



CCPROG2 Project Specifications
Term 3, AY 2024–2025
Deadline: **July 28, 2025 (M)** before 0800

Onitama!

Onitama! is a two-player turn-based game. Each player has five pieces (one SENSEI, and four STUDENTS). To win the game, the player must either capture the opponent's SENSEI, or navigate his own SENSEI to occupy the opponent's BASE.

Components.

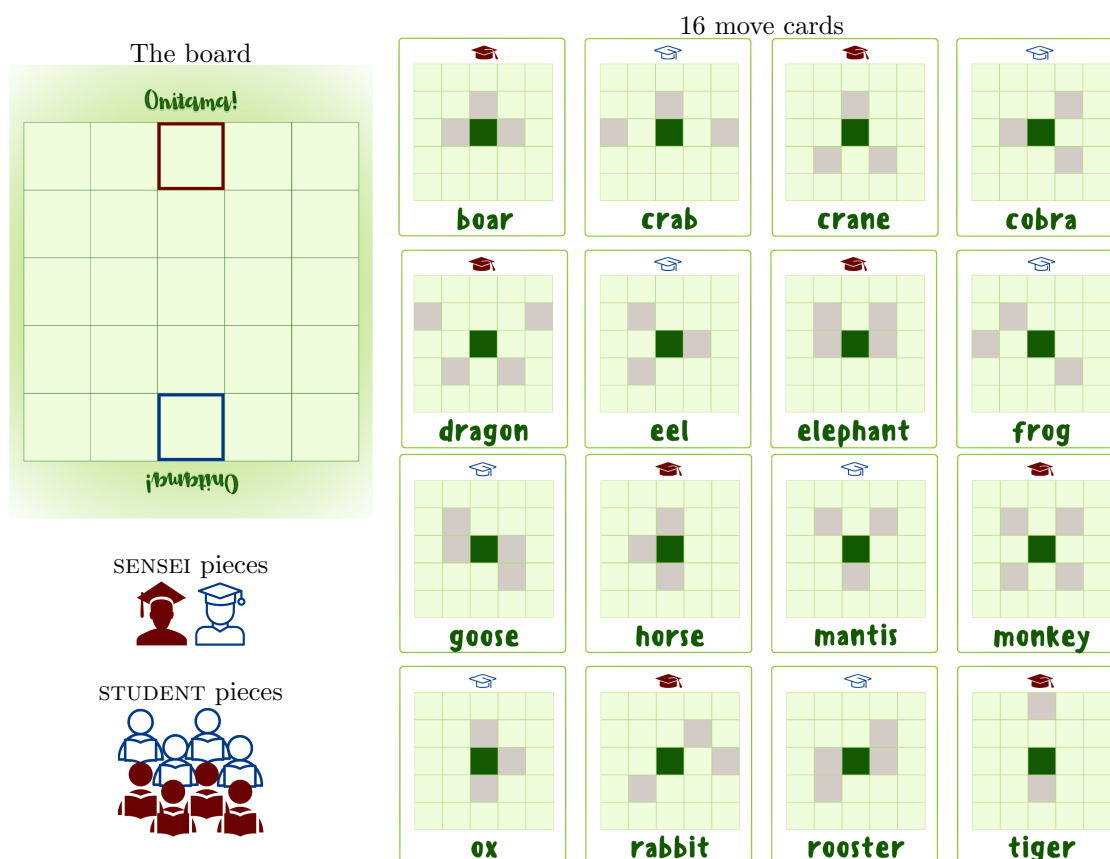




Figure 1: Components of the **Onitama!** game.

Setting up. The board is placed in between the two players. Each player plays a color—**red** or **blue**. On his own side of the board, the player positions his pieces, where the SENSEI is at the base, and the STUDENTS are on the same row on each side the SENSEI.

The move cards are shuffled, and distributed to each player. Each player receives two move cards, placed at their side of the board, face up. Another card is placed at the right side of the board.

The scholar's hat   on the card at the right side of the board, indicates the first player for this game.

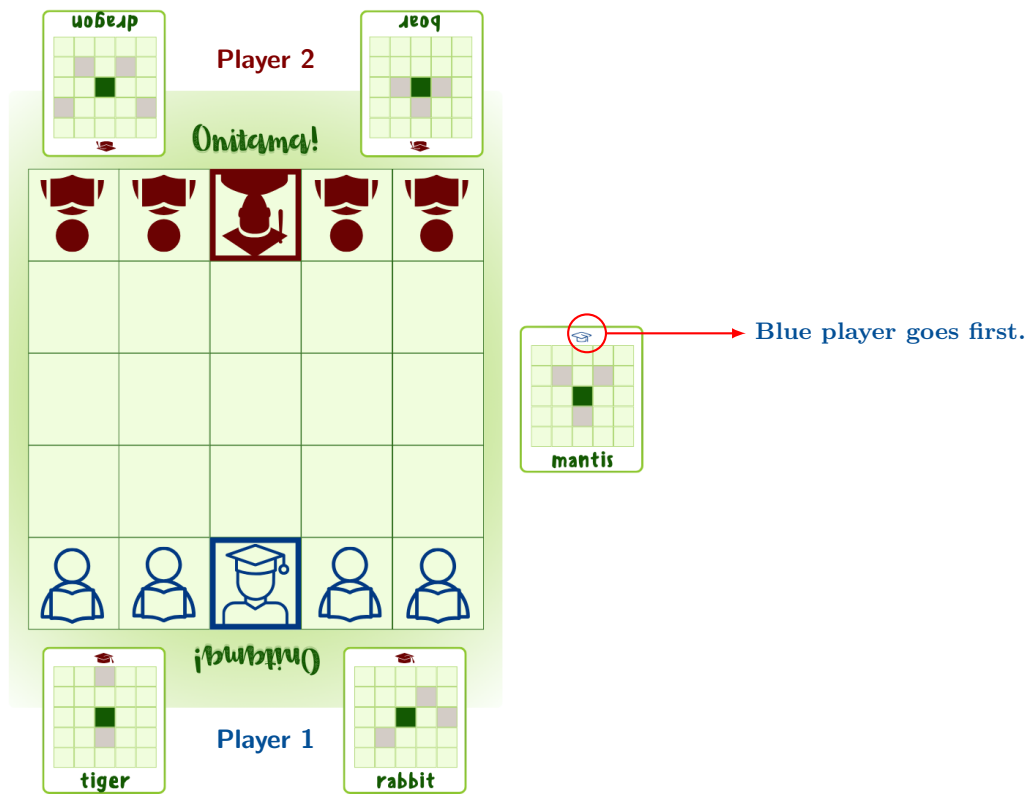


Figure 2: Game Setup.

Playing the game. Each player has two move cards he can choose from. A move card indicates the possible positions a piece can land on relative to its current position. For his turn, a player can choose from one of his two move cards to use. The player also chooses one of his pieces on the board to play. Based on the move card, the chosen piece can land on any valid space on the board, that is either unoccupied or occupied by an opponent piece.

For instance, in the illustration below, it is **Blue** player's turn. He can choose to play the Tiger card or the Rabbit card. Say, **Blue** plays the Rabbit card. As indicated in the move card, there are three possible positions (labeled on the card as **1**, **2**, **3**) to move the selected piece. If the selected piece to move is the **SENSEI**, then there is only one valid move left. The only valid move is the one labeled as **2** on the move card and it corresponds to the position marked with **X** on the board.

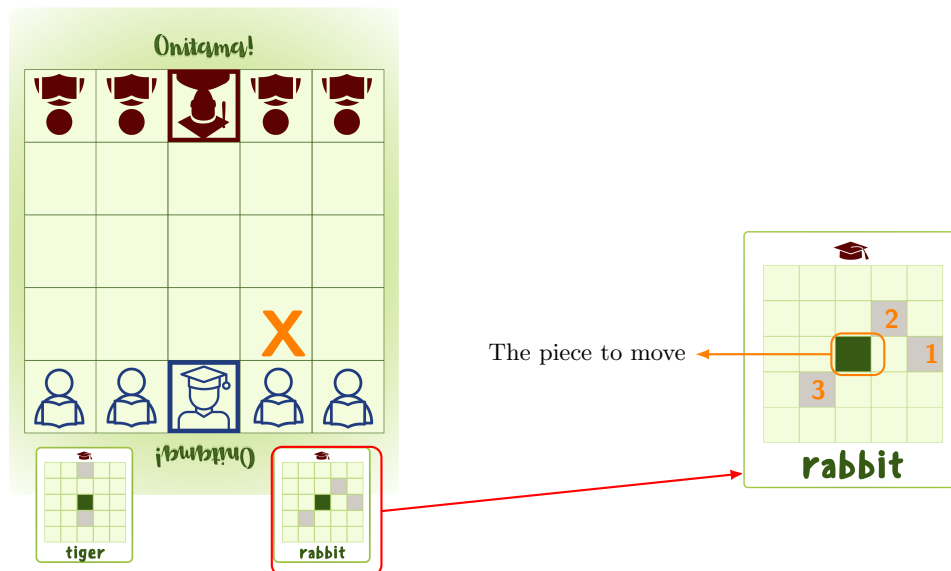


Figure 3: **Blue** player's turn; moving the **SENSEI** using the Rabbit card.

After the **Blue** player moves his **SENSEI**, the card played (Rabbit card) is placed at the left side of the board, and the **Blue** player collects the card that was on the right side of the board (Figure 4). It is now next player's turn.

Let's say the **Red** player played the Dragon card, and chooses to move a **STUDENT**. After moving the **STUDENT**, the played card (Dragon card) is placed on the right side of the board (Figure 5). The **Red** player collects the card on the left side of the board.

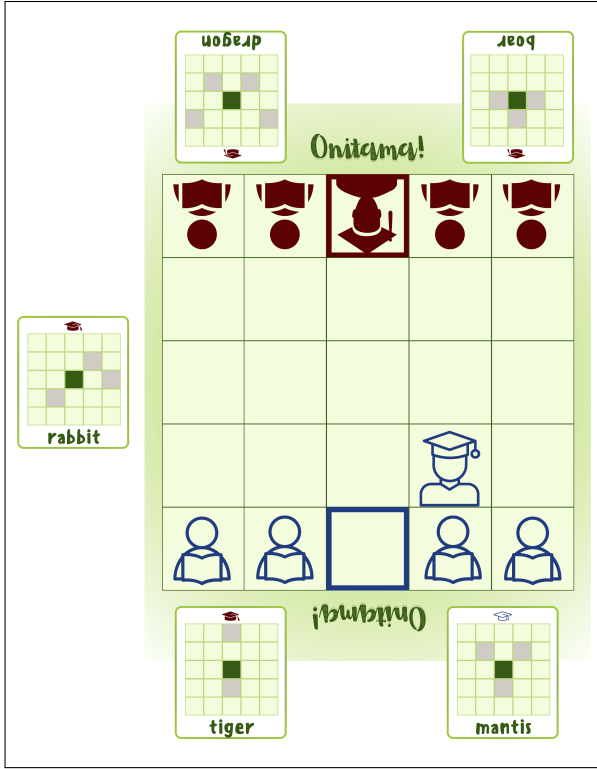


Figure 4: The **Blue** player moves **SENSEI**. The played card (**Rabbit**) is placed on the left side of the board, and **Blue** player takes the move card (**Mantis**) on the right side of the board.

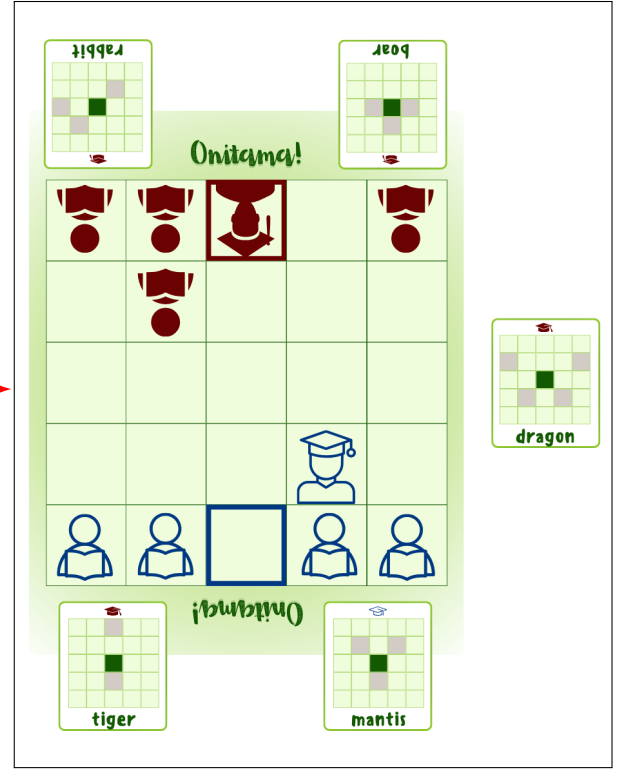


Figure 5: The **Red** player plays the **Dragon** card, and moves a **STUDENT**. The played card (**Dragon**) is placed on the left side of the board, and **Red** player take the move card (**Rabbit**) on the left side of the board.

If the landing position of the piece is occupied by the opponent, the opponent's piece is removed from the board, and the player may place his piece on that position.

The game continues, until one player captures the opponent's **SENSEI** or occupies the opponent's base.

No valid move. If, there are no valid move from any of two move cards of the current player, the player will not move any of his pieces. He will, however, still place one of his move cards on the side of the board, and collect the move card from the other side of the board.

Move Cards. Each move card is saved as a text file. The filename is the name of the move card, e.g. **Rabbit** card is saved as **Rabbit.txt**.

Each file contains six rows. The first row indicates the color of the first player—**red** or **blue**, and the next five rows contains the board and the positions. In general, each row has five dots . except for the positions/moves marked on the move card. The position of the piece is marked as **X**, and the possible destination positions are marked as **x**.

The **Rabbit** card and the contents of **Rabbit.txt** are shown in Figure 6.

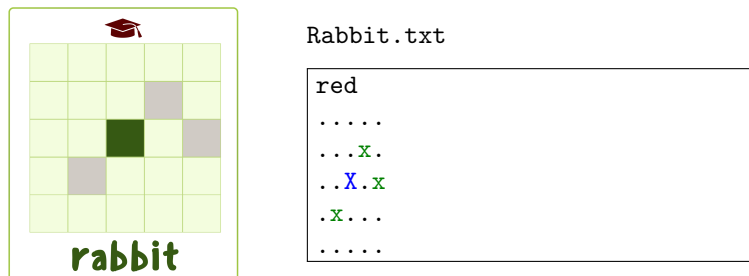


Figure 6: Sample move card and contents of the text file. **Rabbit** card and the contents of the **Rabbit.txt**

Master list of move cards. A text file name **movecards.txt** contains all move card names. The first row is an integer that indicates the number of move cards that will be used in the game. Each of the succeeding rows contains names of each card.

By default, its contents are as follows:

16
Boar
Crab
Crane
Cobra
Dragon
Eel
Elephant
Frog
Goose
Horse
Mantis
Monkey
Ox
Rabbit
Rooster
Tiger

This will be used to load the Move Cards that will be used in the game.

Shuffling of move cards. Before the start of each game, move cards are shuffled, and the result of the shuffling is displayed on the screen.

Interface. You should implement a **menu-driven application**. Generally, there should be no Yes-No questions (e.g. Are you sure? Do you want to continue? Play a game?, etc.) to be answered by the user. The user may choose to start a new game or exit the program.

The game board, the move cards of each player, the extra move card at the side should be *drawn* on the board.

In each turn, the player has to select a piece, a move card, and a valid landing position.

The Player. At the start of each game, names of players are collected. During the game, valid moves played by each player is recorded.



Bonus. Below are some features that may be added once the minimum requirement of this project, as described above, is implemented. Points awarded for the described features below vary. A maximum of 10 points may be awarded.

- **Replay.** After the result of the game is displayed, an option to show a replay of the game is available. If selected, a replay of the game is shown.
- **Hall of Fame.** The top 10 players of the game are recorded. This based on the number of moves executed to win the game. The least number of moves is ranked 1. The Hall of Fame is loaded at the start of the application, updated after every game, and stored before the application exits.

Submission Deadline: [July 28, 2025 \(M\) before 0800](#). All files (source code and header files, if any) to compile and run your program, must be successfully uploaded before the deadline. Only the files uploaded in AnimoSpace CCPROG2 MP will be used in grading your MP. **No late submissions will be accepted.**

1. The following are the deliverables:

Checklist:

- ☐ Upload in AnimoSpace by clicking **Submit Assignment** on Machine Project and adding the following files:
 - ☐ source code
 - ☐ test script 
 - ☐ declaration of original work 
 - ☐ email the softcopies of everything as attachments to [YOUR own email](#) on or before the deadline.

Legend:

- ♣ Test Script should be in a table format as shown below. There should be **at least 3 distinct test classes** (as indicated in the description) **per function**. There is no need to test functions which are only for screen design.

Function: <code>float getAreaTri(float base, float height)</code> // function name for the test cases below					
Function description: computes for the area of the triangle //short description of what the function does					
#	Test Description	Sample Input <i>either from the user or passed to the function</i>	Expected Result	Actual Result	P/F
1.	base is below 0	base = -1 height = 1.2	return 0.0
2.					
3.					
Function: // function name for the test cases below					
Function description: //short description of what the function does					
#	Test Description	Sample Input <i>either from the user or passed to the function</i>	Expected Result	Actual Result	P/F
1.
2.					
3.					

- 🔥 This is a **project by pairs** and must be designed, created, developed and completed by your group mate and yourself. No part of the code submitted should come from another source (online, books, other people). Place in the comment section the following to certify that the project you submitted is your own work.

This is to certify that this project is my/our own work, based on my/our personal efforts in studying and applying the concepts learned. We have constructed the functions and their respective algorithms and corresponding codes either individually or with my group mate. The program was run, tested, and debugged by my/our own efforts. I/We further certify that I/we have not copied in part or whole or otherwise plagiarized the work of other students/groups and/or persons.

<full name>

DLSU ID <number>

<full name>

DLSU ID <number>

- MP Demo:** You will present your project on a specified schedule during the last weeks of classes. Unable to show up on time during your demo schedule, or unable to answer convincingly the questions during the demo will merit a grade of **0** for your machine project. The project is initially evaluated via black box testing (i.e., based on output of running program). Thus, if the program does not compile successfully using `gcc` and execute in the command prompt, a grade of **0** for the project will be incurred. However, a fully working project does not ensure a perfect grade, as the implementation (i.e., correctness and compliance in code) is still checked.
- Any requirement not fully implemented and instruction not followed will merit deductions.
- This is either an **individual project** or by **group of two only**. Any form of cheating (asking other people not in the same group for help, submitting as your (own group's) work part of other's work, sharing your (individual or group's) algorithm and/or code to other students not in the same group, etc.) can be punishable by a grade of **0.0** for the course AND a discipline case.
- The MP should be an HONEST intellectual product of the student/s. For this project, you are allowed to do this individually or to be in a group of 2 members only.

Should you decide to work in a group, the following mechanics apply:

Individual Solution Even if it is a group project, each student is still required to create his/her INITIAL solution to the MP individually without discussing it with any other students. This will help ensure that each student went through the process of reading, understanding, solving the problem and testing the solution.

Group Solution Once both students are done with their solution: they discuss and compare their respective solutions (ONLY within the group) – note that learning with a peer is the objective here – to see a possibly different or better way of solving a problem. They then come up with their group's final solution – which may be the solution of one of the students, or an improvement over both solutions. Only the group's final solution, with internal documentation (part of comment) indicating whose code was used or was it an improved version of both solutions) will be submitted as part of the final deliverables. It is the group solution that will be checked/assessed/graded. Thus, only 1 final set of deliverables should be uploaded by one of the members in the Canvas submission page. *Prior to submission, make sure to indicate the members in the group by JOINing the same group number.*

Grading the MP grade will be the same for both students — UNLESS there is a compelling and glaring reason as to why one student should get a different grade from the other — for example, one student cannot answer questions properly OR do not know where or how to modify the code to solve the demo problem (in which case deductions may be applied or a 0 grade may be given – to be determined on a case-to-case basis).

Must Read!

1. You are required to implement the project using the **C language** (**C99** and **NOT C++**). Make sure to compile and run in both the IDE (DEV-C++) and the command prompt via

```
gcc -Wall <yourMP.c> -o <yourExe.exe>
```

Make sure you test your program completely (compiling AND running).

2. Your implementation has considerable and proper use of arrays, strings, structures, files, and user-defined functions, as appropriate, even if it is not strictly indicated.
3. It is expected that each feature is supported by at least one function that you implemented. Some of the functions may be reused (meaning you can just call functions you already implemented for other features. There can be more than one function to perform tasks in a required feature.
4. Debugging and testing was performed exhaustively. The program submitted must have
 - **NO syntax errors**;
 - **NO warnings**. Make sure to activate **-Wall** (show all warnings compiler option) and that C99 standard is used in the codes;
 - **NO logical errors**, based on the test cases that the program was subjected to;
5. **Do not use brute force**. Use appropriate conditional statements properly. Use, wherever appropriate, appropriate loops and functions properly.
6. You may use topics outside the scope of CCPROG2 but this will be self-study. **Goto** label, **exit()**, **break** (except in switch), **continue**, global variables, calling **main()**, **return** statements inside a loop are not allowed.
7. Follow a consistent coding standard. Use indentation to make your code more readable.
8. **Documentation..** While coding, you have to include internal documentation in your programs. You are expected to have the following:
 - File comments or Introductory comments
 - Function comments
 - In-line comments

Introductory comments are found at the very beginning of your program before the preprocessor directives. Follow the format shown below. Note that items in between **< >** should be replaced with the proper information. Items in between **[]** are optional, indicate if applicable.

```
/*
    Description:  <Describe what this program does briefly>
    Programmed by: <your name here> <section>
    Last modified: <date when last revision was made>
    Version:     <version number>
    [Acknowledgements: <list of sites or borrowed libraries and sources>]
*/

<Preprocessor directives>

<function implementation>

int main()
{
    return 0;
}
```

Function comments precede the function header. These are used to describe what the function does and the intentions of each parameter and what is being returned, if any. If applicable, include pre-conditions as well. Pre-conditions refer to the assumed state of the parameters. Follow the format below when writing function comments:

```

/*
    <Description of function>
    Precondiiton:  <precondition / assumption>
    @param <name>  <purpose>
    @return <description of returned result>
*/
<return type> <function name> ( <parameter list> )
{
    <implementation>
}

```

Example:

```

/*
    This function computes for the area of a triangle
    Precondition:  base and height are non-negative values
    @param base    the base measurement of the triangle in cm
    @param height  the height measurement of the triangle in cm
    @return the resulting area of the triangle
*/
float getAreaTri (float base, float height)
{
    <implementation>
}

```