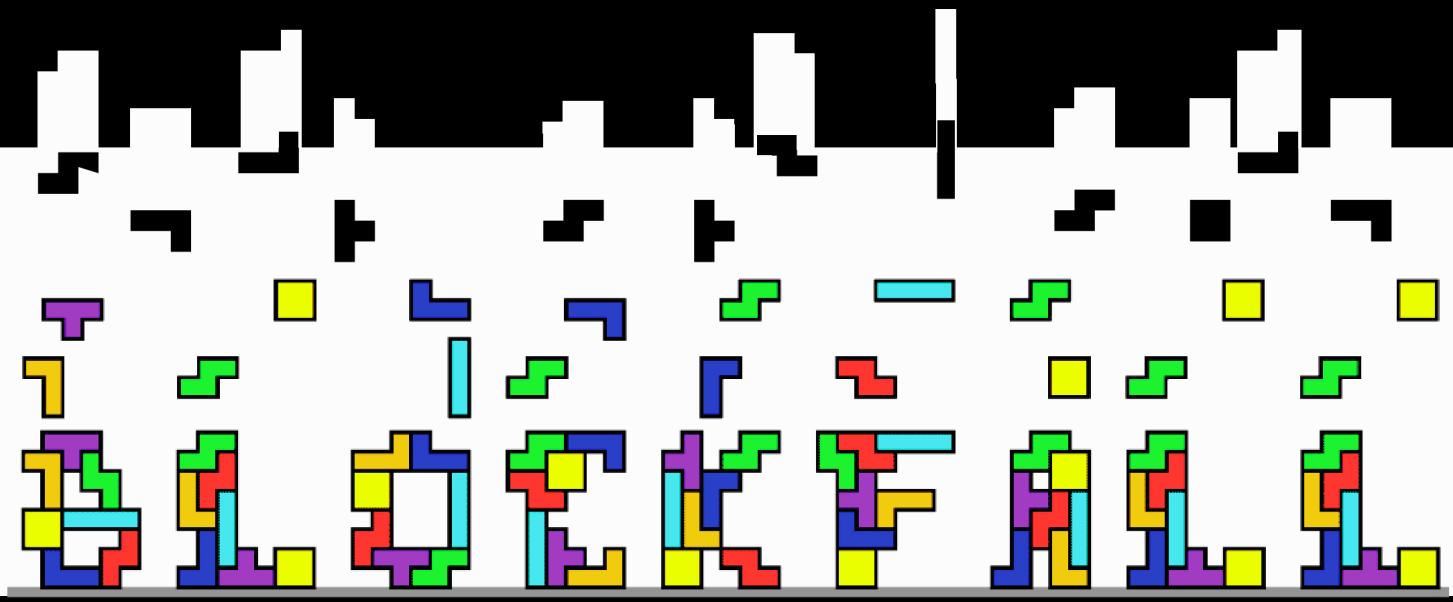
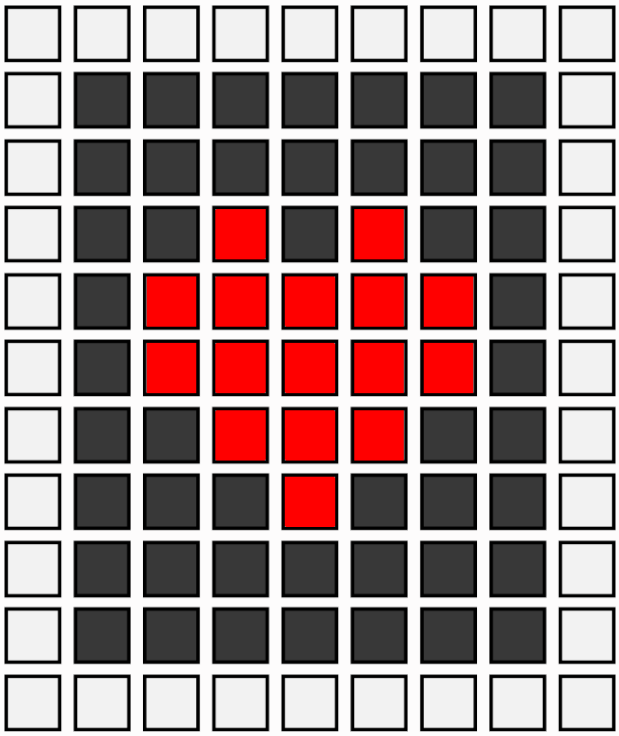


PROGRAMMMING ASSIGNMENT 2



Hacettepe University - Computer Engineering
BBM203 Software Practicum I - Fall 2023

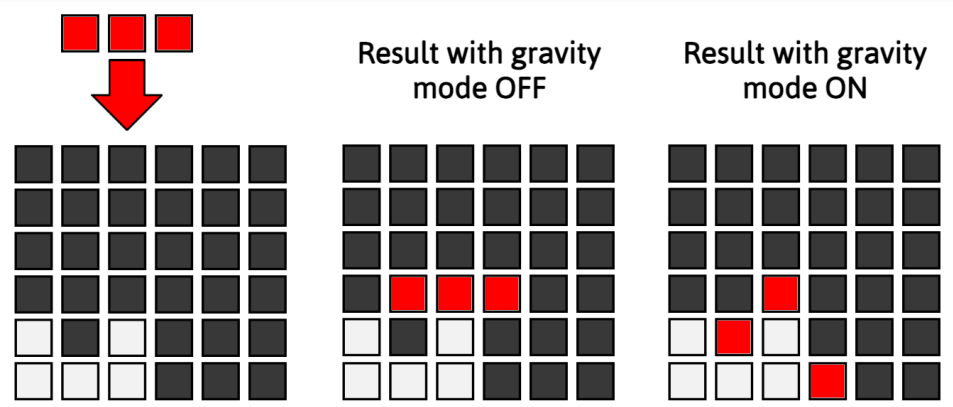
Topics: Linked Lists, Dynamic Memory Allocation, Matrices, File I/O

Course Instructors: Assoc. Prof. Dr. Adnan ÖZSOY, Asst. Prof. Dr. Engin DEMİR, Assoc. Prof. Dr. Hacer YALIM KELEŞ
TAs: Alperen ÇAKIN*, Ardan YILMAZ, **Dr. Selma DİLEK***
Programming Language: C++11 - **You MUST USE this starter code**
Due Date: **Friday, 24/11/2023 (23:59:59)**

BlockFall

The Next Viral Block-Dropping Row-Popping Game for the Masses

Welcome to the intriguing challenge of **BlockFall**, where your understanding of data structures and dynamic memory allocation will be put to the test! This task invites you to delve into a game environment filled with falling blocks, requiring strategic maneuvering. As promising programmers, your mission is to create order within the continuous flow of descending blocks, using the powerful tools of linked list data structures and dynamic memory allocation in C++. Representing **HUBBM** and **HUAIN**, prestigious computer engineering programs, you are entrusted with the development of **BlockFall** destined to be the latest sensation in block-matching games. Hone your coding prowess, confront the forthcoming obstacles, and embark on this instructive adventure.



BlockFall is a console-based tile-matching video game, drawing inspiration from the classic Tetris. The game unfolds within a rectangular playfield, termed the "grid", composed of "cells" with customizable dimensions. Players are tasked with strategically positioning the cascading "blocks" from the screen's upper left, aiming to create complete horizontal lines devoid of spaces. Players can shift "blocks" horizontally or rotate them 90° either clockwise (right) or counterclockwise (left) before they are dropped. Completing a "row" causes it to vanish, prompting any overhead blocks to fall. **BlockFall** includes a range of features such as a scoring mechanism, power-ups, a leaderboard, and two gameplay modes. These encompass a default mode, mirroring traditional Tetris, and a gravity mode where "cells" within the "blocks" can break apart and tumble down, simplifying the scoring process. This assignment provides a comprehensive exercise in C++ programming, covering file I/O, class design, dynamic memory allocation, and a number of data structures, specifically focusing on multilevel linked lists. It is expected that by the end of this assignment, students will have a better understanding of these concepts and hands-on experience in their application.

The success of the BlockFall game lies in your hands - are you up to the challenge?

1 Reading the Input Data and Initializing the Game

In this section, we outline game initialization via provided inputs. Pay close attention to dynamic memory allocation requirements, as they are crucial for full credit.

1.1 Input Files and Command Line Arguments

An input file containing details about the game **grid**, structured as a 2D $rows \times cols$ matrix, will be supplied in `DAT` format via the *first command line argument*. Your task is to parse this file within your program to set up the game `grid` within the `BlockFall` class. The content of a sample input file given as `grid.dat` with 10 rows and 10 columns is illustrated on the right.

```
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
```

The input file, in `DAT` format and supplied as the *second command line argument*, contains the game's **blocks** represented as 2D $height \times width$ matrices within square brackets [ ]. Your program should use this file to generate a linked list of game blocks in the `BlockFall` class. An excerpt from an example file, `blocks.dat`, is displayed on the right. Blocks are placed in order they are meant to appear in-game, top to bottom, in their initial, unrotated state. **The last shape represents the power-up, which shouldn't be added to the block list, but kept in the `power_up` member variable of the class `BlockFall`.** It is crucial to recognize that the $height$ and $width$ of individual blocks may vary, necessitating dynamic memory allocation for blocks and their rotations. A representation of the blocks from this sample input is provided below:

```
[101
 111]

[010
 111
 010]

[01
 10]

[101
 111
 101]
```

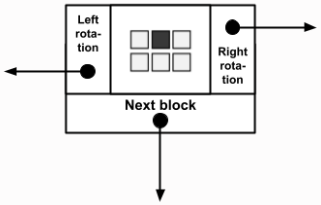
The input file containing the game's **commands**, represented as strings, will be supplied in `DAT` format as the *third command line argument*. Your program is tasked with interpreting this file's contents to facilitate gameplay, an operation to be implemented within the `GameController` class. An excerpt from a typical input file, named `commands.dat`, is displayed to the right for reference.

```
ROTATE_LEFT
ROTATE_RIGHT
MOVE_RIGHT
MOVE_LEFT
GRAVITY_SWITCH
DROP
```

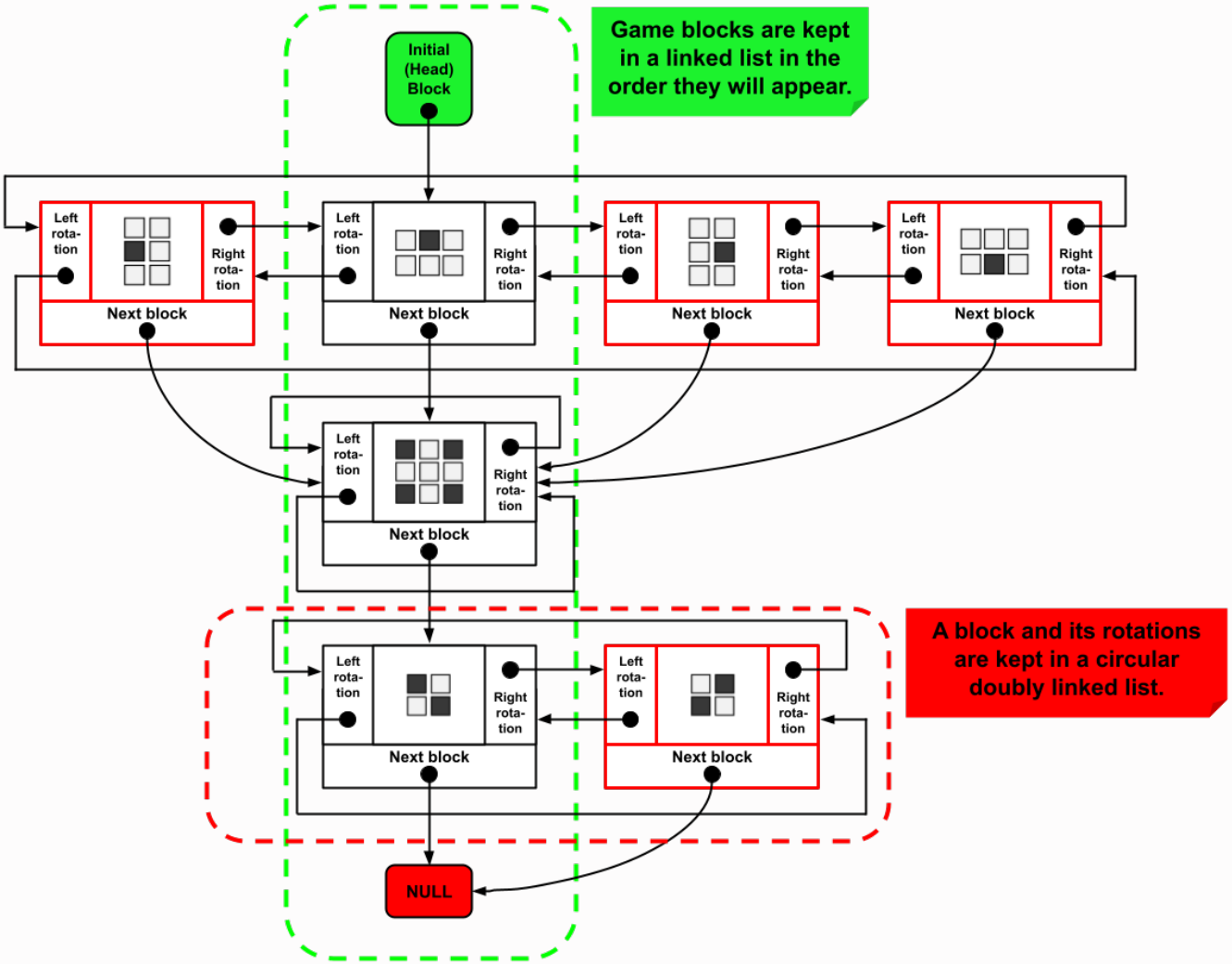
The *fourth command line argument* sets the initial state of the **gravity mode** to either **ON** or **OFF**, accepting values of `GRAVITY_ON` or `GRAVITY_OFF`. The *fifth command line argument* designates the filename of a text file where the **leaderboard** data is stored (further details will follow in the subsequent sections of the assignment instructions). Lastly, the *sixth command line argument* determines the **current player's name** for leaderboard identification purposes.

1.2 Initializing the Block List

In the game, **blocks will be represented as nodes in a multilevel linked list**. Upon reading the sequential game blocks from the corresponding input file, your first step is to identify the upcoming block's *shape*, instantiate it, and then integrate it into the game's linked list of blocks. This integration is achieved by accurately assigning the `next_block` pointer of the list's preceding block. Furthermore, you're required to compute all possible rotations of the block, cataloging them within a circular doubly linked list. Crucially, the *shape* of each block should be represented as a dynamically expandable 2D matrix due to the indeterminate nature of the shape's dimensions prior to initialization.



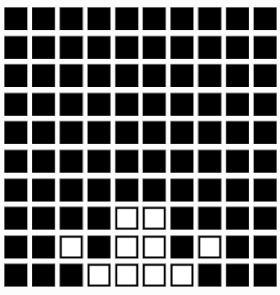
Each block's `right_rotation` pointer should allow sequential access to its clockwise rotations, looping back in a circular fashion. Analogously, the `left_rotation` pointer should enable consecutive access to its counter-clockwise rotations, also in a circular pattern. It is imperative to note that both the original block and each of its rotations must consistently point to the default (unrotated) state of the subsequent incoming block via the `next_block` pointer variable.



1.3 Initializing the Grid

The game `grid` should be managed as a dynamically sized 2D integer matrix, where **zeroes** (0's) represent empty cells and **ones** (1's) denote occupied cells. The initial grid may be pre-populated. You may assume that the grid's minimum dimensions will equate to those of the largest block in play. Within the input file, digits are divided by a single space, and each row concludes with a newline character.

```
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 1 0 0 0 0
0 0 1 0 1 1 0 1 0 0
0 0 0 1 1 1 1 0 0 0
```



2 Key Functionalities to Be Implemented

In this section, we cover gameplay rules and essential functionalities for implementation. Close adherence to dynamic memory allocation requirements is crucial for full credit.

2.1 Reading and Processing Commands

BlockFall features seven commands that will be listed in the corresponding input file, each on a new line, without spaces. Commands must be interpreted dynamically and immediately as they are encountered, not stored in memory, mimicking real-time gameplay conditions. The commands are as follows:

- `PRINT_GRID`: Print the current state of the grid.
- `ROTATE_RIGHT`: Rotate the active block to the right (clockwise), if possible.
- `ROTATE_LEFT`: Rotate the active block to the left (counterclockwise), if possible.
- `MOVE_RIGHT`: Move the active block one space to the right, if possible.
- `MOVE_LEFT`: Move the active block one space to the left, if possible.
- `DROP`: Drop the active block, handling power-ups, clearing full rows, and gaining points.
- `GRAVITY_SWITCH`: Toggle gravity mode.

Unknown commands are handled with printing an error message, but processing continues with the next command. See an example on the right.

```
Unknown command: GIMME_POINTS
```

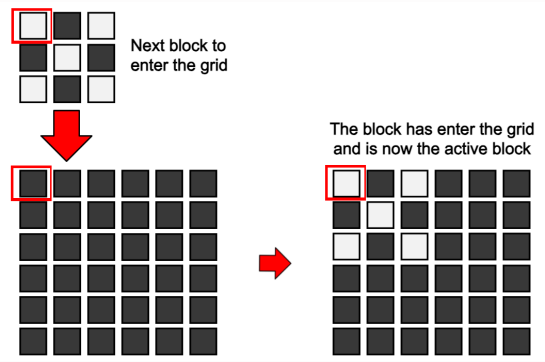
2.2 Game Rules

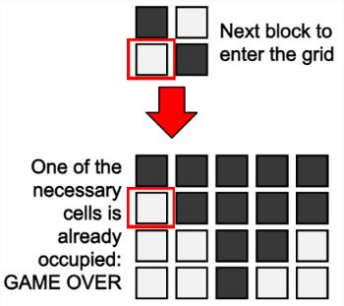
While this game draws inspiration from the classic Tetris, it diverges delicately in its ruleset. As such, it is crucial to meticulously review the game rules and execute the assignment tasks, paying close attention to each key requirement and detail outlined.

2.2.1 Block Movements and Collision Detection

An incoming active block initiates entry into the game `grid` from the top-left corner, meaning the upper leftmost "cell" of the block aligns with the top-left "cell" of the `grid`.

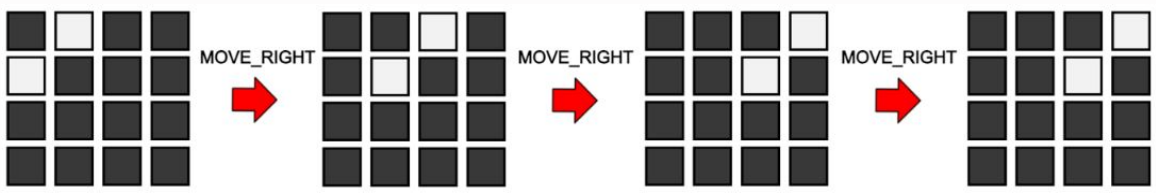
The game should continuously monitor for any collisions between the active block and occupied spaces on the grid. If a collision occurs during a movement or rotation attempt, the respective action is nullified (not performed). Moreover, if a collision happens as a new block makes its entry, it signifies the end of the game.



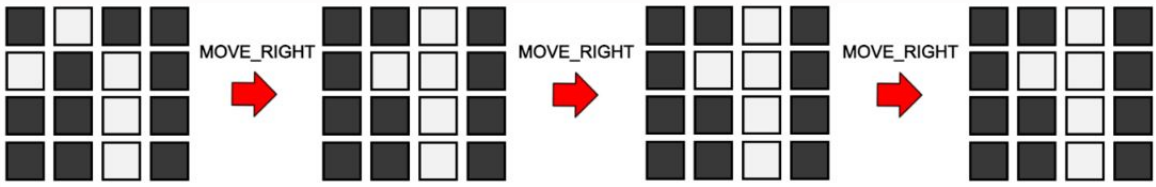


Collisions when trying to enter the grid: If any cell necessary to accommodate an incoming block on the game `grid` is already filled by a previously settled block, the new block will be prevented from entering, signaling the end of the game.

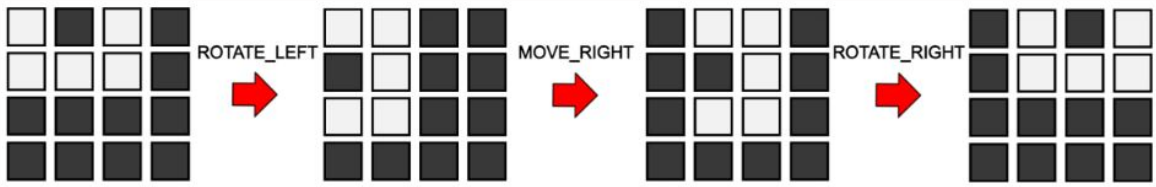
Collisions when trying to move left or right on the grid: Blocks are permitted to shift left or right by one cell, provided there are no obstructions. Upon reaching the grid's edge, further lateral movement in that direction becomes impossible, although reversal is an option. Despite any issued command, the block should remain stationary under such circumstances.



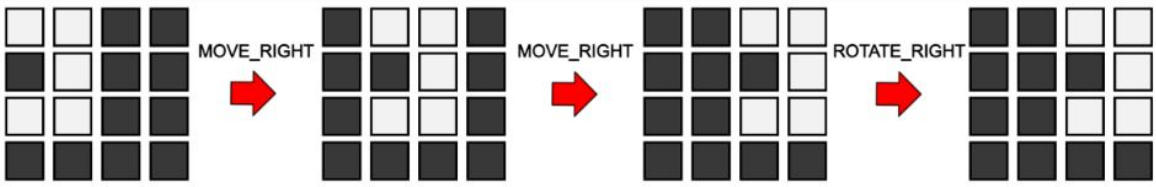
Occasionally, a block's movement may be prevented before it reaches the grid's edge. If an occupied cell obstructs its path, the block must remain stationary, regardless of movement commands received.



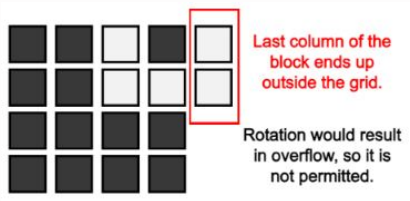
Collisions when trying to rotate left or right: When rotating a block, it's crucial to maintain the position of the block's top leftmost cell on the `grid`. This means that if you have a block of size $n \times m$, and its top leftmost cell is positioned at the $grid[i][j]$, then after rotation, which will produce a block of size $m \times n$, the top leftmost cell of the newly rotated block must also be located at $grid[i][j]$.



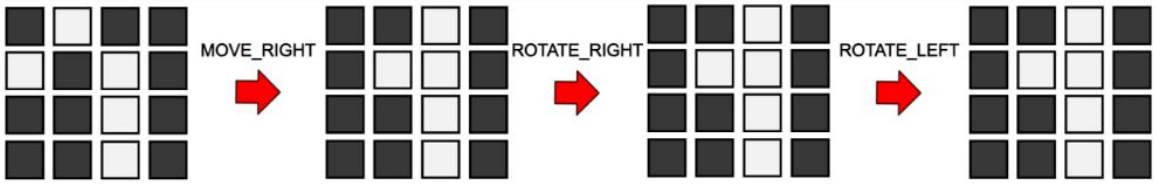
This means that in some cases rotation will not be possible. The first case occurs when the rotation of the block cannot fit inside the `grid`:



The rotation in the example above is prohibited because it would result in the block overflowing beyond the grid's boundary. This restriction adheres to the rule that mandates the block's top leftmost cell to retain its position on the grid during rotation.



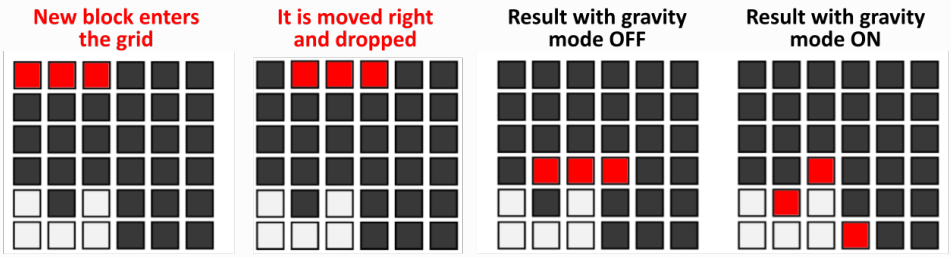
The second case occurs when an already occupied cell is preventing the rotation, in which case, the block should also not be allowed to rotate.



Horizontal movements and rotations are permissible only at the top of the `grid` on the active block, prior to its descent. Once the block is dropped, the block is immovable and unrotatable, continuing its fall until it lands on the grid's base or atop previous blocks, where gravity mode significantly influences gameplay.

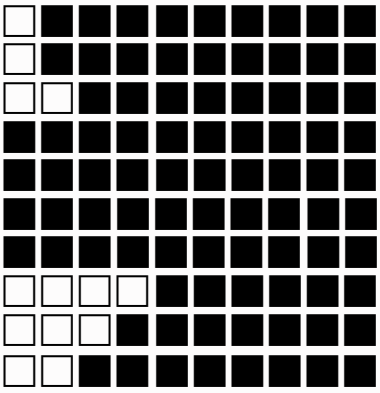
Dropping a block: With `GRAVITY_OFF`, the block will drop intact as far as possible (without breaking to pieces), halting upon encountering any occupied cell, preserving its structural integrity. The presence of even a single occupied cell is sufficient to halt the block's downward trajectory.

Conversely, when the gravity mode is set to `GRAVITY_ON`, the block interacts differently with the `grid` and other blocks. In this mode, each individual cell within the block obeys the law of gravity independently. If the block collides with previously settled blocks or the bottom of the `grid`, any unobstructed individual cells within the block will continue to fall until they are stopped by another occupied cell or reach the bottom. This means that the original block may break apart during the fall, allowing spaces to be filled more efficiently, potentially leading to more dynamic gameplay strategies. You need to consider how the block's structure will be affected during the fall and plan the cells' placement accordingly.



Printing the Grid: The `PRINT_GRID` command first prints the player's current score, followed by the all-time high score. If the all-time high score has not been set yet (indicating a first-time play) or if the player's current score exceeds the existing record at any point in the game, then the player's score should be recorded and displayed as the new all-time high. Subsequently, the grid itself must be printed row-by-row, taking into account the overlay of the presently active block and any other occupied cells. The currently active block must be superimposed (shown) on the grid, considering its current rotation and position on the X-axis (determined as the result of any previous `MOVE_LEFT` or `MOVE_RIGHT` commands). An example output of the `PRINT_GRID` command is illustrated on the right.

```
Score: 70
High Score: 1837
```

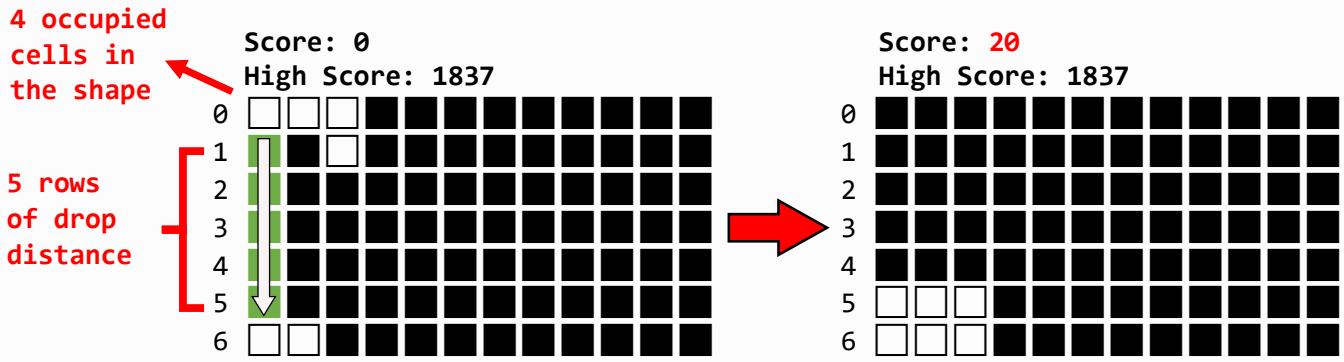


2.2.2 Clearance and Scoring

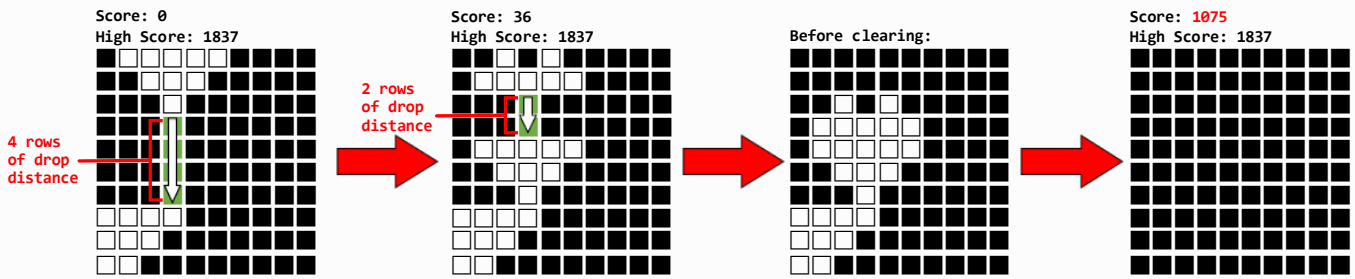
When a block is dropped, the game should first check for and clear any power-ups, and then, it should check for and clear fully filled rows, awarding points based on the performed action(s). It is not necessary to check for power-ups after clearing the rows. The scoring guidelines are detailed in the table below.

Action	Score Impact
Block Drop	The score increases by an amount equal to the number of occupied cells in the block multiplied by the distance it falls.
Power-up Detected	Upon detection, the grid is instantly cleared, awarding 1000 bonus points plus an additional point for each occupied cell on the grid that is being cleared.
Clearing Rows	The score increases by an amount equal to the number of columns in the grid multiplied by the number of rows cleared.

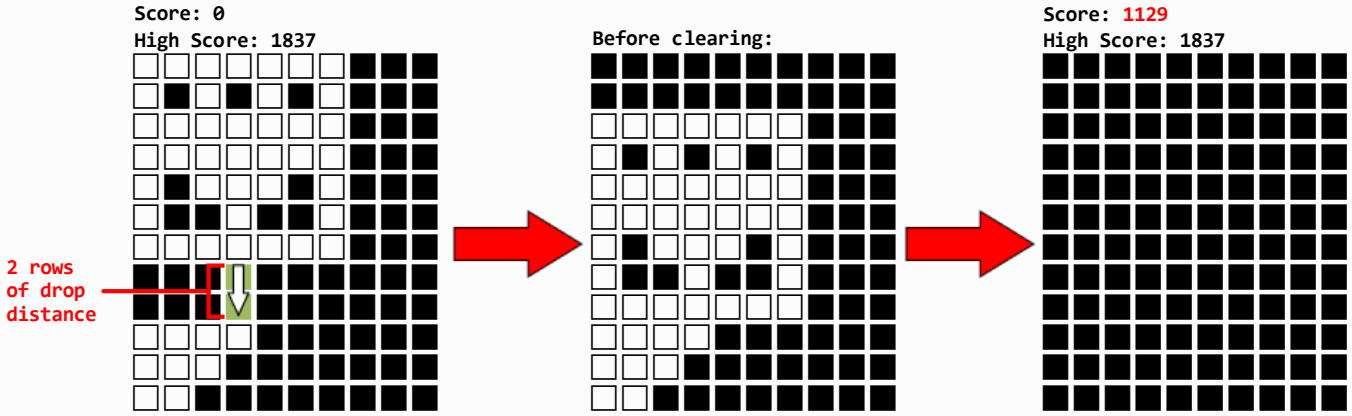
The figures below illustrate various scoring scenarios. The first figure showcases the scoring for a block drop: a block comprising four occupied cells descends five rows, resulting in a total addition of 20 points to the score.



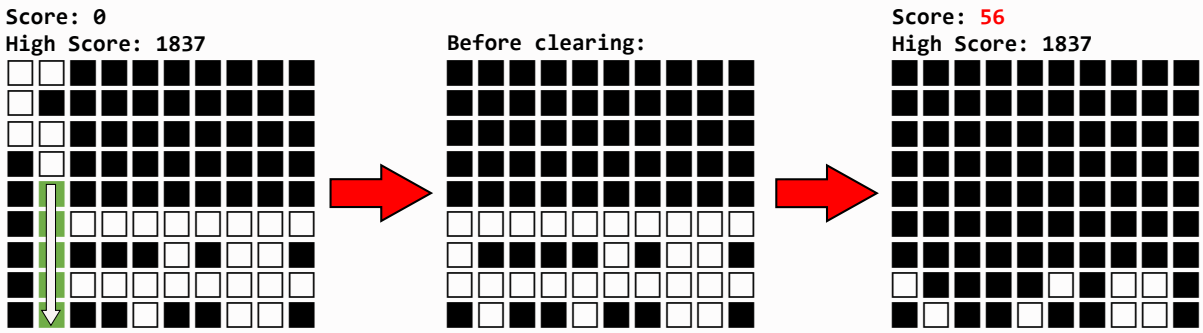
In **BlockFall**, power-ups allow players to boost their scores more quickly. One notable power-up shape is the game's iconic 5×5 **heart** shape, **though other shapes can serve as power-ups too**. **The specific power-up shape for each game session is determined by the last block shape listed in the blocks `DAT` input file given as the *second command line argument***. If players manage to form the given shape on the `grid` using fallen blocks, they earn a bonus of 1000 points. Furthermore, the `grid` is immediately cleared, and players get an additional point for each occupied cell removed, adding to their bonus. For the power-up to activate, the shape on the `grid` must be distinctly recognizable. This means, while the shape can touch other occupied cells, to trigger the power-up, every cell of the shape must match its status on the `grid`: if a cell is empty in the shape, it must also be empty on the grid, and vice versa.



Another example of power-up shape being recognized on the `grid` is given below. In this example, the power-up is again the game's iconic **heart** shape.



After checking for power-ups and finding none, the game should then scan the `grid` for any completed rows. When players complete one or more rows on the `grid`, these rows are cleared, allowing the remaining unfilled rows to shift downwards. Players are then awarded bonus points, specifically, one additional point per cleared row, multiplied by the total number of columns in the `grid`. After clearing the completed rows, the game won't recheck for power-ups. You can assume this scenario won't arise. **Your program must also print `Before clearing:` message and the current `grid` state below it before clearing the rows to `STDOUT`.**



2.2.3 Leaderboard

The game features a leaderboard that holds up to **10** high score entries, with each entry comprising a **score**, **timestamp** (acquired by *time(nullptr)* upon entry creation), and **player name**. Prior to each game session, the system should try to load the leaderboard from an existing file given as the *fifth command line argument*, if available. However, this file may not be present during an initial run, indicating an absence of high scores. Once a game session concludes, regardless of the reason for which the game has ended, the leaderboard needs to be refreshed to reflect the latest scores and then saved back to the same leaderboard file. This leaderboard retains a maximum of **10** top scores across all sessions and must be consistently sorted in descending order based on scores. It's important to note that there may be instances where fewer than ten high scores are stored. In such scenarios, avoid attempts to parse ten entries. Instead, adopt a flexible approach to prevent memory errors.

The text file contents should be structured as follows:

```
<score> <timestamp> <player_name>
```

As an example:

```
40000 1697910655 BlockBuster
1200 1697910655 StackOverthrower
...
```

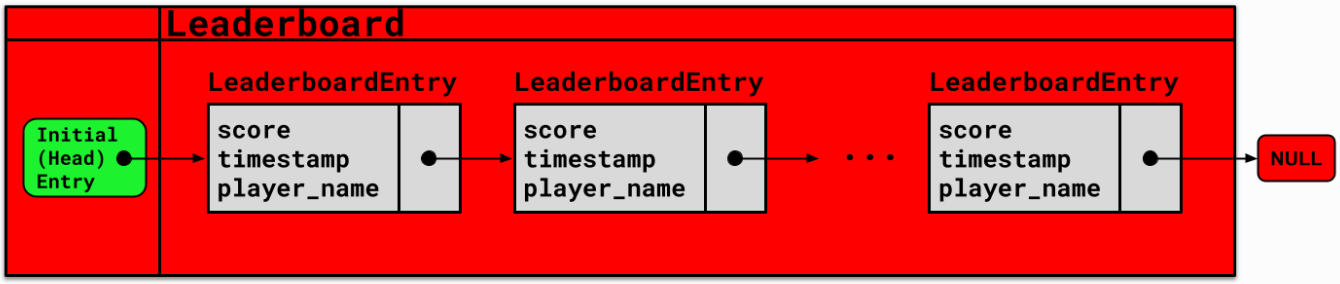
Moreover, the leaderboard should be printed to STDOUT in the following format:

```
Leaderboard
-----------
<#order>. <player_name> <score> <timestamp formatted as %H:%M:%S/%d.%m.%Y>
```

As an example:

```
Leaderboard
-----------
1. BlockBuster 40000 20:50:55/21.10.2023
2. StackOverthrower 1200 20:50:55/21.10.2023
...
```

The `Leaderboard` class has a pointer to the first (top) `LeaderboardEntry` named `head_leaderboard_entry`. It will be `NULL` if there are no highscores yet. Leaderboard must be stored as a linked list of `LeaderboardEntry` instances, such that each entry points to the next high score.



The leaderboard is designed to retain a maximum of ten all-time high scores, necessitating a dynamic implementation that can efficiently manage insertions and deletions while maintaining score order. As such, you are tasked with developing functionalities that correctly integrate new high scores into the leaderboard. This involves dynamically allocating memory for new entries and responsibly deallocating memory associated with those that are displaced from the top ten rankings. It's crucial to ensure that these operations preserve the leaderboard's integrity and its real-time reflection of player achievements.

2.2.4 Game Over

The game progresses until it meets one or more of the following termination conditions:

- The grid can no longer accommodate incoming blocks, signaling the end of the game,
- The input file has been exhausted of commands,
- No additional blocks remain available for gameplay.

Gameplay concludes under circumstances where either a new block is precluded from entering the grid due to potential collisions with existing blocks, or when the supply of blocks or commands depletes. Upon the game's termination, the final grid layout and score are revealed, and the leaderboard is updated to include any newly achieved high scores. Depending on the way the game has ended, three different output formats are expected as illustrated below.

```
GAME OVER!
Next block that couldn't fit:
□
□

Final grid and score:

Score: 10
High Score: 40000
■■■■■■■■■■
□□■■■■■■■■
□□■■■■■■■■
□■□■■■■■■■

Leaderboard
-----------
1. BlockBuster 40000 20:50:55/21.10.2023
2. StackOverthrower 1200 20:50:55/21.10.2023
3. DebugDropper 10 22:00:50/21.10.2023
```

```
GAME FINISHED!
No more commands.
Final grid and score:

Score: 6
High Score: 40000
■■■■■■■■■■
■■■■■■■■■■
■■■■■■■■■■
□■□■■■■■■■

Leaderboard
-----------
1. BlockBuster 40000 20:50:55/21.10.2023
2. StackOverthrower 1200 20:50:55/21.10.2023
3. DebugDropper 10 22:00:50/21.10.2023
4. LineLiberator 6 22:10:00/21.10.2023
```

```
YOU WIN!
No more blocks.
Final grid and score:

Score: 10
High Score: 40000
■■■■■■■■■■
□□■■■■■■■■
□□■■■■■■■■
□■□■■■■■■■

Leaderboard
-----------
1. BlockBuster 40000 20:50:55/21.10.2023
2. StackOverthrower 1200 20:50:55/21.10.2023
3. DebugDropper 10 22:00:50/21.10.2023
4. NullNinja 10 22:20:50/21.10.2023
5. LineLiberator 6 22:10:00/21.10.2023
```

2.3 Assignment Implementation Tasks and Requirements

In this section, we outline the classes and functions you are required to implement. As aspiring engineers, it's crucial to base your code on the provided starter code that offers a predetermined class structure. This ensures code clarity, proper encapsulation, and facilitates unit testing. It's imperative that you do not alter the names or signatures of functions provided in the template files. Likewise, refrain from renaming or changing the types of the specified member variables. Other than that, you're free to introduce any additional functions or variables as needed.

2.3.1 `Block` Class

This class represents a block in the game.

- **Operators:** $==$ **and** $!=$: Overload these operators to compare two blocks based on their shapes.

2.3.2 `BlockFall` Class

This class represents the main game logic and structure.

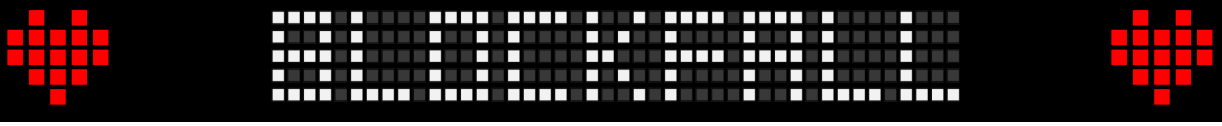
- **Constructor**:

  ```
  BlockFall(string grid_file_name, string blocks_file_name, bool
      ↪ gravity_mode_on, const string &leaderboard_file_name, const string
      ↪ &player_name)
  ```

  - Initialize the game with the given parameters.
- **Function**:

  ```
  read_blocks(const string &input_file)
  ```

  - Read the blocks from the input file and initialize member variables such as `initial_block` and `active_rotation`. Make sure to also generate rotations for each block and properly set the linked list structure of the game blocks. You should also initialize the `power_up` member variable as the last block from the input file (do not add it to the linked list!)



- **Function**:
  `initialize_grid(const string &input_file)`
  - Initialize the game grid from the given input file.
- **Destructor**:
  `~BlockFall()`
  - Clean up any dynamically allocated memory used for storing game blocks. Do not forget to delete the rotations.

2.3.3 `GameController` Class

- **Function**:
  `play(BlockFall& game, const string& commands_file)`
  - Implement the gameplay using commands from the provided input file.

2.3.4 `LeaderboardEntry` Class

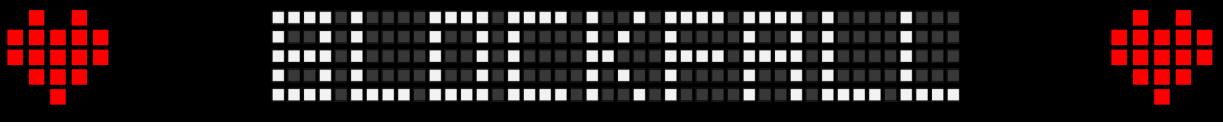
This class represents a single entry in the game's leaderboard.

- **Constructor**:
  `LeaderboardEntry(unsigned long score, time_t lastPlayed, const string &playerName)`
  - Initialize a leaderboard entry with the given parameters.

2.3.5 `Leaderboard` Class

This class manages the game's leaderboard.

- **Function**:
  `insert_new_entry(LeaderboardEntry * new_entry)`
  - Insert a new leaderboard entry into the list with the head pointer `head_leaderboard_entry`, ensuring the scores are in descending order. Maintain a maximum of 10 entries.
- **Function**:
  `read_from_file(const string& filename)`
  - Read the leaderboard from a specified file.
- **Function**:
  `write_to_file(const string& filename)`
  - Write the current leaderboard to a specified file conforming to the given format.
- **Function**:
  `print_leaderboard()`
  - Print the current leaderboard status to standard output.
- **Destructor**:
  `~Leaderboard()`
  - Clean up any dynamically allocated memory used for storing leaderboard entries.



Must-Use Starter Codes

You MUST use **this starter (template) code**. All headers and classes should be placed directly inside your **zip** archive.

Grading Policy

- No memory leaks and errors: 10%
  - No memory leaks: 5%
  - No memory errors: 5%
- Implementation of the game: 80%
  - Proper game grid initialization: 5%
  - Correct implementation of the multilevel linked list structure for game blocks and their rotations, and related operations: 20%
  - Correct implementation of the horizontal block movement 5%
  - Correct implementation of block rotation: 10%
  - Correct implementation of block drop: 7.5%
  - Correct implementation of the gravity modes and gravity mode change: 7.5%
  - Correct implementation of the scoring mechanism: 5%
  - Correct implementation of the leaderboard linked list and related operations: 15%
  - Proper game termination: 5%
- Output tests: 10%

Important Notes

- Do not miss the deadline: **Friday, 24.11.2023 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (https://piazza.com/hacettepe.edu.tr/fall2023/bbm203), and you are supposed to be aware of everything discussed on Piazza.
- You must test your code via **Tur^{3}Bo Grader** https://test-grader.cs.hacettepe.edu.tr/ (**does not count as submission!**).
- You must submit your work via https://submit.cs.hacettepe.edu.tr/ with the file hierarchy given below:
  - **b<studentID>.zip**
    - Block.h <FILE>
    - BlockFall.h <FILE>
    - BlockFall.cpp <FILE>
    - GameController.h <FILE>
    - GameController.cpp <FILE>
    - LeaderboardEntry.h <FILE>
    - LeaderboardEntry.cpp <FILE>
    - Leaderboard.h <FILE>
    - Leaderboard.cpp <FILE>
- **You MUST use this starter code**. All classes should be placed directly in your **zip** archive.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).

Run Configuration

Here is an example of how your code will be compiled (note that instead of `main.cpp` we will use our test files):

```
$ g++ -std=c++11  main.cpp Block.h BlockFall.h BlockFall.cpp GameController.h
↪   GameController.cpp LeaderboardEntry.h LeaderboardEntry.cpp Leaderboard.h
↪   Leaderboard.cpp -o blockfall
```

Or, you can use the provided `Makefile` or `CMakeLists.txt` within the sample input to compile your code:

```
$ make
```

or

```
$ mkdir blockfall_build
$ cmake -S . -B blockfall_build/
$ make -C blockfall_build/
```

After compilation, you can run the program as follows:

```
$ ./blockfall grid.dat blocks.dat commands.dat GRAVITY_ON leaderboard.txt
↪   BlockBuster
```

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in the suspension of the involved students.

Bonus Challenge With Tiny Awards

The first **three** students who

- successfully complete the assignment (get 100 points on the **Tur^{3}Bo Grader**), and
- develop an interactive interface for their game

will be rewarded with tiny awards.

Here is our implementation in action: Gameplay Recording