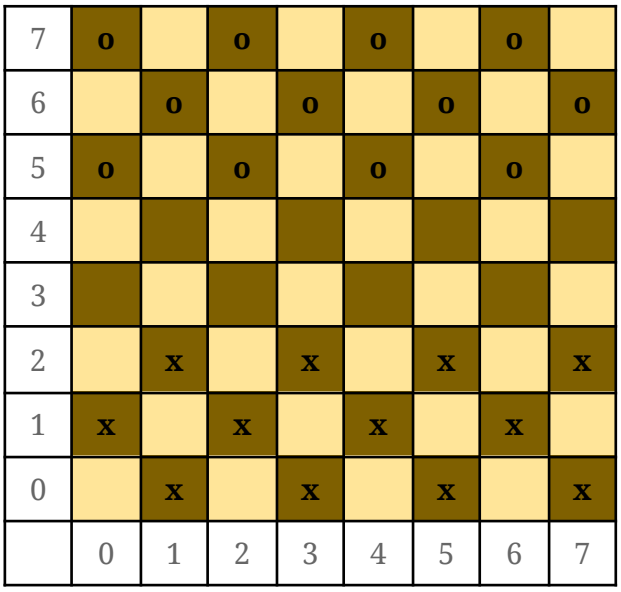
Progetto CTO442
anno 2021/2022

Il progetto consiste nella definizione di una classe per il gioco della **dama** (con regole leggermente semplificate, leggi sotto): https://it.wikipedia.org/wiki/Dama



1. Regole del gioco

I giocatori sono due: **player1** (rappresentato con delle **x** nell'immagine sopra) e **player2** (rappresentato con delle **o** sopra). Il gioco ha luogo su una scacchiera 8x8. Ad ogni istante, i pezzi si trovano solo sulle celle marroni (vedi immagine). I **pezzi** sulla scacchiera sono di due tipi:

- "Pedina": sono i pezzi inizialmente a disposizione di ogni giocatore (all'inizio, 12 per giocatore). Nelle immagini successive, verranno indicate con **o**/**x**

- “Dame”: sono pezzi con mosse speciali. Quando una pedina di un giocatore raggiunge la riga opposta (riga 7 per player1 o riga 0 per player2), si trasforma automaticamente in Dama. Nelle immagini successive, verranno indicate con **O/X**

Tutti i pezzi possono muoversi in solo due modi:

1. Ad una cella marrone adiacente in diagonale, non occupata da un altro pezzo:
    - le Pedine di player1 possono solo muoversi in alto a destra (ossia incrementando il numero di riga e di colonna di 1 unità) oppure in alto a sinistra (ossia incrementando il numero di riga di 1 unità e decrementando il numero di colonna di 1 unità).
    - In modo analogo, le Pedine di player1 possono solo muoversi in basso a sinistra o in basso a destra.
    - Le Dame di entrambi i giocatori possono muoversi in diagonale di una cella in qualsiasi direzione (alto/sinistra, alto/destra, basso/sinistra, basso/destra).
2. In diagonale di due celle per mangiare un pezzo avversario: in questo caso, la cella diagonale adiacente deve contenere un pezzo avversario (che viene mangiato), mentre la cella successiva in diagonale deve essere vuota per consentire lo spostamento. Esempio:

7	o		o		o		o	
6		o		o		o		o
5	o		o				o	
4								
3					o			
2		x		x		x		x
1	x		x		x		x	
0		x		x		x		x
	0	1	2	3	4	5	6	7

La **x** nella cella azzurra si può spostare nella cella azzurra in alto a destra (libera), mangiando la **o** nella cella rossa. Dopo lo spostamento, la **o** nella cella rossa viene rimossa dalla scacchiera.

Ulteriori regole per mangiare un pezzo:

1) le Pedine di player1 possono solo mangiare in direzione alto/sinistra oppure alto/destra. Viceversa, le Pedine di player2 possono solo mangiare in direzione basso/sinistra oppure basso/destra.
2) Le Pedine possono solo mangiare altre Pedine (non Dame).
3) Si possono mangiare solo pezzi dell'avversario (non i propri!)
4) Dame di entrambi i giocatori possono mangiare in qualsiasi direzione (alto/sinistra, alto/destra, basso/sinistra, basso/destra).
5) Le Dame possono mangiare sia Pedine che Dame.
6) Importante: per semplificare l'implementazione del gioco, **è consentito mangiare** (al massimo) **un solo pezzo per turno**. Non sono ammessi "salti" in diagonale tra molteplici celle che permettono di mangiare due o più pezzi avversari (in alcune varianti del gioco questo è concesso).
7) Non è obbligatorio mangiare un pezzo avversario se ci si trova in condizione di poterlo fare.
8) E' obbligatorio fare una mossa durante il proprio turno: una mancata mossa (scacchiera invariata) porta a perdere il gioco automaticamente.
9) Se si raggiunge la riga opposta con una Pedina, è obbligatorio promuoverla a Dama.

7	o		o		o		X	
6		o		o				o
5	o		o		o			
4				o				o
3					o		x	
2		x		x				x
1	x				x		x	
0		x		x		x		x
	0	1	2	3	4	5	6	7

Esempio: una Pedina di player1 ha raggiunto la riga opposta (cella azzurra) ed è quindi stata promossa a Dama. Da qui in poi, il pezzo sarà sempre una Dama.

2. Come si svolge una partita

I vostri codici verranno fatti giocare l'uno contro l'altro in un torneo. L'esame si può passare anche con una strategia "naive" (o completamente casuale), ma ai fini della valutazione finale vi conviene implementare una qualche strategia intelligente: ai primi classificati nel torneo verranno assegnati punti extra.

Voi dovete solo completare l'implementazione della **classe Player** (dettagli nelle prossime sezioni) contenuta nell'header player.hpp (lavorate all'interno di questo header e non createne altri): gli altri file sorgente servono solo a noi per lanciare il vostro codice e non vanno toccati. Non preoccupatevi troppo di come faremo gareggiare i vostri codici: voi dovete solo implementare correttamente la classe Player. Per completezza (può tornarvi utile per testare il vostro codice), in questa sezione vi accenniamo come avviene una partita:

1. Ad ogni istante della partita, tre eseguibili sono in esecuzione: due binari compilati da "play.cpp" con i vostri giocatori (player1 e player2 - il turno viene assegnato da noi in modo casuale), e il binario compilato da "verify.cpp" (il verificatore, contenente un'implementazione nostra corretta di Player).
2. Inizialmente, il verificatore crea una scacchiera iniziale e la salva in "board_1.txt"
3. Player1 carica "board_1.txt", esegue una mossa, e salva la scacchiera risultante in "board_2.txt"
4. Il verificatore carica board_2.txt e verifica che la mossa sia valida. Se non lo è, il giocatore viene squalificato. Se lo è, il gioco continua.
5. Player2 carica "board_2.txt", esegue una mossa, e salva la scacchiera risultante in "board_3.txt"
6. Il verificatore verifica che la mossa sia valida. Se non lo è, il giocatore viene squalificato. Se lo è, il gioco continua.
7. … e così via, fino alla fine della partita.

Warning: ogni tentativo di barare (per esempio salvando file extra in modo da bypassare il verificatore) porterà alla bocciatura dell'intero progetto. Ricordate che controlleremo i vostri codici anche a mano.

2.1 Termine partita

La partita termina in quattro casi:

1. Un giocatore ha terminato le pedine: l'altro giocatore è il **vincitore**.
2. Un giocatore ci mette troppo tempo ad eseguire una mossa: **timeout** (imposteremo un limite massimo di qualche decina di secondi per ogni mossa, ma una buona implementazione dovrebbe scegliere la mossa da eseguire in una frazione di secondo). Il giocatore in timeout perde la partita.
3. Un giocatore usa **troppa memoria RAM** (più di 100 MiB). Il giocatore che usa troppa memoria perde la partita.
4. **La partita dura troppo a lungo** (diciamo 100 mosse). In questo caso abbiamo un pareggio.

3. Classe Player (file player.hpp)

La classe implementa un giocatore "con memoria" (history). Ad ogni istante, la classe deve ricordare tutte le scacchiere viste in precedenza durante la partita. La decisione su come strutturare in memoria queste informazioni spetta a voi: l'importante è che la classe fornisca un'interfaccia corretta verso l'esterno tramite i metodi discussi qui sotto.

3.1 #include

Non potete importare niente con la macro **#include** (gli unici #include ammessi sono quelli già presenti nell'header file). Prima di compilare il vostro codice elimineremo tutti gli eventuali #include aggiunti da voi: se il codice non compila, il progetto non passa l'esame. Potete invece usare la keyword **using** per importare namespace o parti di essi.

3.2 Membri pubblici di classe

```
enum piece {x, o, X, O};
```

Questo enum è il tipo di dato che va immagazzinato all'interno delle scacchiere (player1: x/X, player2: o/O).

```
Player(int player_nr = 1);
```

Costruttore: il costruttore accetta in input il turno del giocatore. Di default il turno è 1. Se player_nr è diverso da 1 o 2, lanciare una **player_exception** con err_type uguale a index_out_of_bounds(e campo msg a scelta)

```
~Player();
```

Distruttore: attenzione a rilasciare correttamente le risorse allocate. Come avete capito, il vostro codice dovrà allocare dinamicamente della memoria quindi fate molta attenzione a questo aspetto (useremo valgrind per rilevare memory leaks).

```
Player(const Player&);
```

Copy constructor: tutti i dati del Player in input vanno copiati nell'oggetto costruito. Attenzione ad allocare correttamente la memoria.

```
Player& operator=(const Player&);
```

Copy assignment: tutti i dati del Player in input vanno copiati in this. Attenzione: non dimenticatevi di de-allocare la memoria di this prima di effettuare una nuova allocazione e copia.

```
piece operator()(int r, int c, int history_offset = 0) const;
```

Operatore di accesso alla memoria del giocatore. Questo operatore restituisce la pedina contenuta in riga r e colonna c della "history_offset"-esima scacchiera più recente. In altre parole, se history_offset=0 va restituita la pedina in coordinata (r,c) dell'ultima scacchiera vista; se history_offset=1 va restituita la pedina in coordinata (r,c) della penultima scacchiera vista, … Lanciare una **player_exception** con campo err_type uguale a index_out_of_bounds (msg a scelta) se le coordinate (r,c,history_offset) non esistono in memoria.

```
void load_board(const string& filename);
```

Caricamento da file (percorso "filename") di una scacchiera. Si veda sezione 3.3 per il formato del file. La scacchiera caricata deve essere salvata nella posizione più recente della history del Player (ossia, dopo il caricamento le sue pedine devono essere accessibili tramite operator()(r,c,0)). Se il file non esiste lanciare una **player_exception** con err_type uguale a missing_file. Se la scacchiera caricata non è valida (esempio: troppe pedine, pedine su celle bianche, ecc …), lanciare una **player_exception** con err_type uguale a invalid_board.

Suggerimento: dato che la history prevede di dover "appendere" nuove scacchiere, codificatela con una struttura opportuna! un array di scacchiere può avere senso, ma in questo caso dovreste implementare una strategia di doubling per non perdere troppo tempo in ri-allocazioni (https://stackoverflow.com/questions/26190789/why-is-it-common-practice-to-double-array-capacity-when-full). Una strategia alternativa più semplice è implementare una lista di scacchiere. Ricordate che non potete #includere <vector> o simili.

```
void store_board(const string& filename, int history_offset = 0) const;
```

Salvataggio di una scacchiera su file (al percorso “filename”). Deve venire salvata la “history-offset”-esima scacchiera più recente (ossia, se history-offset=0, la più recente; se history-offset=1 la penultima, e così via). Se history_offset è più lungo della history, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

```
void init_board(const string& filename) const;
```

Questa funzione deve: creare una nuova scacchiera (con i pezzi in posizione iniziale, vedi immagine in pagina 1), aggiungere la scacchiera nella posizione più recente della history del giocatore, e infine salvare su file la nuova scacchiera.

```
void move();
```

Questa è la funzione che implementa la vostra strategia. La funzione deve eseguire una mossa sulle vostre pedine (per il turno: vedi costruttore) e aggiungere la scacchiera risultante nella posizione più recente della history del giocatore. Le scacchiere già presenti precedentemente nella history non vanno modificate! Chiaramente, se la mossa prevede l’eliminazione di una pedina dell’avversario, questa pedina non deve essere presente nella scacchiera risultante (va eliminata): sta a voi implementare correttamente il cambio di stato della scacchiera in seguito alla mossa! (attenzione: noi verificheremo che la vostra classe effettui mosse valide). E’ possibile non effettuare alcuna mossa (ossia aggiungere una scacchiera identica alla precedente), ma questo porta alla perdita della partita. Ricordate che, se un giocatore raggiunge la riga opposta con un pezzo di tipo “Pedina”, questo pezzo va obbligatoriamente promosso a “Dama” (la mancata promozione è considerata una mossa non valida). Se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

Suggerimento: se siete incerti su quale mossa effettuare (o se non c’è una mossa che possa portare un evidente beneficio), usate la randomness (rand()), ossia scegliete un vostro pezzo a caso e muovetelo a caso (ovviamente in una cella valida). Questo ridurrà le possibilità che la partita entri in loop (ricordate che se la partita entra in loop, entrambi i giocatori sono squalificati dal torneo).

```
bool valid_move() const;
```

Questa funzione compara le due scacchiere più recenti nella history (l’ultima e la penultima) e decide se la mossa corrispondente (quella che ha trasformato la penultima scacchiera nell’ultima) è valida. Nota: la mossa potrebbe essere stata eseguita da uno qualsiasi dei due giocatori, non necessariamente da voi. Se ci sono meno di due scacchiere in history, va lanciata una **player_exception** con err_type uguale a index_out_of_bounds.

void pop();

La funzione rimuove dalla history la scacchiera più recente. Se la funzione viene chiamata su una history vuota, va lanciata una **player_exception** con err_type uguale a index_out_of_bounds.

bool wins(int player_nr) const;

Restituisce true se e solo se l'ultima scacchiera (la più recente) in history è vincente per il giocatore numero player_nr. Se player_nr è diverso da 1 o 2 oppure se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

bool wins() const;

Restituisce true se e solo se l'ultima scacchiera (la più recente) in history è vincente per il giocatore associato all'istanza corrente (this) di Player. Se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

bool loses(int player_nr) const;

Restituisce true se e solo se l'ultima scacchiera (la più recente) in history è perdente per il giocatore numero player_nr. Se player_nr è diverso da 1 o 2 oppure se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

bool loses() const;

Restituisce true se e solo se l'ultima scacchiera (la più recente) in history è perdente per il giocatore associato all'istanza corrente (this) di Player. Se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

int recurrence() const;

Restituire quante volte la scacchiera più recente compare nella history. Per esempio, se la history contiene le scacchiere A, B, C, B, D, B, E, B (dove A,B,C,D,E sono scacchiere distinte e la più recente è quella più a destra), allora la funzione deve restituire 4 perchè la scacchiera più recente (B) compare 4 volte nella history. Se la history è vuota, lanciare una **player_exception** con err_type uguale a index_out_of_bounds.

3.3 Membri privati di classe

Scegliete voi come memorizzare la history di scacchiere (e il turno del giocatore). L'importante è che:

- il salvataggio/caricamento su/da file avvenga rispettando il formato descritto nella prossima sezione,
- i metodi implementati forniscano un'interfaccia corretta sulla history di scacchiere (verificheremo con una nostra implementazione: i metodi di accesso alla history devono restituire gli stessi risultati di un'implementazione corretta)
- non usiate strutture esterne (esempio std::vector). Ogni #include aggiuntivo verrà eliminato dal codice. Gestite correttamente l'allocazione/deallocazione di memoria dinamica come visto a lezione.

3.4 Formato file scacchiera

La scacchiera va salvata/caricata in formato testuale (non ios::binary!). I pezzi vanno salvati usando le stringhe (senza virgolette) "x", "X", "o", "O" (pedina/dama di player 1/2). Ogni cella della scacchiera contiene una di queste stringhe oppure è vuota (se la cella non contiene alcuna pedina). Nel formato, ogni cella è separata dalla successiva a destra da uno spazio. Celle vuote sono codificate con uno spazio. Una newline segnala il passaggio alla riga successiva.

Esempio: la seguente scacchiera

7	o		o		o		X	
6		o		o				o
5	o		o		o			
4				o				o
3					o		x	
2		x		x				x
1	x				x		x	
0		x		x		x		x
	0	1	2	3	4	5	6	7

Va codificata come segue (nessuna newline dopo l'ultima riga):

```
o o o X
 o o   o
o o o
   o   o
    o x
 x x   x
x   x x
 x x x x
```

4. Valutazione progetti

Nella sessione di giugno i vostri progetti verranno testati in un torneo. I primi classificati (leggi sotto) otterranno punti aggiuntivi. Nelle sessioni successive, non verrà effettuato alcun torneo e quindi non ci sarà la possibilità di ottenere questi punti extra.

- Codice che non compila non passa l'esame (attenzione: ricordate che non potete #includere niente. Se aggiungete un #include, questo verrà eliminato e quindi probabilmente il vostro codice non compilerà).
- Lanceremo un tool antiplagio automatico: progetti copiati non passano l'esame (tutti i progetti coinvolti).
- Per il resto, il punteggio terrà conto dei seguenti fattori: memory test (valgrind), correttezza dei metodi implementati, strategia di gioco non banale, assenza di timeout / utilizzo di troppa memoria.
- Un bonus (punti aggiuntivi o lode) verrà assegnata se nella gara il codice si classifica nel primo 15%.