

# Programming Principles

Semester 1, 2023

Object-Orientated Design and Implementation

Due Date: 02/06/2023

Weighting: 50%

Assessment Type: Individual

## Overview

Thank you for agreeing to work on this project for our organisation. I will give a little bit of a backstory, then get right into the details of what we want.

In the past few years, chess has had a bit of a resurgence, with a lot of people playing online against others and watching their favourite players stream. As you know, our company has a board game, **Advance**, which is a little bit similar to chess and we believe that this is a good time to test the waters and see if it can become successful online as well. We already have a team of developers putting together an online portal for playing Advance against other players around the world, but what we need from you is to create a **bot** that will play games automatically. This will allow players to practice before playing other humans, as well as being a valuable source of testing data for the online service.

What we need from you is to create a C# .NET Core 6.0 console application that will read in a board state from a text file, make a move, then write out the board state with that move made. We will pass information to your program in the form of command-line arguments and invoke it multiple times to result in it playing a game. In the next section we will go over the rules of Advance.

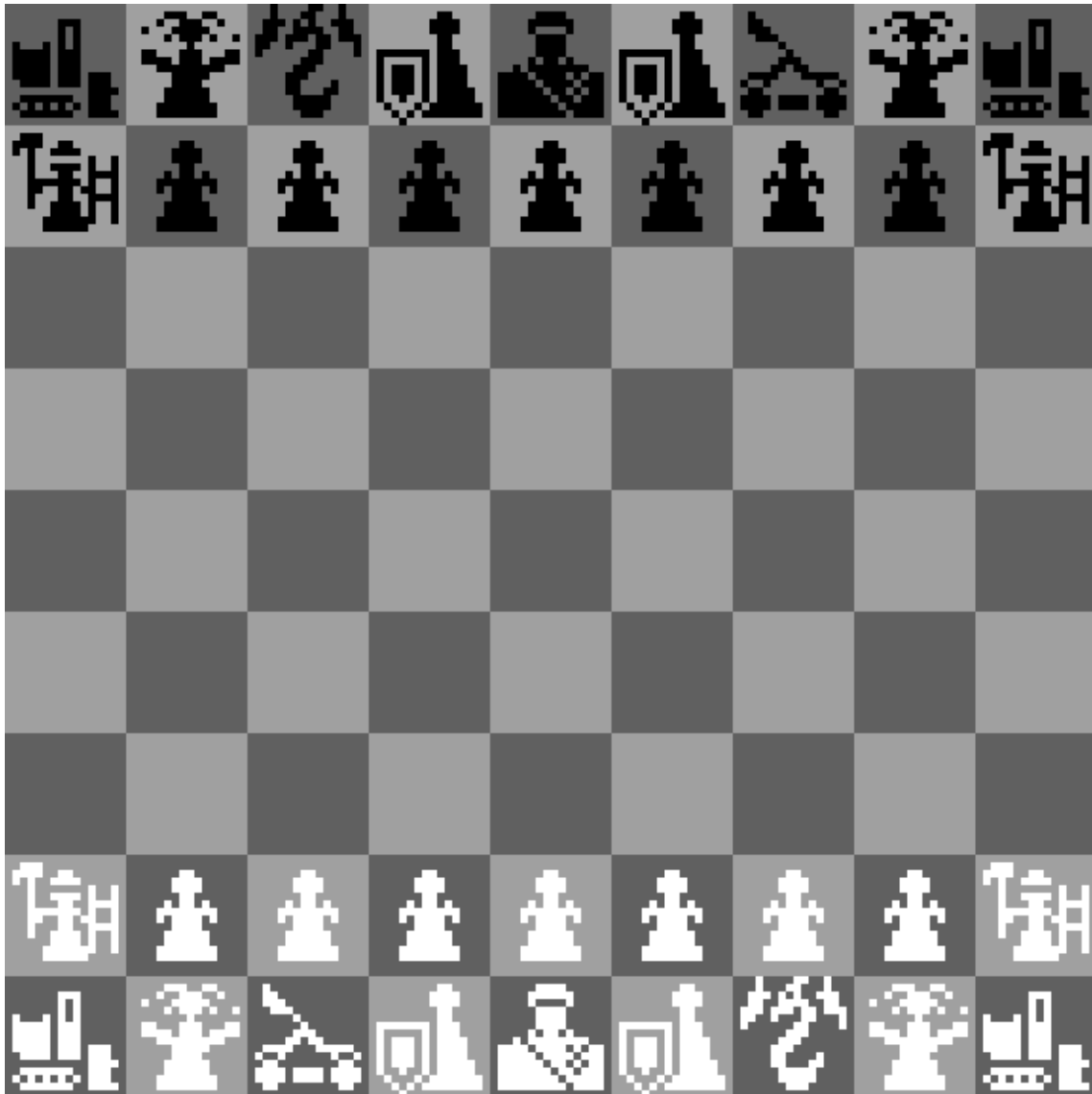
## The rules of 'Advance'

Advance is played on a 9x9 board, with two sets of pieces: one set white, the other set black. One player is in charge of each set of pieces, and the player playing white goes first.









On each turn, the current player gets to move a piece according to the rules for that particular piece. Different pieces have different rules as to how they can move and what they can do.

## The initial board state

The initial board state looks like this:



Each player starts with the following pieces:

- 7 Zombies:  (represented with 'Z' or 'z' in the text format)
- 2 Builders:  (represented with 'B' or 'b' in the text format)
- 2 Miners:  (represented with 'M' or 'm' in the text format)
- 2 Jesters:  (represented with 'J' or 'j' in the text format)
- 2 Sentinels:  (represented with 'S' or 's' in the text format)
- 1 Catapult:  (represented with 'C' or 'c' in the text format)
- 1 Dragon:  (represented with 'D' or 'd' in the text format)
- 1 General:  (represented with 'G' or 'g' in the text format)

## Winning the game

There are two ways to win the game. The first is to trap the opponent by removing all possible ways of making a move. If it is a player's turn and they are unable to make a legal move, they lose the game. The other is that, after 100 full turns (that is, after both players have made 100 moves) the player with the greatest material advantage remaining wins. This is calculated based on the following formula: 1 point for each zombie, 2 points for each builder, 3 points for each jester, 4 points for each miner, 5 points for each sentinel, 6 points for each catapult and 7 points for each dragon. The player with the highest score wins. If both players have the exact same score, the game ends in a tie.

In the next section we will describe how each of the pieces move.

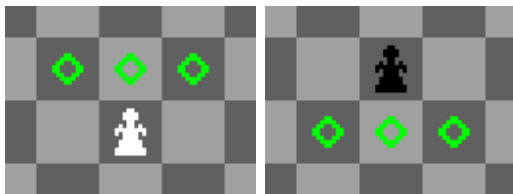
## How the pieces move

### Zombie (Z)

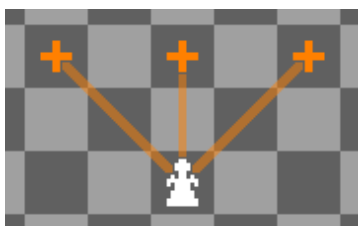


**Score: 1**

Zombies can move to and capture pieces on any of the three adjoining squares in front of the Zombie (that is, facing upwards for white and downwards for black) like this:



In addition, if there is an enemy piece two squares away in any of those three directions and the intermediate square is empty, a Zombie can perform a leaping attack, capturing the piece on that square:



However, it can only perform this move if there is an enemy piece there that can be legally captured.

As the Zombie can only advance, once it reaches the back row (the top row for white, the bottom row for black) it will no longer be able to make any moves.

In this example, there is an enemy Dragon, but because there is a friendly Miner in the way, the Zombie cannot make the leaping attack and capture the Dragon:



However, if the Miner is gone, the Zombie can capture the Dragon as follows:



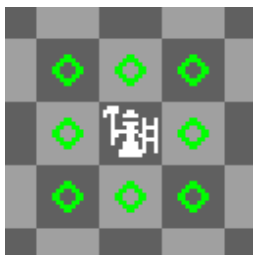
The Zombie can also capture a piece that is within its normal movement range as well – the leap attack is merely optional.

#### Builder (B)



**Score: 2**

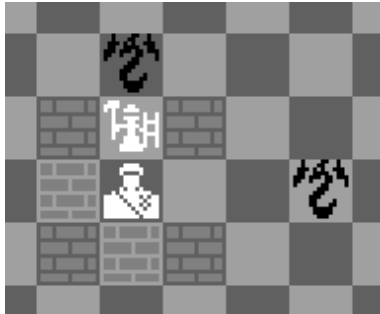
The Builder can move and capture on any of the 8 adjoining squares:



Builders can also build **walls** on any of the 8 adjoining squares as well, as long as there is nothing occupying that square. The builder does this without moving:



Walls are special pieces. They do not belong to either player and they do not move. However, they do obstruct other pieces as neither player can move a piece into a square that is being occupied by a wall. The only way to get rid of a wall is to capture it with a Miner. Walls can be used strategically, in order to limit the path of movement for enemy pieces. For example:



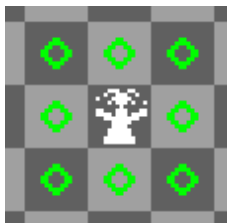
Here, the white General is under attack from an enemy Dragon. The Builder cannot move to block the Dragon because this would place the General in danger from the other Dragon. However, the Builder can build a wall to the right of the General, protecting the General from the Dragon on the right.

Jester (J)



**Score: 3**

The Jester is the only piece that cannot capture other pieces. It can move to any of the 8 adjoining squares, and it has two special abilities that make it a versatile, useful piece.



The first ability is that the Jester is nimble and can exchange places with a friendly piece, if it is on one of the adjoining squares. The only limitation here is that the Jester cannot exchange places with another Jester (as this would result in no change to the board state.) This means Jesters can quickly move throughout your ranks as you do not need to make room for them. In this example, a Jester swaps places with a friendly Zombie to move to the front row:



The other ability a Jester has is to convince enemy pieces to change to your side. This ability can be used on an enemy piece (**other than the enemy General**) in one of the 8 adjoining squares and changes that piece into one of your pieces. The Jester performs this action without moving:



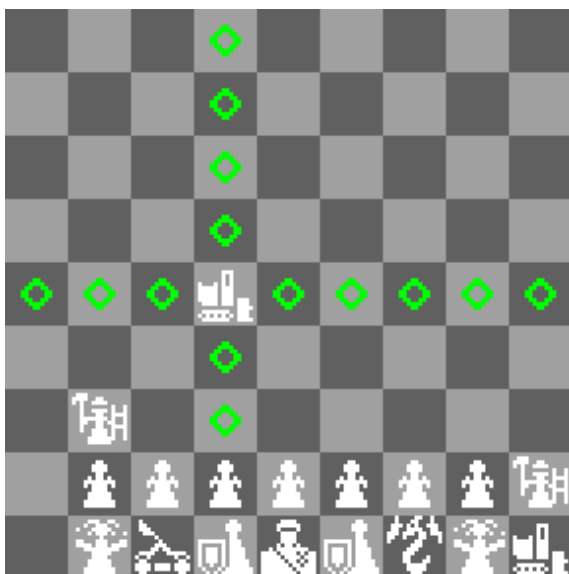
This is a very powerful ability, and can allow you to gain multiple Catapults and Dragons on your side.

### Miner (M)



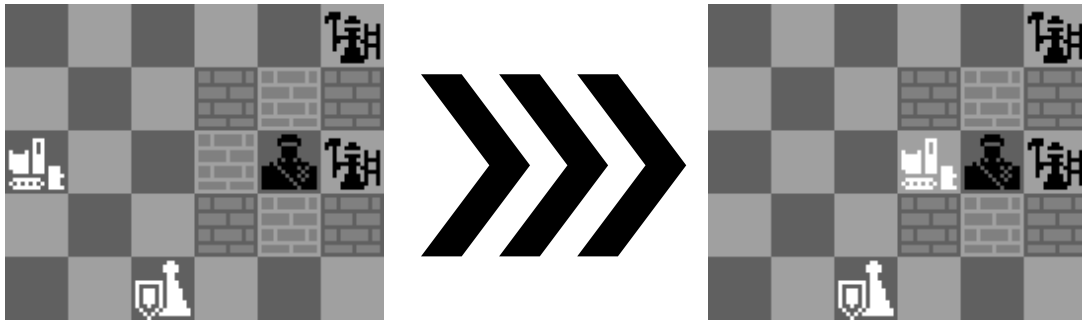
**Score: 4**

The Miner moves like a Rook does in chess; it can move any number of squares in one of the 4 cardinal directions (up, down, left and right) and can capture at any of those positions as well:



In addition, the Miner is able to capture walls, which can make it a useful piece to use in your offensive if the enemy has used Builders to place fortifications. The Miner captures walls in the same way it captures pieces, and it is subject to the same limitations: it can only capture one piece (or wall) in a move and there must be an unobstructed path to that piece in one of the cardinal directions.

In this example, the enemy General is protected by walls. The Builder can move in and capture a wall, trapping the enemy General and winning the game:



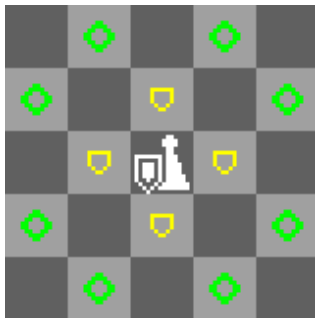
Sentinel (S)



#### Score: 5

The Sentinel is a strong protective piece. It moves and captures similarly to a Knight in the game of chess, moving two squares in one cardinal direction and then one square in a perpendicular direction, jumping over any intervening pieces (or walls).

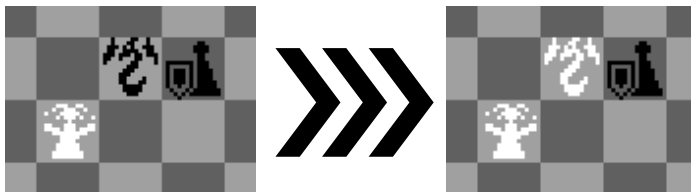
In addition to this, the Sentinel **protects** any friendly pieces (including other Sentinels) on the 4 adjoining squares in cardinal directions, preventing enemy pieces from capturing your pieces on those squares:



(The circles represent squares the Sentinel can jump to. The shields represent the Sentinel's protection range.)

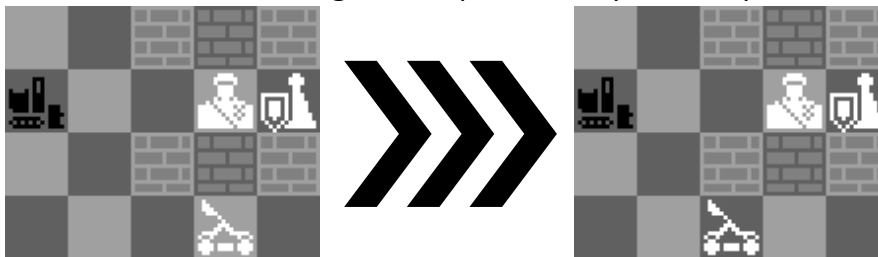
The Sentinel's protection effect comes with the following rules:

- A Sentinel will protect any piece of the same colour on those 4 squares, including other Sentinels.
- Enemy pieces will not be able to capture a piece protected by a Sentinel. This includes the Catapult's ranged attack. However, it does **not** include the Jester's conversion move, which can be used even if the piece is protected by a Sentinel:



In this example, the enemy Dragon can be converted despite being protected by the enemy Sentinel.

- The General is not **in danger** if it is protected by a friendly Sentinel:



In this example, the General is not in danger from the Miner due to the presence of a Sentinel, and therefore we can move the Catapult into position to threaten the Miner.



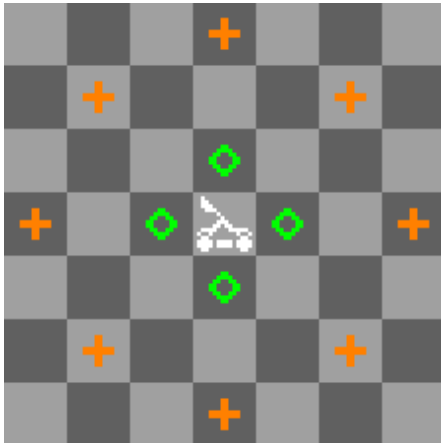
## Catapult (C)



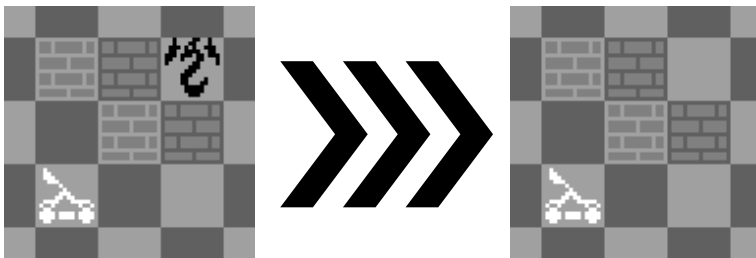
**Score: 6**

The Catapult can only move 1 square at a time, and only in the 4 cardinal directions. Also, it can only move to those squares – it cannot capture enemy pieces on those squares.

However, the Catapult is able to capture pieces that are either 3 squares away in a cardinal direction or 2 squares away in two perpendicular cardinal directions:



The Catapult's attack is special, in that it can remove an enemy piece on one of its capture squares, but it does not itself move when doing so. It also does not matter if there are other pieces in the way, as the Catapult's projectile flies over the battlefield.



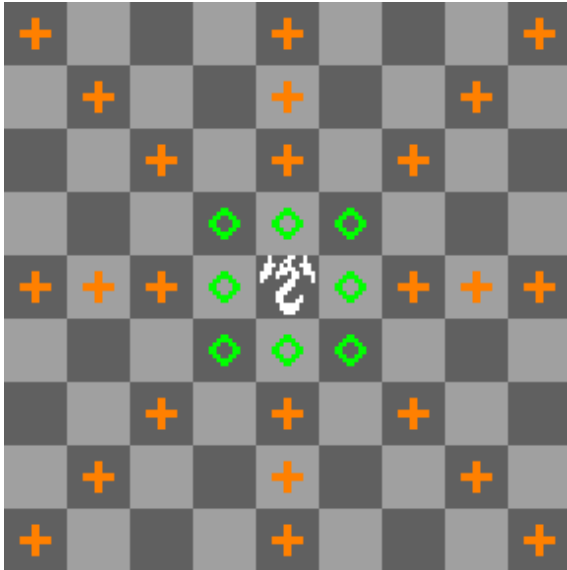
In this example, the catapult can capture the enemy Dragon even though it is surrounded by walls.

## Dragon (D)



**Score: 7**

The Dragon is a powerful piece that can move any number of squares in a straight line in any of the 8 directions. It is most similar to a Queen in the game of chess, except for one downside: the Dragon cannot capture any piece it is immediately next to.



The Dragon can move to the squares indicated by the circles, but cannot capture pieces on them. In addition, the Dragon moves in a straight line, and cannot move to or capture a piece if there are pieces or walls in the way.



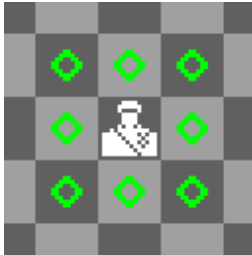
In this example, the Dragon cannot take the enemy Miner as it is too close. It can, however, capture the enemy Catapult.

## General (G)



Score: N/A

The General functions almost identically to the King in chess. It can move and capture on any of the 8 adjoining squares, like the Builder.



However, the General is not allowed to be captured. Moving the General to a square where an enemy piece could capture it (moving it **into danger**) is an illegal move. In addition, if your opponent makes a move that places your General in the attacking range of one of their pieces (putting your General **in danger**), you must get your General out of danger on the next move (e.g. by moving your General, capturing the threatening piece or obstructing it from being able to attack your General by moving a piece into its way or building a wall) – any move that does not achieve this is an illegal move. For this reason, targeting the enemy General is usually the key to winning games – once the General is in danger, the player controlling that General usually has a very limited selection of moves, and can be led into a situation where they have no possible moves at all and lose as a result.

## Your software

### Basic operation

Your program will read in a file path containing a board state, play a single legal move as either the white or black player (depending on what is passed as the first command-line argument), then write out the board state resulting from that move to a file. In this way, your program will play one turn each time it is run.

We will pass your program **three command-line arguments**:

- The text 'white' or 'black' depending on whether the bot is playing white or black this turn. Alternatively, this argument could contain 'name' instead, in which case your program is to output the bot's **name** and exit. See [naming your bot](#). If 'name' is passed to your program, no other arguments will be.
- The filename path to the text file that contains the current board state.

- The filename path to the file where the new board state is to be written after making a move. This could be the same as the previous path, so you will have to make sure to close that file after opening it.

Your program will just make a single move, then write to the file. We will invoke your program multiple times to get it to play a game.

Your program will be given the board state in the form of a text file consisting of 9 lines of text, each 9 characters long. The original board state looks like this in text format:

mjdsgscjm	
bzzzzzzzb	
.....	
.....	
.....	
.....	
BZZZZZZB	
MJCSGSDJM	

Lowercase letters are black pieces. Uppercase letters are white pieces. There are two other characters: . and #. The . character indicates an empty square, while the # character indicates a wall (a piece that does not belong to either player but is placed by the Builder.)

Your program needs to analyse the board state, determine a legal move and then make that move. For example, if your program is playing white and chooses to advance one of the zombies by one square, your program will write this to the output file:

mjdsgscjm	
bzzzzzzzb	
.....	
.....	
.....	
...Z.....	
BZZ.ZZZB	
MJCSGSDJM	

We will then invoke your program repeatedly in order to get it to play a game. You can do this yourself during testing, and even play your bot against itself to make sure things are working correctly.

Your program does **not** need to check that the board state given to it is legal. For instance, your program will never be given an input file where black has a piece that is putting white's General in danger and it's black's turn to play – because then black could take white's General and this is not something that can happen in a game of Advance (just like you can never take a King in chess.) However, your program **does** need to perform a legal move for the specified colour. There will **always** be a legal move that your bot can play. If there is no

legal move available for that player, the player will lose the game and your program will not be invoked.

## Levels of functionality

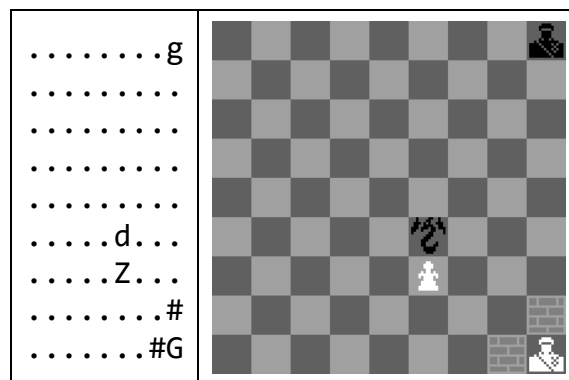
Based on the grade level you are hoping to achieve for the 'Functionality' marking criterion, you will need to implement a certain amount of game logic into your program. The most basic level required for a passing grade against this criterion simply requires that your program know the rules of Advance and be able to play legal moves, while later levels require that your program be able to employ certain levels of reasoning to produce better moves.

Note that the functionality described is only what is necessary to get the **minimum** percentage score for that grade level. If you want a higher score, you will need to make some progress towards a higher functionality level. For example, if you meet all of the requirements for a functionality grade of 4 and none of the higher requirements, you will receive exactly 50% for functionality.

### Functionality grade 4: Plays legal moves

To meet this level of functionality, your program must always produce a legal move if given a legal Advance board state. This means your program needs to know how every piece moves and the restrictions that pieces must follow when moving. This includes following rules like keeping the General out of danger.

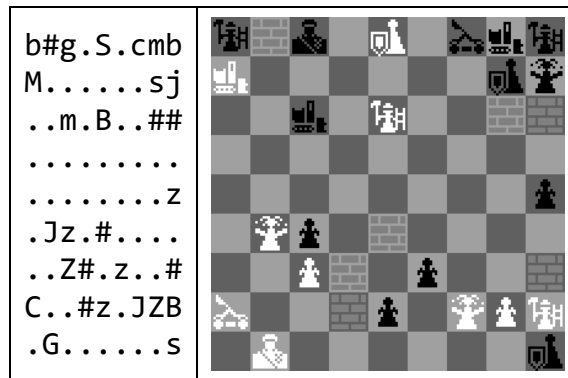
Here is one of the tests your program will be tested with (playing as white):



In this example, the white General is in danger due to being attacked by a Dragon. The General cannot move out of the way due to the walls (the only exit is still in the Dragon's path). Therefore the only legal move that can be made here is to use the Zombie to capture the Dragon.

### Functionality grade 5: Must make winning move if possible

To meet this level of functionality, your program must not only always make legal moves, but if there is a move it can make that will result in the current player immediately winning the game, that move must be made. (If there are multiple such moves, one of them must be made.)

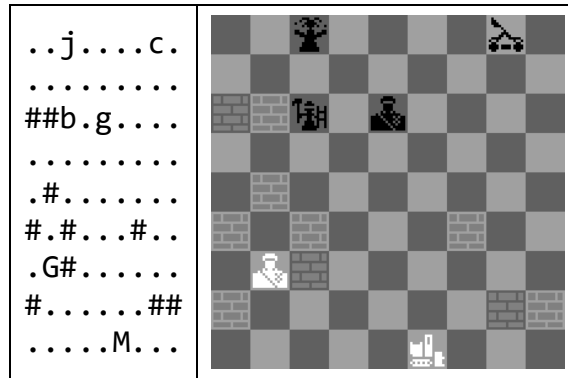


In this example (playing as white) we can win the game in one move by moving the Builder in the 3<sup>rd</sup> row one square up and to the left. This places the enemy General in danger. The General can't move out of the way and it can't capture the Builder either, as the Builder is protected by the Miner. In order to achieve a grade of 5 your program must successfully play the winning move if given a board state where such a move exists.

### Functionality grade 6: Obtaining material advantage

Material advantage is important in Advance; more pieces puts you in a better position to defeat your opponent, and the alternate win condition is to have a greater material score than your opponent after both players have each made 100 moves. Accordingly, your program needs to try and maximise material. Your program must still function as described previously- only making legal moves and making winning moves if any are available, but when determining which non-winning move to play, you must take the one that maximises your material score – in other words, by capturing enemy pieces (or better yet, converting them with the Jester.) Use the same scoring table used to determine the winner (Zombie=1, Builder=2, Jester=3, Miner=4, Sentinel=5, Catapult=6 and Dragon=7) and, out of the available legal moves, pick the one that maximises your advantage over your opponent. (If there are multiple such moves that give the maximum score advantage, pick one of them.)





In this example, white cannot capture any enemy pieces, and so must make a move that will put white into a good position next time. There are 20 legal moves white can make from here: moving the General to one of the 4 surrounding squares and moving the Builder to either one of the 8 available squares on the bottom row or one of the 8 available squares in the sixth column. Most of these moves are material-advantage-neutral because the only enemy piece that can attack with range is the Catapult in the top-right corner. In order for any of those pieces to take our Builder we would have to move the Builder up to them, allowing either the Catapult or General to capture it. Obviously we do not want to do that- it cannot possibly result in a material advantage for us because we have no pieces that can take advantage of this to capture something better. However, let's say we move the Builder to the very top row. This allows it to attack both the Catapult and Jester. The opponent is assumed to use the functionality level 6 program, which could do anything because every move the opponent could make in response to this is materially neutral. However, what the opponent cannot do is save both the Jester and Catapult, so we will be taking one of those in the next turn. This means the only move we can take, if we have a functionality level 7 bot, is to move the Builder to the top row. No other move guarantees the taking of an enemy piece on the next turn.

Some other things to note:

- You might predict that, if you make a particular move, your opponent will be able to immediately win. In that case, apply a large penalty such that you will not make that move (unless there is no choice).
- You might also predict that, if you make a particular move and your opponent responds as predicted, that you will be able to immediately win on the next turn. In that case, apply a large bonus to that move to ensure that you pick it.



### Functionality grade 100%: Higher order reasoning

If your program implements functionality level 7 perfectly (that is, it passes all 200 test cases) you will get at least a score of 85% against the functionality criterion. Achieving a higher score here requires you to make more improvements to your bot while still following the rules for functionality level 6 and earlier – in other words, you will need to add your own logic to break ties when there are multiple equally good moves to make at functionality level 7. (Otherwise your program will not succeed in passing the earlier levels and will therefore not even be tested against this higher level.) If your program passes all 200 of the earlier test cases, we will have your program play 50 games against a bot implementing functionality level 7. Your bot is expected to win 25 of those games, but winning more often will result in a higher % in the functionality criterion.

The exact way you will be marked for this is described in the [How you will be marked](#) section.

### What you need to submit

You will submit your assignment via GradeScope. As with the previous GradeScope assessment items, you will need to submit your source files. It is not necessary to submit your .csproj file (we will replace it with our own .csproj, which also means you cannot add Nuget dependencies to your project or change the configuration). In addition, you will submit a file named 'Report.txt' (along with your .cs source files) containing a **plain text** report on the design of your software. More information about the report is below.

**Submit Programming Assignment**

Upload all files for your submission

Submission Method

☒ Upload ☐ GitHub ☐ Bitbucket

Add files via Drag & Drop or [Browse Files](#).

Name	Size	Progress	✕
Jester.cs	6.1 KB	<div></div>	✕
Miner.cs	4.3 KB	<div></div>	✕
Piece.cs	6.1 KB	<div></div>	✕
Program.cs	5.1 KB	<div></div>	✕
Report.txt	5.9 KB	<div></div>	✕
Sentinel.cs	5.2 KB	<div></div>	✕
Zombie.cs	4.7 KB	<div></div>	✕
Builder.cs	6.1 KB	<div></div>	✕
Catapult.cs	4.7 KB	<div></div>	✕
Dragon.cs	5.6 KB	<div></div>	✕
General.cs	0.2 KB	<div></div>	✕

(Example submission to GradeScope. You don't need to have these exact class names, of course.)

Your program's functionality (its ability to play games of Advance) will be marked on GradeScope with a set of 200 predefined test files. These files are available on Canvas so you can test them on your code without submitting to GradeScope. More information about the automated marking process will appear later in this report. The other aspects of your software (your design, documentation and code quality) will be marked by teaching staff after the due date has passed, but you will still receive your results through GradeScope.

## The report

Your submission must include a plain text report with the filename 'Report.txt' describing your software – how your classes fit together, how your program evaluates board states, determines legal moves etc. If you implemented higher order reasoning, your report should also describe what you did and why you believe this will make your bot win more often. This will be used as part of our evidence for determining what mark to give your object-oriented design. This report does not need to be very long- 1000 words or so.

## Documentation

You are required to fully document the source code of your implementation. This will involve two types of in-source documentation:

- [XML-style comments](#) to document your classes, methods and any other custom types (e.g. enums, structs) that your code defines. Classes and other custom types must have at minimum a <summary> tag describing the class. Methods must be documented with a <summary> tag describing the method, a <param> tag for each parameter and a <returns> tag if it returns anything.
- Line comments or block comments within the body of your methods, describing in general terms what your code is doing. There should not be too many of these in each method unless you are describing something particularly complex, and they should not repeat things that are obvious from the code. Use comments to describe the intent of the code.

## Object-oriented design

This assignment is designed to test your ability to design an object-oriented solution to this problem. The design is up to you, but you will be graded on it. There is ample opportunity in this assignment to make use of inheritance and polymorphism to greatly simplify this task, which is an inherently complex one if you use only imperative programming techniques.

You do not need to draw a UML diagram or document your design explicitly (other than an overall description of what you are doing in the report) – we will look at your code and determine the level of achievement to award your submission based on your object-oriented design.

## Exceptions

You are required to use exceptions to gracefully handle unexpected results. Even though we will not test your program with invalid input in the functionality stage, you are still required to handle it (by displaying a reasonable error message and then aborting) and it will affect your design mark. Examples of things you need to handle include:

- The first argument passed to your program is something other than 'white', 'black' or 'name'.
- Insufficient arguments passed to your program (3 arguments if the first is 'white' or 'black' and 1 if it's 'name').
- Unable to open up the file path in the 2<sup>nd</sup> argument.
- File contains invalid characters (characters other than ZBMJSDCGzbmjsdcg.# and newline).
- File contains the wrong number of characters (too few or too many to specify a board state).
- Impossible to make any legal moves
- Unable to open the file in the 3<sup>rd</sup> argument for writing.

## How you will be marked

There are three criteria your submission will be evaluated against: **functionality**, **design** and **documentation**. Design and documentation will be marked as per the CRA, and this will happen after the assignment is due. Functionality will be marked automatically, and it will be marked when you submit. This gives you multiple opportunities to fix your code and resubmit, up until the deadline.

We will first run your program against 50 test cases (25 as white and 25 as black) to ensure that your code is capable of producing legal moves in virtually every scenario. These first 50 test cases are designed so that there is **only one legal move** that can be made. If your program makes that move, it will pass the test case. Otherwise, it will not.

If you fail any of these test cases, you will receive a mark for functionality based on the number you passed – if you passed 49, you will receive a 49% for functionality, and so on. If you pass every test case, you will receive a passing mark of 50%, and will be tested on additional test cases for higher grade levels.

The test data your program will be evaluated against is fixed, but note that **if your program hardcodes answers to our test data you will receive 0% for functionality**. You must actually implement logic in C# for finding legal moves, not take advantage of the fixed test data. We will determine this by running your code against some hidden test cases as well. There is no penalty for failing these hidden test cases, but it failing these will cause us to look at your code carefully and determine if you are hardcoding in responses, in which case your functionality mark will be 0%.

If your program passes all 50 test cases, you will receive a minimum score of 50% for functionality. We will then run another 50 test cases (25 white 25 black) to determine if your program meets [functionality grade 5](#). These test cases all have multiple legal moves that can be made, but one of those moves will result in the player immediately winning that game. Your program will pass the test case if it makes the move and fail otherwise. If your program does not pass all of the test cases, you will receive a functionality score proportionally between 50% and 65% based on the number of tests you do pass (each test in this category being worth 0.3%).

If your program passes all 100 of the test cases so far, you will receive a minimum score of 65% for functionality. We will then run another 50 test cases (again, 25 white and 25 black) to determine if your program meets [functionality grade 6](#). These test cases will have multiple legal moves (none of which will result in an immediate win), but one will result in the greatest material score relative to the opponent's, and this is the move your program must make. Your program will pass the test case if it makes the move and fail otherwise. If your program does not pass all of the test cases, you will receive a functionality score proportionally between 65% and 75% based on the number of tests you do pass (each test in this category being worth 0.2%).

If your program passes all 150 of the test cases so far, you will receive a minimum score of 75% for functionality. We will then run another 50 test cases (again, 25 white and 25 black) to determine if your program meets [functionality grade 7](#). These test cases will have multiple legal moves, and multiple moves that would result in the highest achievable material score, but only one move that results in the greatest relative score when predicting our opponent's next move and our own following move. Your program will pass the test case if it makes the move and fail otherwise. If your program does not pass all of the test cases, you will receive a functionality score proportionally between 75% and 85% based on the number of tests you do pass (each test in this category being worth 0.2%).

Finally, if your program has passed all 200 test cases, you will receive a minimum score of 85% for functionality. We will then have your program play 100 games of Advance against a functionality-grade-7-compliant bot (in 50 of the games your bot will be playing white; in the other 50, black). This bot is designed to, after having determined the best moves to make with grade 7 logic, if there is more than one move with the same level of preference, just pick one of those moves at random. If your program is compliant with functionality grade 7 we expect it to win about half the time. However, if your program wins more than 30 of the games, each game you win beyond that point will add an additional 0.5% to your functionality score (up to a maximum of 100%) – this means that to get a mark of 100% for functionality, your submission will have to win 90+ games.

## Addendum

Initially, we will do all of this marking via GradeScope, including the 50 games against a bot for determining functionality scores between 85% and 100%. However, we do not know how long this will take to run. GradeScope will time out submissions that take over 40 minutes and 500 games could mean up to 10,000 moves, which means running over even if

each move takes only 0.25 seconds to compute. We will keep an eye on this when submissions get close to passing all test cases, and if we start seeing many timeouts, we will remove this part of the marking criteria from GradeScope and do the simulated games for those submissions with functionality scores of 85% offline at a later date.

## Tournament

As a (possibly) fun little diversion I am planning to run a little tournament out of your submissions. The way this will work is that I will take the submissions that pass at least the first 50 test cases and pit them against each other in a tournament. I'll then make videos out of the games and post the results.

I will not reveal any personal information about the students behind the submissions- however, for the sake of allowing you to personalise your entry (and so that you can identify your own bot in the tournament), you can give your bot a **name**.

## Naming your bot

The limits for bot names are:

- Maximum of 32 characters
- No non-ASCII or non-printable characters
- Nothing offensive, explicit, illegal, political or otherwise questionable (I'll be the judge of this, and my decision is final and unappealable)

To provide your bot name, simply have your program print out its name with `Console.WriteLine()` when invoked with just the command-line argument 'name'.

## When the tournament will take place

Not currently sure about the timing. I would like to run a few of them at least. It will depend on how many functional submissions we get and when we get them. There is nothing special you need to do (other than handling the 'name' argument) to enter the tournament.