches on each non-empty cell. You can make use of the return value of `findAndRemoveMatch`() to determine whether a match is found.

In the cascade function, repeat the process until no further matches can be found in the board. In each checking round:

1. Apply the gravity.
2. Fill in all the empty cells on the board and get the updated value of `current`
3. Print the Gameboard to show the current status
4. Remove all matches found on the board. Starting from the top-left corner (0,0), check each cell for matches in the following order:
    a. *Row by Row, Column by Column*: For each cell, proceed across each row from left to right, then move down to the next row.
    b. *Check Directions*: For each cell, check for matches in the order of Left, Right, Top, and Bottom

   c. *Clear Matches*: If a match is found, clear the matching candies immediately before proceeding to the next cell.

5. If there is any removal, output the following at the end of checking:

```
Cascade Matches found!\n
```

6. After the cascade checking completes with no further matches, return the updated value of `current` to the main() function.

Repeat these steps until no matches are left, ensuring that all cascading actions are completed before returning to the main function.

Below is the sample of a cascade matching, the matches found by the cascade function are in red font.

```
=====
New Round:
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | @ | * | # | * |
 1| @ | @ | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Enter the coordinate (row, column) of the candy:1 2
Enter the direction to swap (U for Up, D for Down, L for Left, R
for Right):U
Horizontal Match found at row 1!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | % | * | * | # | * |
 1|   |   |   | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0|   |   |   | * | # | * |
 1| # | % | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | @ | % | * | # | * |
 1| # | % | * | # | # | % |
 2| # | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Vertical Match found at column 0!
Cascade Matches found!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0|   | @ | % | * | # | * |
 1|   | % | * | # | # | % |
 2|   | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | @ | % | * | # | * |
 1| @ | % | * | # | # | % |
 2| % | % | % | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
Horizontal Match found at row 2!
Cascade Matches found!
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0|   |   |   | * | # | * |
 1| # | @ | % | # | # | % |
 2| @ | % | * | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
=====
  | 0 | 1 | 2 | 3 | 4 | 5 |
 0| # | * | # | * | # | * |
 1| # | @ | % | # | # | % |
 2| @ | % | * | @ | % | * |
 3| % | * | @ | # | * | @ |
 4| # | * | @ | % | % | @ |
 5| % | @ | # | * | % | % |
```

7. Academic Honesty and Declaration Statement

Attention is drawn to University policy and regulations on honesty in academic work, and to the disciplinary guidelines and procedures applicable to breaches of such policy and regulations. Details may be found at https://www.erg.cuhk.edu.hk/erg/AcademicHonesty .

You are required to fill in the following declaration statement as the comment at the beginning of the main.c source code with your information. Assignments without the properly signed declaration will not be graded.

Tools such as software similarity measurement, AI-tool fingerprint detection, etc., might be used to inspect all submissions.

```
/**
* ENGG1110 24R1 Problem Solving by Programming
 *
 * Course Project
 *
 * I declare that the project here submitted is original
 * except for source material explicitly acknowledged,
 * and that the same or closely related material has not been
 * previously submitted for another course.
 * I also acknowledge that I am aware of University policy a nd
 * regulations on honesty in academic work, and of the disciplinary
 * guidelines and procedures applicable to breaches of such
 * policy and regulations, as contained in the website.
 *
 * University Guideline on Academic Honesty:
 *   https://www.cuhk.edu.hk/policy/academichonesty/
 *
 * Student Name  :
 * Student ID    :
 * Class/Section :
 * Date          :
 */
```

8. Grading Platform and Submission

We will grade your work in Gradescope autograder. Please note that while passing the test cases is necessary, it is not sufficient on its own for full marks. Additional marks will be allocated based on the completeness and adherence to all requirements specified in the project documentation, including code organization, functionality, and adherence to the design specifications.

Please follow the steps below to submit your work by the deadline.

1. Rename the modified main.c for part I to **main_part1.c**
2. Rename the source code for part II to **main_part2.c**
3. Submit your .c file to the Gradescope.

Please only submit the source file (.c). You may submit many times before the due date, but the latest attempt will be graded.

9. Late Submission

The late submission penalty is as follows:

- Within 3 days (72 hours): 10% per 24 hours
- 3-7 days: 50% penalty
- More than 7 days: 100% penalty (Your work will not be graded)

NOTE: Submitting the project is crucial in PASSING the course.