UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Motori scacchistici in linguaggio C: rappresentazione della scacchiera, algoritmi di ricerca e funzioni di valutazione

Relatore: Prof. Antonio Giunta

Laureando: Lorenzo Carisi

ANNO ACCADEMICO 2021 – 2022
Data di laurea: 21/09/2022

SOMMARIO

1
FINALITÀ DI QUESTO LAVORO

1.1
L'informatica nel mondo degli scacchi

1.1.1 Cenni storici sul gioco

L'invenzione del gioco degli scacchi viene comunemente fatta risalire in India intorno al VI secolo. Per oltre quindici secoli è continuato ad evolversi e ad accompagnare le correnti letterarie e filosofiche dell'uomo. Ne è un esempio il periodo romantico dove i giocatori cercavano di esprimere la loro soggettività, attraverso l'utilizzo di sacrifici spettacolari che creassero più stupore possibile tra il pubblico; è proprio in questo periodo che si è soliti far coincidere la nascita del gioco degli scacchi moderno. Nel 1924 infatti, si viene a creare la FIDE, dal termine francese per, Federazione Internazionale Degli Scacchi e viene organizzato il primo campionato del mondo. Gli scacchi si liberano quindi della loro reputazione di gioco d'azzardo.

1.1.2 Avvento dell'informatica

Negli ultimi decenni, con l'avvento dell'informatica il gioco degli scacchi è diventato anche un terreno fertile per i programmatori che hanno cercato di ideare quello che oggi prende il nome di motore scacchistico, ovverosia un programma che potesse emulare un giocatore, mantenendo però la capacità di calcolo tipica di un calcolatore; iniziò così una lunga sfida tra la macchina e l'uomo che si concluse definitivamente nel 1997 quando, l'allora campione del mondo Garry Kasparov venne lasciato impotente da Deep Blue, il motore ideato dall'IBM.

1.1.3 Età contemporanea

Da allora i motori scacchistici sono diventati strumenti essenziali nella preparazione di un giocatore professionista e vengono giornalmente impiegati per trovare nuove idee nelle aperture e controllare l'accuratezza delle mosse giocate nei tornei nell'ambito del mediogioco e dei finali. Il predominio dei motori scacchistici e in particolare, di Stockfish, il motore open source più potente del momento, sembrava incontrastabile, fino al 2017 quando l'azienda Google mise a punto AlphaZero, un nuovo programma basato sull'intelligenza artificiale che mostrò al mondo intero come in realtà anche i motori scacchistici, non sono perfetti.

1.1.4 Il gioco

Dopotutto, pensare di poter ideare una macchina che sia in grado di completare il gioco, ovverosia salvare in un database ogni singola variante ottenibile giocando, è impossibile. Si stima infatti che il numero di mosse ottenibili sia nell'ordine di 10^{120}, più di un googol, il numero che dà il nome al famoso motore di ricerca, ma soprattutto molto più della stima del numero di atomi presenti nell'universo osservabile; pertanto servirebbe una memoria di dimensioni irraggiungibili. Proprio per questo si sono tentati altri approcci rispetto al mero calcolo di tutte le varianti e ad oggi i motori continuano ancora a migliorare, evidenziando come l'informatica stia progredendo sempre più.

1.1.5 Finalità del progetto

Tutta questa storia porta a domandarsi cosa ci sia dietro ad un motore scacchistico e come riesca a superare i più grandi giocatori della storia. Ebbene, questa tesi di laurea si pone l'ambizioso obiettivo di mostrare alcuni degli algoritmi che portano alla realizzazione di questo programma. In particolare l'attenzione viene posta sulla rappresentazione della scacchiera e la realizzazione delle mosse e delle regole in lingugaggio C, "Un linguaggio che è stato creato per unire i vantaggi della programmazione a basso a livello con quelli della programmazione ad al-

to livello", citando le parole del mio relatore, il professor Antonio Giunta.

1.1.6 Considerazioni

La programmazione del motore scacchistico ha portato ad approfondire molti concetti visti nel corso di laurea, come la manipolazione di stringhe, l'utilizzo di strutture dati per progettare algoritmi più efficienti (come, ad es., matrici, sia rettangolari che non, e array di puntatori). Anche dal punto di vista algoritmico si sono sfruttati paradigmi visti a lezione, come la programmazione dinamica e il paradigma Greedy. A titolo di curiosità il progetto iniziale prevedeva di sfruttare la rappresentazione binaria di numeri a 64 bit, in grado cioè di contenere una cella di memoria per ciascuna delle 64 caselle di gioco, e sfruttare operazioni a livello di singolo bit; ma nell'ambiente che si è utilizzato, il compilatore C90 possiede soltanto il tipo di dato $long che è un intero a 32 bit e ciò comportava in fase di esecuzione un errore logico che causava l'interruzione del programma stesso a causa dello sfondamento dell'intero. Per far fronte a tale inconveniente si è deciso di sostituire tali interi con degli array a 64 celle, che hanno eliminato il problema a discapito di un maggiore uso della memoria. Un'alternativa sarebbe potuta essere quella di creare un tipo strutturato $longlong come unione di due interi a 32 bit e implementare delle funzioni alternative che permettessero di eseguire le stesse operazioni sui bit che sono presenti per i tipi di dati primitivi.

In conclusione, al di là delle scelte progettuali effettuate, ritengo che il lavoro di tesi sia stato decisamente sfidante e per certi versi anche inaspettato, dietro un motore scacchistico risiede una solida base di programmazione che unisce in maniera armonica i concetti dell'informatica e fa apparire il gioco degli scacchi sotto una nuova luce che, sebbene non sia spettacolare come gli scacchi romantici dell'Ottocento, è altrettanto magnifica.

2
EVENTUALI LIMITI

Il programma che ho creato non emula il gioco degli scacchi nella sua interezza, ho volutamente eliminato due regole che conducono alla patta, ovvero la regola per la quale se una stessa posizione occorre per tre volte in una partita, la partita è da considerarsi finita, così come se non avvengono per 50 turni, ovvero una mossa per ciascun giocatore, catture di pezzi o spinte di pedone.

Il motore scacchistico che ho implementato, sebbene sappia le regole e dunque giochi esclusivamente mosse legali, non sa valutare quale mossa sia la migliore, di conseguenza tra le mosse legali, ne sceglie una in maniera linearmente casuale, il che lo rende molto facile da battere. Si sarebbe potuto scrivere qualche algoritmo ulteriore che fosse in grado di cercare le mosse candidate, ovvero le mosse che sembrano più forti e un algoritmo di valutazione per tali mosse.

La scacchiera non fa uso di librerie grafiche; per questo risulta un po' difficoltoso giocare; viene rappresentata usando solo lettere specifiche per i vari pezzi e il maiuscolo/minuscolo per distinguere il bianco dal nero.

Anche le mosse non si possono effettuare spostando manualmente i pezzi disegnati sulla scacchiera, ma devono essere inserite attraverso la scrittura delle caselle di partenza e arrivo del pezzo, risultando abbastanza scomodo per giocare; ma tutto questo esula dal progetto che ho realizzato.

3
ALGORITMI E STRUTTURE DI DATI PIÙ SIGNIFICATIVI

3.1
Algoritmi

3.1.1 Rappresentazione binaria dei pezzi:

Un primo cenno particolare va fatto per la rappresentazione dei pezzi. Essi sono stati trasformati in numeri che garantissero la possibilità di scoprire in maniera rapida quale fosse la loro tipologia e a quale colore appartenessero, sfruttando la matematica modulare. L'idea è stata di pensare agli interi con la loro rappresentazione in bit e sfruttare la conversione decimale/binario. I valori sono stati: 0 per l'assenza di pezzi, 1 per il re, 2, per il pedone e 3 per il cavallo. Ai pezzi che si muovono lungo le direzioni diagonali e ortogonali come alfieri, torri e regine sono stati assegnati i valori 5, 6 e 7, così che venissero identificati dal bit in posizione due, ovvero il 4 in decimale. Per i colori sono stati scelti l'8 e il 16, cosìcché la presenza del bit in posizione tre identificasse un pezzo bianco e il quarto bit un pezzo nero. In tal modo un pezzo bianco risulta essere la semplice somma dei valori del colore e della tipologia di pezzo. Ad esempio un alfiere nero avrà valore 5+16=21 e per scoprire il suo colore basterà sfruttare i bit posti a 1 nella sua rappresentazione binaria.

3.1.2 Notazione FEN:

Il primo vero e proprio algoritmo riguarda invece la notazione FEN per impostare la posizione di partenza in una partita. Si è scelto di implementare questa funzionalità per rendere più rigoroso il codice, ma soprattutto per la fase di debug, dove una volta scritto il programma che calcola le mosse legali, era necessario verificare se se ne erano perse alcune o se ne erano aggiunte alcune che invece erano di fatto illegali. Un esempio di codice FEN è il seguente:

```
r1b1k1nr/p2p1pNp/n2B4/1p1NP2P/6P1/3P1Q2/P1P1K3/q5b1
```

che rappresenta questa posizione sulla scacchiera:

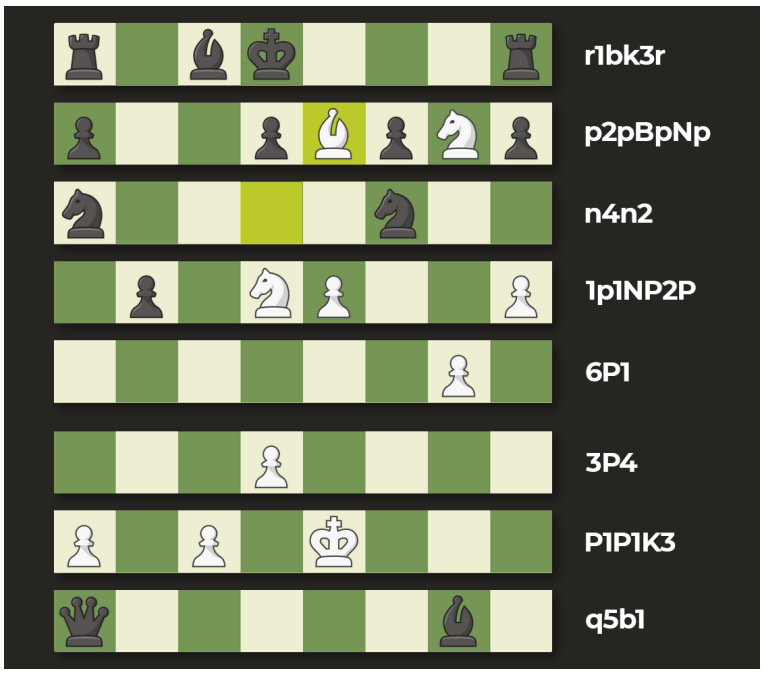


Figura 1: fonte Chess.com

dove per i pezzi si sono usate le lettere maiuscole per il bianco e minuscole per il nero; a ciuscuna tipologia è stata assegnata una lettera, in particolare 'R' (Rook) per la torre, 'N' (kNight)

per il cavallo, ‘B’ (Bishop) per l’alfiere, ‘P’ (Pawn) per il pedone, ‘Q’ (Queen) per la regina e ‘K’ (King) per il re. Le caselle vuote vengono interpretate con un 1 e se ce ne sono di consecutive si sommano i loro valori. Inoltre ogni ‘/’ indica la fine di riga e si parte a leggere dalla casella nell’ultima riga e nella prima colonna, spostandosi a scendere. La notazione FEN non si ferma qui; contiene anche altri valori, separati da uno spazio, che seguono quelli appena descritti, che sono una lettera ‘w’ o ‘b’ per indicare se tocca al bianco (white) o al nero (black), seguono ancora le lettere “KQkq-” a seconda che sia possibile effettuare l’arrocco corto (Kingside) o lungo (Queenside) per il bianco e nero usando la convenzione di prima per distinguere tra i colori e il trattino ‘-‘ per indicare che nessun arrocco è possibile. Infine ci sono scritti una casella, se è possibile effettuare una presa al passo (en passant) e due numeri per il numero di mosse effettuate ed il contatore per la regola delle 50 mosse, che però si sono omessi. L’implementazione di un algoritmo in grado di convertire il FEN in una posizione sulla scacchiera e viceversa permette di vedere in azione alcune manipolazioni di stringhe attraverso strtok, strchr e strlen, e anche di vedere dal punto di vista della progettazione l’ispezione della scacchiera per righe, argomento trattato dal punto di vista teorico nel corso di “Algoritmi per l’Ingegneria” nella programmazione dinamica.

3.1.3 Esecuzione di una mossa e rimozione della mossa eseguita:

La prima funzione che è stata sviluppata nell’algoritmo di esecuzione della mossa è stata quella che elimina il pezzo nel caso se ne fosse catturato uno, facendo attenzione ad escludere il caso di presa al passo che invece prevedeva l’eliminazione del pedone non dalla casella di arrivo. Poi è venuto il turno del movimento del re, con l’aggiornamento dei dati relativi ai diritti di arrocco che venivano persi. A seguire, invece, i movimenti di tutti gli altri pezzi che sono avvenuti aggiornando le PieceList. Infine ci si è occupati dei casi di mosse speciali; prima, il caso della promozione, con l’eliminazione del pedone che raggiunge l’ottava traversa (riga), come si chiama nel gergo scacchistico, e l’aggiunta del nuovo pezzo promosso sulla scacchiera; poi, il caso della presa al passo con lo spostamento del pedone in diagonale e l’eliminazione del pedone nella casella ortogonale alla cattura attraverso un salto di 8 caselle per il nero e -8 caselle per il bianco; ad esempio, se il nero cattura alla casella 16, il pedone da eliminare si trova alla 16 + 8 = 24; il caso dell’arrocco è diverso a seconda che sia corto o lungo, ma lavorando con gli indici delle caselle si poteva riuscire ad eliminare la torre dalla casella iniziale e porla in quella finale e allo stesso tempo spostare il re.

L’algoritmo di rimozione della mossa eseguita prevede invece di prendere le informazioni del Gamestate e sfruttarle per ripercorrere al contrario quanto appena fatto con l’esecuzione, con ancora una volta i casi più ostici da eseguire, la promozione, la presa al passo e l’arrocco, che, oltre a ripristinare i pezzi, necessitavano di modificare anche gli stati delle mosse speciali stesse.

3.1.4 Creazione delle informazioni sul movimento dei pezzi:

Nella stesura delle funzioni per l’inizializzazione delle informazioni contenute nella struttura dati $PrecomputedMoveData si sono dovuti implementare diversi algoritmi per la creazione delle mosse dei pezzi. Notevole è ad esempio il caso del cavallo, la cui idea si ripete poi anche per gli altri pezzi; la mossa viene identificata mediante un offset rispetto alla posizione di partenza; dopo aver aggiunto l’offset alla posizione, però, bisogna fare attenzione, perché il pezzo potrebbe essere finito fuori dalla scacchiera, o, peggio ancora si sia “teletrasportato” da una parte opposta della scacchiera Bisogna dunque evitare queste situazioni aggiungendo delle condizioni sull’offset generato, come ad esempio che la distanza rispetto alla casella di partenza lungo le direzioni ortogonali (nord, sud, est e ovest) non superi il 2, condizione necessaria affinché una mossa sia legale.

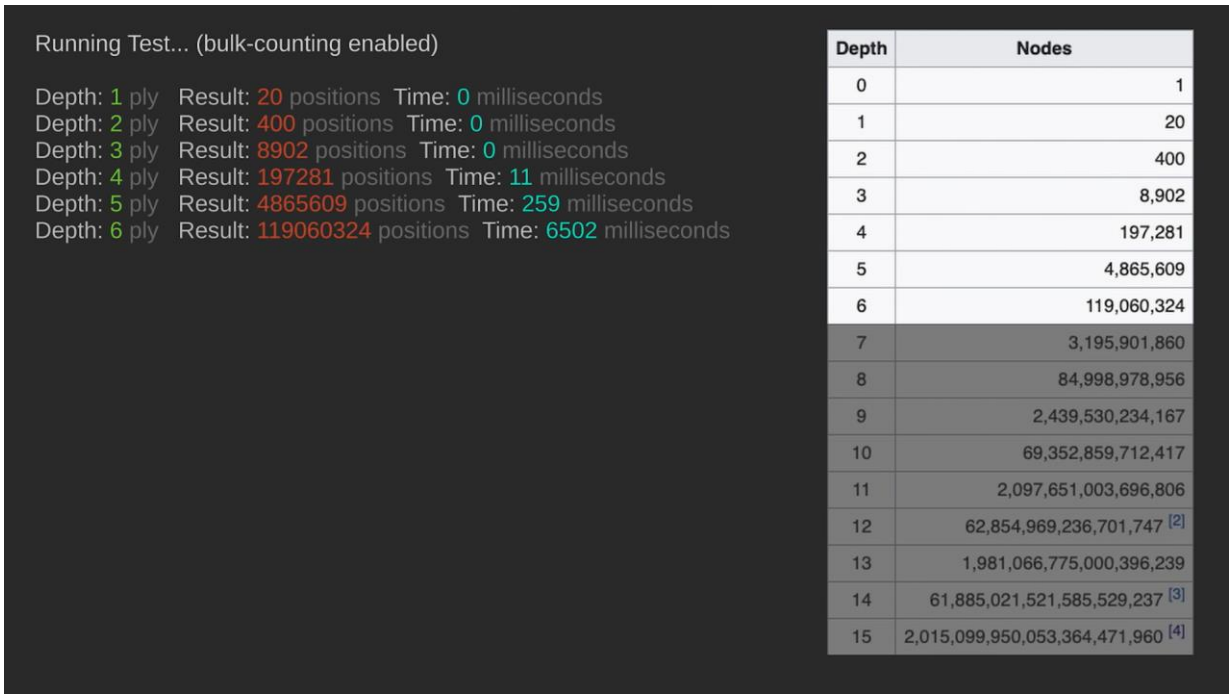
3.1.5 Generazione delle mosse legali:

La generazione delle mosse legali data una posizione è stato l’algoritmo più sfidante dell’intero

progetto. L'idea è stata di generare le mosse pezzo per pezzo, e solo alla fine unirle insieme in una lista concatenata. Le prime mosse generate sono state quelle di re, perché nella situazione in cui si è in uno scacco doppio, ovverosia il proprio re è attaccato da due pezzi avversari, le uniche mosse a disposizione del giocatore sono proprio quelle di re. In questa sede è necessario fare due note. La prima è su come si è gestita la creazione degli array "precompilato" che contengono le mosse possibili dai vari pezzi data una posizione che sono descritti nel seguito. Gli array sono stati generati a partire da una lista in fase di esecuzione; per questo era necessario salvare in qualche cella di memoria la grandezza dell'array. Per farlo, la prima cella di memoria di ciascun array contiene un valore che non è una mossa bensì la grandezza dell'array stesso. Fatta questa premessa, è possibile proseguire con la descrizione dell'algoritmo. La seconda è sul motivo per cui si è fatto uso del costrutto continue nella generazione delle mosse di re e poi anche degli altri pezzi, sebbene il suo utilizzo si sarebbe potuto evitare con un costrutto if/else. La scelta di utilizzarlo è giustificata dal fatto che le funzioni che si stanno descrivendo sono già di per sé molto ricche di condizioni; per non appesantire ulteriormente il codice con queste condizioni di skip, si è preferito apporre un continue, opportunamente commentato, che consente di saltare all'iterazione successiva grazie a "condizioni scorciatoia" che si verificano di tanto in tanto. Ad esempio, nel caso delle mosse di re appare chiaro che se la mossa che si sta considerando va a catturare un pezzo amico (cioè un altro pezzo che appartiene al giocatore), allora si può concludere immediatamente che la mossa è illegale e non può essere effettuata.

Un algoritmo particolare, tra la molteplicità presente in questa parte del progetto, è quello che controlla la legalità delle mosse di presa al passo. Per riuscire ad identificare se tale mossa fosse possibile, l'idea che si è adottata è stata, diversamente dagli altri casi, quella di effettuare in un primo momento la mossa sulla scacchiera, studiarne le colonne e diagonali aperte alla ricerca di un attacco di scoperta sul re avversario nella posizione risultante e solo nel caso in cui tale ricerca non trovasse nessun attacco considerare legale la mossa.

Per testare la correttezza delle mosse generate si è chiamata ricorsivamente la funzione incaricata di generare le mosse, in maniera che da ciascuna mossa effettuata si calcolassero poi tutte le possibili risposte successive e attraverso un contatore si teneva traccia del numero di varianti che si venivano a generare globalmente; infine, si confrontava il numero ottenuto con quello degli altri programmatori che hanno pubblicato il loro lavoro sul web. Nel caso della posizione iniziale si è ottenuto questo risultato:



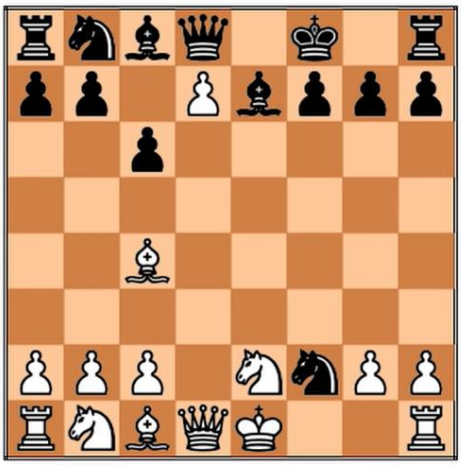
Depth	Nodes
0	1
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417
11	2,097,651,003,696,806
12	62,854,969,236,701,747 [2]
13	1,981,066,775,000,396,239
14	61,885,021,521,585,529,237 [3]
15	2,015,099,950,053,364,471,960 [4]

Figura 2 fonte Coding Adventure: Chess AI - Youtube

Sfruttando il codice FEN però si è potuto considerare anche altre posizioni, tra cui alcune proposte da Wikipedia che avevano lo scopo di effettuare bug detecting per i programmatori, qui di seguito si può vedere la posizione e le insidie che il programma doveva essere in grado di gestire:

Position 5

This position was discussed on Talkchess [7] and caught bugs in engines several years old at depth 3 [8] and was also reported wrong here [9], hopefully now corrected with the results given by Steven Edwards, July 18, 2015 [10]



Depth	Nodes
1	44
2	1,486
3	62,379
4	2,103,487
5	89,941,194

```
rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8
```

A causa di qualche bug nel programma, il numero di nodi a partire dalla profondità 3 non corrispondeva; per questo si è fatto uso di Stockfish, si è effettuata la stessa procedura e si è arrivati a scoprire in quale posizione avvenivano gli errori. Gli errori principali riguardavano l'arrocco; ad esempio, se un pezzo avversario cattura la torre del giocatore, l'arrocco lungo quel lato non dovrebbe più essere disponibile e vale come se la torre si fosse mossa. Allo stesso modo si sono venute a scoprire le problematiche riguardanti la presa al passo che scopriva scacchi sul

proprio re, e per questo si è arrivati all'algoritmo descritto sopra.

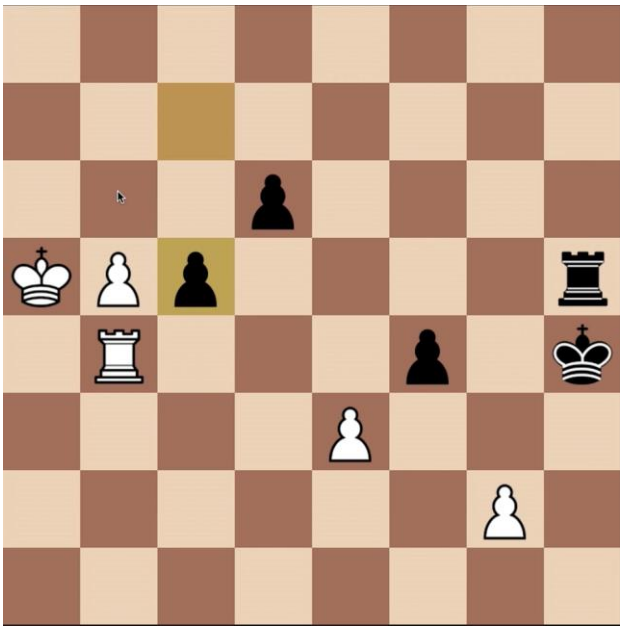


Figura 3 Il caso en passant

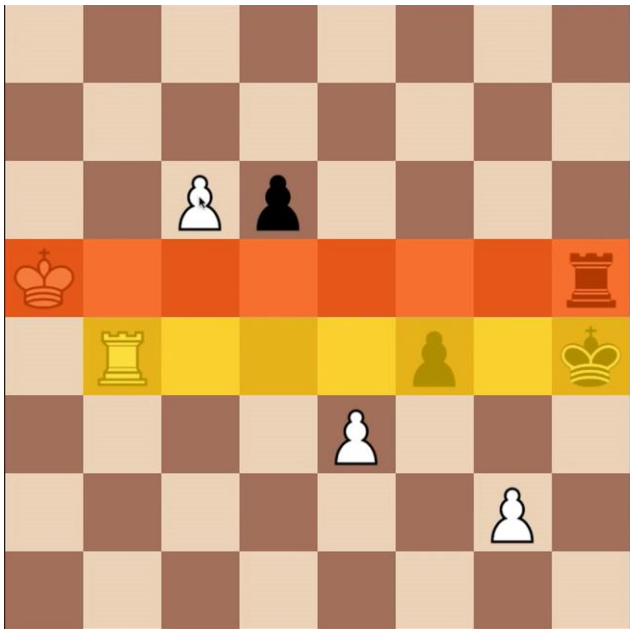


Figura 4 Si scopre un attacco sul re

Tutte le immagini e l'idea stessa per il debug fanno riferimento al video: " Coding Adventure: Chess AI" presente sulla piattaforma Youtube.

3.2
Strutture di dati

3.2.1 Coord:

La prima struttura dati di interesse è stata $Coord, la cui funzione è stata quella di rappresentare una casella della scacchiera attraverso due campi: l'indice di riga e l'indice di colonna. Il compilatore però funziona meglio attraverso la rappresentazione di una casella come la cella di un array lineare di 64 interi; per questo è stato necessario sviluppare delle funzioni che permettessero di passare da una notazione all'altra, al fine di risparmiare spazio in memoria e/o migliorare le prestazioni del programma.

3.2.2 Move:

Per identificare le mosse di un giocatore è stata adottata una struttura dati $Move in cui sono salvati soltanto la posizione di partenza e di arrivo del pezzo, più un eventuale valore che rappresentasse le mosse speciali, perché non è necessario conoscere il pezzo che compie la mossa per poterla creare, bensì è necessario più avanti per il controllo della legalità della mossa effettuata.

3.2.3 Gamestate:

Per far fronte alle mosse speciali che accadono in una partita, come l'arrocco, la presa al passo e la cattura di pezzi avversari si è fatta la scelta di identificare ciascuna mossa con un intero all'interno di una struttura $Gamestate; ho sfruttato lo stesso concetto utilizzato per i pezzi per scegliere i valori di ciascuna mossa speciale.

3.2.4 Pile e Liste Concatenate:

Al fine di migliorare le performance di calcolo del motore si è fatto uso di strutture dati di appoggio; in particolare ho usato una pila (stack) per tener conto dei valori contenuti in $Gamestate delle mosse precedenti; infatti il programma, oltre a effettuare una mossa sulla scacchiera, è anche in grado di ripristinare la posizione precedente alla mossa stessa; per fare questo, la pi-

la era la struttura dati che meglio funzionava. Inoltre nel momento della generazione delle mosse, non sapendo a priori quanti sarebbero potute essere, si è fatta la scelta di usare delle strutture dati lista (linked list); questa implementazione torna molto utile quando un motore deve calcolare molto avanti nel futuro e il numero di mosse che deve considerare diventa anche considerevolmente alto, sebbene non si sia arrivati a completare la parte di ricerca della mossa. È bene far presente che l'implementazione scelta per le liste non è stata la migliore; infatti, l'aggiunta di nuovi elementi viene messa in coda alla lista, che quindi deve essere scorsa tutta ogni volta; questa soluzione, sebbene sia temporalmente più leggibile, riduce le prestazioni di calcolo.

3.2.5 Board:

La scacchiera con i pezzi è stata la prima grande struttura dati scritta nel progetto. La sua funzione era quella di permettere di rappresentare la partita sul calcolatore, senza preoccuparsi delle regole, ma facendo possibile effettuare tutte le mosse. È bene sottolineare che oltre alla rappresentazione per array lineare, si sono introdotte delle strutture dati $PieceList che avevano lo scopo di semplificare il modo in cui era possibile accedere ai pezzi; talvolta si conosceva la loro posizione sulla scacchiera e dunque sarebbe bastato solo l'array iniziale, ma nella generazione delle mosse si dovevano prendere uno ad uno tutti i vari pezzi e calcolare le mosse che potevano effettuare; così facendo non era più necessario fare una ricerca lineare sull'array per trovare i pezzi, ma bastava selezionarli attraverso tale struttura dati, che per l'appunto contiene un array che dice quali sono le caselle occupate dai vari pezzi. C'è poi anche la pila che, come detto in precedenza, salva le informazioni per tornare indietro nel tempo alle mosse precedenti e c'è un array per salvare la posizione del re, il cui ruolo è cruciale nel gioco e serve saperne la posizione ad ogni mossa per evitare la generazione di mosse illegali dovute ad inchiodature, ovverosia pezzi bloccati, che se si muovessero esporrebbero il re ad un attacco diretto dai pezzi dell'avversario e scacchi diretti al re.

3.2.6 PreComputedMoveData:

Con la struttura dati PreComputedMoveData ci si avvicina alla struttura dati finale che collega tutto il progetto. Lo scopo è quello di salvare in memoria alcune informazioni utili su come si possono muovere i pezzi, per poterle utilizzare quando ce n'è bisogno. In particolare si trova un array contenente le 8 direzioni di movimento che vengono rappresentate come dei salti (offset) da una casella ad un'altra, così che per muoversi verso nord ci si deve muovere di 8 celle di memoria avanti nell'array che rappresenta la scacchiera, verso nord/est di 9 celle e così via.

Si trova poi una matrice 64×8 che, data una delle 64 caselle e data una delle 8 direzioni, contiene il numero di caselle che ci sono in quella direzione prima di raggiungere il bordo della scacchiera. Ci sono due array di 64 puntatori che puntano ad altrettanti array contenenti le mosse effettuabili dal re o dal cavallo data l'indice della casella di riferimento, che formano due matrici non rettangolari e così lo stesso anche per i pedoni, bianchi, e neri. Ci sono poi delle matrici definite BitBoard per ciascuno dei pezzi, che hanno la prima dimensione che identifica la casella sulla scacchiera e la seconda contiene il valore 1 in tutte le posizioni sulle quali può saltare il pezzo a cui appartiene la BitBoard. La differenza tra l'array di puntatori e le bitboard è che mentre nella prima tipologia vengono salvate le caselle di destinazione come intero da 0 a 63, nelle seconde ci sono già tutte e 64 le posizioni e quelle sulle quali si può atterrare contengono il valore 1; il vantaggio di avere entrambe le rappresentazioni si rivela nel momento in cui si tenta di eseguire le mosse, nel qual caso è necessario utilizzare la prima tipologia, a differenza di quando si deve controllare se una casella è attaccata dal pezzo nemico, nel qual caso si fa uso della seconda per la verifica, perché la prima richiederebbe una ricerca lineare tra tutte le possibilità. Un'altra informazione presente nella struttura dati è l'array che permette di verificare quale sia la direzione che collega due caselle, definito come $directionLookup ed infine delle informazioni sulla distanza del re e delle torri da una posizione all'altra della scacchiera, che però sarebbero servite per il calcolo della valutazione della mossa che non è stato poi im-

plementato.

3.2.7 MoveGenerator:

Il culmine del progetto si raggiunge con la struttura $MoveGenerator, o meglio con gli algoritmi che la gestiscono, ma anche la struttura stessa contiene alcune interessanti caratteristiche. La generazione delle mosse veniva resa difficile soprattutto dall'esistenza negli scacchi delle inchiodature, che causano il blocco dei movimenti di un pezzo, a causa di una direzione che se scoperta permetterebbe la cattura del re, mossa considerata illegale negli scacchi. Per tener conto di questa particolarità è stato dunque necessario mappare la scacchiera alla ricerca di queste inchiodature e linee di scacco diretto al re. Per farlo è stato necessario introdurre in $MoveGenerator degli array che si occupassero di mantenere le informazioni su quali caselle sono attaccate dai pezzi avversari e quali direzioni creano le inchiodature.

4
INPUT E OUTPUT DEL PROBLEMA

4.1
Input

Gli input che necessita il problema in esame, sono:

1. La posizione di partenza da sistemare sulla scacchiera, sottoforma di notazione FEN, Forsyth-Edwards Notation, che è lo standard utilizzato da tutti i motori scacchistici.
2. La notazione per le mosse da effettuare, che il programma riconosce nel formato casella_di_partenza-casella_di_arrivo-valore, dove il campo valore identifica attraverso un numero intero la tipologia di mossa che viene eseguita, se essa è speciale.

4.2
Output

Gli output del problema in esame risultano essere:

1. La scacchiera di gioco, che viene disegnata nel terminale usando solo simboli della tastiera.
2. Le mosse del motore scacchistico che vengono disegnate sulla scacchiera e indicate a schermo con la stessa notazione con cui vanno inserite.
3. Le indicazioni del termine della partita, come la fine della partita a causa di uno stallo o di scacco matto o l'utilizzo di una mossa illegale.

5
CODICI SORGENTI

Il file Coord.h contiene la struttura dati per la rappresentazione delle caselle della scacchiera.

```
#include <stdbool.h>

/* Rappresenta una casella della scacchiera come coordinata colonna-riga */
typedef struct {
    int fileIndex;
    int rankIndex;
} Coord;

bool IsLightSquare(Coord*);

int CompareTo(Coord*, Coord*);
```

Il file Coord.c contiene le funzioni per la rappresentazione delle caselle della scacchiera.

```
#include "Coord.h"

/*
    IP coord coordinata che rappresenta una casella della scacchiera
    OR true se $coord e' una casella bianca, false se e' una casella nera
*/
bool IsLightSquare (Coord* coord) {
      return (coord->fileIndex + coord->rankIndex) % 2 != 0;
} /* IsLightSquare */

/*
    IP coord coordinata che rappresenta una casella della scacchiera
    IP other coordinata che rappresenta una casella della scacchiera
    OR 0 se le coordinate rappresentano la stessa casella 1 altrimenti
*/
int CompareTo(Coord* coord, Coord* other) {
      return (coord->fileIndex == other->fileIndex && coord->rankIndex ==
other->rankIndex) ? 0 : 1;
} /* CompareTo */
```

Il file BoardRepresentation.h contiene la definizione dei dati per la rappresentazione della scacchiera, come l'indicizzazione delle caselle e la proprietà delle caselle di essere bianche o nere.

```
#include "Coord.h"

/* Le caselle della prima e dell'ultima riga sono le piu' utilizzate e' co-
modo
// dunque definirle come macro in modo da poterle utilizzare in modo piu'
leggibile
*/
#define a1 0
#define b1 1
#define c1 2
#define d1 3
#define e1 4
#define f1 5
#define g1 6
#define h1 7

#define a8 56
#define b8 57
#define c8 58
#define d8 59
#define e8 60
#define f8 61
#define g8 62
#define h8 63

/* Questa stringa e' utile per estrarre dall'indice di colonna la
// lettera che gli corrisponde, nei diversi file del motore
*/
#define fileNames "abcdefgh"

int RankIndex(int);

int FileIndex(int);

int IndexFromTwoCoord(int, int);

int IndexFromCoord(Coord*);

void CoordFromIndex(int, Coord*);

bool LightSquare (int, int);

void SquareNameFromTwoCoordinate (int, int, char*);

void SquareNameFromCoordinate(Coord*, char*);

void SquareNameFromIndex (int, char*);
```

Il file BoardRepresentation.c contiene le funzioni per la rappresentazione della scacchiera, come l'indicizzazione delle caselle e la proprietà delle caselle di essere bianche o nere.

```
#include <string.h>
#include "BoardRepresentation.h"

/* Da queste stringhe estraggo il nome della riga
   o colonna di cui ho bisogno*/
const char fileNamesString[9] = fileNames;
const char rankNames[9] = "12345678";

/*
// IP casella rappresentata come un intero da 0 a 63
// OR riga a cui appartiene tale casella
// la riga a cui appartiene la si trova prendendo il quoziente della divisione intera per 8,
// es 12 corrisponde alla riga 12 / 8 = 1 ovvero riga 2 come si vede dall'indice 2 della stringa delle righe
*/
int RankIndex(int squareIndex) {
      return squareIndex / 8;
} /* RankIndex */

/*
// IP casella rappresentata come un intero da 0 a 63
// OR colonna a cui appartiene tale casella
*/
int FileIndex(int squareIndex) {
      return squareIndex % 8;
} /* FileIndex */

/*
// IP fileIndex indice della colonna
// IP rankIndex indice della riga
// OR intero da 0 a 63 che rappresenta la casella, partendo da a1 e terminando in h8
*/
int IndexFromTwoCoord(int fileIndex, int rankIndex) {
      return rankIndex * 8 + fileIndex;
} /* IndexFromTwoCoord */

/*
// IP coord casella rappresentata nel formato colonna/riga
// OR intero da 0 a 63 che rappresenta la casella, partendo da a1 e terminando in h8
*/
int IndexFromCoord(Coord* coord) {
      return IndexFromTwoCoord(coord->fileIndex, coord->rankIndex);
} /* IndexFromCoord */

/*
// IP squareIndex intero da 0 a 63 che rappresenta la casella, partendo da
```

```
a1 e terminando in h8
// IOP coord struct Coord che conterra' la rappresentazione della casella
nel formato colonna/riga
*/
void CoordFromIndex(int squareIndex, Coord* coord) {
    coord->fileIndex = FileIndex(squareIndex);
    coord->rankIndex = RankIndex (squareIndex);
} /* CoordFromIndex */

/*
// IP fileIndex colonna a cui appartiene la casella di interesse
// IP rankIndex riga a cui appartiene la casella di interesse
// OR true se la casella e' bianca false se e' nera
*/
bool LightSquare(int fileIndex, int rankIndex) {
      return (fileIndex + rankIndex) % 2 != 0;
} /* LightSquare */

/*
// IP fileIndex colonna a cui appartiene la casella di interesse
// IP rankIndex riga a cui appartiene la casella di interesse
// IOP string stringa che conterra' il nome della casella nel formato: let-
tera per colonna numero per riga
*/
void SquareNameFromTwoCoordinate (int fileIndex, int rankIndex, char*
string) {
      string[0] = fileNamesString[fileIndex];
    string[1] = rankNames[rankIndex];
    string[2] = '\0';
} /* SquareNameFromTwoCoordinate */

/*
// IP coord coordinata che rappresenta la casella
// IOP string stringa che conterra' il nome della casella
*/
void SquareNameFromCoordinate (Coord* coord, char* string) {
      SquareNameFromTwoCoordinate    (coord->fileIndex,    coord->rankIndex,
string);
} /* SquareNameFromCoordinate */

/*
// IP squareIndex intero da 0 a 63 che rappresenta la casella, partendo da
a1 e terminando in h8
// IOP string stringa che conterra' il nome della casella
*/
void SquareNameFromIndex (int squareIndex, char* string) {
      Coord coord;
    CoordFromIndex(squareIndex, &coord);
    SquareNameFromCoordinate (&coord,string);
} /* SquareNameFromIndex */
```

Il file Piece.h contiene le informazioni su come sono stati rappresentati i pezzi per la scacchiera, attraverso cioè, numeri interi che soddisfavano requisiti utili al calcolo della loro tipologia e colore.

```
#include <stdbool.h>

/* La scelta di queste costanti per i vari pezzi
   e' utile a facilitare la scrittura di funzioni che
   servono ad identificare di che pezzo si tratta
   ad esempio il re bianco avra' valore 3+8=10
   l'idea viene usata spesso nei motori per evitare
   di appesantire la memoria e poter sfruttare particolari
   operazioni binarie come l'and e l'or sul singolo bit
*/
#define None 0
#define King 1
#define Pawn 2
#define Knight 3
#define Bishop 5
#define Rook 6
#define Queen 7

#define White 8
#define Black 16

bool IsColour (int, int);

int Colour (int);

int PieceType(int);

bool IsRookOrQueen (int);

bool IsBishopOrQueen (int);

bool IsSlidingPiece (int);
```

Il file Piece.c contiene le funzioni che restituiscono informazioni sui pezzi della scacchiera, come ad esempio tipologia e colore.

```
#include "Piece.h"

/* Il pezzo viene rappresentato come la somma tra il
   valore del colore e il valore della tipologia. */

/*
// IP piece intero che rappresenta il pezzo con colore da identificare
// IP colour colore che ci si aspetta abbia il pezzo
// OR true se $piece e' del colore $colour, false altrimenti
// Dato in input un colore e un pezzo restituisce vero
// se il pezzo e' del colore in input
*/
bool IsColour (int piece, int colour) {
      return (piece > 0) ? (piece / 16 == colour / 16) : false;
} /* IsColour */

/*
// IP piece intero che rappresenta il il pezzo con colore da identificare
// OR intero che rappresenta il colore del pezzo
// Restituisce il colore di un pezzo dato in input
*/
int Colour (int piece) {
      if( piece / 16 == 0) return White;
      else return Black;
} /* Colour */

/*
// IP piece intero che rappresenta il pezzo con colore da identificare
// OR intero che rappresenta la tipologia del pezzo
// Restituisce la tipologia di pezzo dato in input
*/
int PieceType (int piece) {
    return piece % 8;
} /* PieceType */

/*
// IP piece intero che rappresenta il pezzo con colore da identificare
// OR true se la tipologia di pezzo di $piece e' una regina o una torre
*/
bool IsRookOrQueen (int piece) {
      return ( (piece % 8) == 6 ) || ( (piece % 8) == 7 );
} /* IsRookOrQueen */

/*
// IP piece intero che rappresenta il pezzo con colore da identificare
// OR true se la tipologia di pezzo di $piece e' una regina o un alfiere
*/
bool IsBishopOrQueen (int piece) {
```

```
      return ( (piece % 8) == 5 ) || ( (piece % 8) == 7 );
} /* IsBishopOrQueen */

/*
// IP piece intero che rappresenta il pezzo con colore da identificare
// OR true se la tipologia di pezzo di $piece e' una regina o un alfiere o
una torre
*/
bool IsSlidingPiece (int piece) {
      return ( (piece % 8) == 5 ) || ( (piece % 8) == 6 ) || ( (piece % 8)
== 7 );
} /* IsSlidingPiece */
```

Il file Move.h contiene la struttura dati utile a rappresentare una mossa effettuabile sulla scacchiera.

```
#include "Piece.h"
#include "BoardRepresentation.h"

/* Questi sono i valori per i flag ovvero
   identificano se e' possibile o meno fare
   le mosse speciali che occorrono solo in
   particolari condizioni di gioco */
#define EnPassantCapture 1
#define Castling 2
#define PromoteToQueen 3
#define PromoteToKnight 4
#define PromoteToRook 5
#define PromoteToBishop 6
#define PawnTwoForward 7

/* La mossa viene rappresentata come la casella
   da cui si parte, la casella a cui si arriva e
   il flag che dice che tipologia di mossa speciale
   e', nel caso non fosse speciale tale valore e' nullo */
typedef struct {
    int fromSquare;
    int toSquare;
    int flag;
} Move;

void MoveInit(int, int, int, Move*);

int StartSquare(Move*);

int TargetSquare(Move*);

int MoveFlag(Move*);

bool IsPromotion(Move*);

int PromotionPieceType(Move*);

void InvalidMove(Move*);

bool SameMove (Move*, Move*);

bool IsInvalid(Move*);

void Name(Move*);
```

Il file Move.c contiene le funzioni che gestiscono la creazione di una mossa sulla scacchiera.

```
#include <stdio.h>
#include "Move.h"

/* Inizializzo una mossa dando in ingresso i parametri che la costituiscono
// IP startSquare casella di partenza
// IP targetSquare casella di destinazione
// IP flag identificatore di mossa speciale
// IOP move conterra' la mossa creata
*/
void MoveInit (int startSquare, int targetSquare, int flag, Move* move) {
    move->fromSquare = startSquare;
    move->toSquare = targetSquare;
    move->flag = flag;
} /* MoveInit */

/* funzioni che ritornano i campi dello struct Mossa*/
int StartSquare(Move* move) {
      return move->fromSquare;
} /* StartSquare */

int TargetSquare(Move* move) {
      return move->toSquare;
} /* TargetSquare */

int MoveFlag(Move* move) {
            return move->flag;
} /* MoveFlag */

/* restituisce vero se la mossa contiene una promozione di un pedone
// IP move mossa da analizzare
// OR true se la mossa e' una promozione di pedone
*/
bool IsPromotion(Move* move) {
            int flag = move->flag;
            return flag == PromoteToQueen || flag == PromoteToRook || flag
== PromoteToKnight || flag == PromoteToBishop;
} /* IsPromotion*/

/* Restituisce la tipologia di pezzo nel quale il pedone promuove
// IP move mossa da analizzare
// OR intero che rappresenta la tipologia di pezzo nel quale il pedone pro-
muove
*/
int PromotionPieceType(Move* move) {
      switch (move->flag) {
            case PromoteToRook:
                  return Rook;
            case PromoteToKnight:
                  return Knight;
            case PromoteToBishop:
```

```
                return Bishop;
            case PromoteToQueen:
                return Queen;
            default:
                return None;
      } /* switch */
} /* PromotionPieceType */

/* Annulla la mossa se non e' valida
// IOP move tutti i parametri della mossa vengono posti a zero
**nota: tale mossa non puo' avvenire in partita in quanto un pezzo non puo' rimanere fermo nella stessa casella
*/
void InvalidMove(Move* move) {
      move->fromSquare = 0;
      move->toSquare = 0;
      move->flag = 0;
} /* InvalidMove */

/* Compara due mosse se vedere se sono uguali
// IP a mossa da confrontare
// IP b mossa da confrontare
// OR true se le due mosse sono uguali, false altrimenti
*/
bool SameMove (Move* a, Move* b) {
      return a->fromSquare == b->fromSquare && a->toSquare == b->toSquare && a->flag == b->flag;
} /* SameMove */

/* Restituisce vero se la mossa non e' valida
// IP move mossa da analizzare
// OR true se la mossa e' invalida, false altrimenti
*/
bool IsInvalid(Move* move) {
      return move->fromSquare == 0 && move->toSquare == 0 && move->flag == 0;
} /* IsInvalid */

/* Restituisce il nome della mossa
// IP move mossa da scrivere
//    OV    nome    della    mossa    nel    formato:    casella_di_partenza-casella_di_destinazione
*/
void Name(Move* move) {
    char strStartSquare[3];
    char strTargetSquare[3];
    SquareNameFromIndex(move->fromSquare, strStartSquare);
    SquareNameFromIndex(move->toSquare, strTargetSquare);
    printf("%s-%s-%d", strStartSquare, strTargetSquare, move->flag);
} /* Name */
```

Il file FenUtility.h contiene la struttura dati per impostare una posizione sulla scacchiera, può essere usata per partire a giocare da una posizione non standard.

```
#include "Piece.h"

/* Posizione di partenza standard utile per inizializzare una partita */
#define startFen "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"

/* I valori booleani contengono le informazioni sulle mosse speciali, epFile sulle
   colonne che possono usufruire della presa al passo (mossa speciale dei pedoni)
   $whiteToMove se la mossa sta al bianco $plyCount il numero di mosse fatte da un
   giocatore e dall'altro sommate queste informazioni sono necessarie per scrivere
   il FEN di una posizione
   ***nota: NON SI E' CONSIDERATA LA CONDIZIONE DELLE 50 MOSSE SENZA CATTURE O SPINTE DI PEDONE
*/
typedef struct {
      int squares[64];
      bool whiteCastleKingside;
      bool whiteCastleQueenside;
      bool blackCastleKingside;
      bool blackCastleQueenside;
      int epFile;
      bool whiteToMove;
      int plyCount;
} LoadedPositionInfo;

int pieceTypeFromSymbol(char);

char pieceFromCode(int);

void PositionFromFen (LoadedPositionInfo*, char[]);
```

Il file FenUtility.c contiene la funzioni che permettono di impostare una posizione sulla scacchiera, può essere usata per partire a giocare da una posizione non standard.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "BoardRepresentation.h"
#include "FenUtility.h"

/* Converte le lettere del fen nei valori dei pezzi
// IP pieceChar pezzo rappresentato come lettera (formato FEN)
// OR intero che rappresenta tale tipologia di pezzo secondo le notazioni
definite in Piece.h
*/
int pieceTypeFromSymbol(char pieceChar) {
    int pieceType;
    switch(pieceChar) {
            case 'R':
                  pieceType = Rook;
                  break;
            case 'N':
                  pieceType = Knight;
                  break;
            case 'B':
                  pieceType = Bishop;
                  break;
            case 'Q':
                  pieceType = Queen;
                  break;
            case 'K':
                  pieceType = King;
                  break;
            case 'P':
                  pieceType = Pawn;
                  break;
            } /* switch */
      return pieceType;
} /* pieceTypeFromSymbol */

/* Converte i valori del motore nelle lettere dei pezzi per  il fen
// IP pieceCode intero che rappresenta il pezzo nel formato definito in Pie-
ce.h
// OR carattere che rappresenta il pezzo secondo lo standard FEN
*/
char pieceFromCode(int pieceCode) {
    char pieceChar;
    switch(pieceCode) {
            case 9:
                  pieceChar = 'K';
                  break;
```

```
            case 10:
                  pieceChar = 'P';
                  break;
            case 11:
                  pieceChar = 'N';
                  break;
            case 13:
                  pieceChar = 'B';
                  break;
            case 14:
                  pieceChar = 'R';
                  break;
            case 15:
                  pieceChar = 'Q';
                  break;
            case 17:
                  pieceChar = 'k';
                  break;
            case 18:
                  pieceChar = 'p';
                  break;
            case 19:
                  pieceChar = 'n';
                  break;
            case 21:
                  pieceChar = 'b';
                  break;
            case 22:
                  pieceChar = 'r';
                  break;
            case 23:
                  pieceChar = 'q';
                  break;
            } /* switch */
      return pieceChar;
} /* pieceFromCode */

/* Carica la posizione da un fen
// IOP loadedPositionInfo punta ad uno struct che conterra' le informazioni
estrapolate dal FEN
// sufficienti a determinare la posizione sulla scacchiera
// IP fen stringa di caratteri che rappresenta la posizione nel formato FEN
*/
void PositionFromFen (LoadedPositionInfo* loadedPositionInfo, char fen[]) {
    char* token = strtok(fen," "); /* Spezza il fen in base al carattere
spazio */
    char* symbol = token;
      int file = 0; /* La lettura parte dalla prima colonna e ultima riga
dunque inizializzo i valori */
      int rank = 7;
      int pieceColour; /* Contengono le informazioni sui pezzi che si incon-
```

```
trano nella lettura */
      int pieceType;
      char castlingRights[10];
      char fileNamesInFen[] = fileNames;

    /* Innanzitutto creo la scacchiera $squares vuota */
    memset(loadedPositionInfo->squares, 0, 64*sizeof(int));
    /* finche' la prima stringa del fen non finisce*/
      while (*symbol != '\0') {
        /* Simbolo di fine riga, torno alla riga precedente */
        if (*symbol == '/') {
                  file = 0;
                  rank--;
                  symbol++;
            } /* if */
        else {
                  /* Simbolo di salto di caselle*/
            if (isdigit(*symbol))
                        file += strtol(symbol, &symbol, 10);
            /* Simbolo che e' un pezzo, lo aggiungo all'array squares*/
            else {
                        pieceColour = (isupper(*symbol)) ? White : Black;
                        pieceType = pieceTypeFromSymbol(toupper(*symbol));
                        loadedPositionInfo->squares[rank * 8 + file] =
pieceType + pieceColour;
                        file++;
                        symbol++;
                  } /* else*/
            } /* else */
      } /* while */
      /* Seconda stringa del FEN, giocatore a cui spetta la mossa */
      token = strtok(NULL," ");
      loadedPositionInfo->whiteToMove = (*token == 'w');

      /* Terza stringa del FEN diritti di arrocco */
      token = strtok(NULL," ");
      (strlen(token) > 2) ? strcpy(castlingRights, token) :
strcpy(castlingRights, "KQkq");
      loadedPositionInfo->whiteCastleKingside = strchr(castlingRights, 'K')
!= NULL;
      loadedPositionInfo->whiteCastleQueenside = strchr(castlingRights, 'Q')
!= NULL;
      loadedPositionInfo->blackCastleKingside = strchr(castlingRights, 'k')
!= NULL;
      loadedPositionInfo->blackCastleQueenside = strchr(castlingRights, 'q')
!= NULL;

    /* Quarta stringa del FEN presa al passo*/
    token = strtok(NULL," ");
      if (strlen(token) > 1) {
            char enPassantFileName = *token;
            if (strchr(fileNamesInFen, enPassantFileName) != NULL) {
```

```
                loadedPositionInfo->epFile = *strchr(fileNames, enPassant-
FileName) + 1;
            } /* if */
      } /* if */
      /* Altrimenti non ci sono state prese al passo */
    loadedPositionInfo->epFile = 0;

      /* Ultima stringa del FEN numero di mosse giocate */
      token = strtok(NULL," ");
      loadedPositionInfo->plyCount = atoi(token);
} /* PositionFromFen */
```

Il file PieceList.h contiene le strutture dati utili per la gestione dei pezzi sulla scacchiera, permette di salvare la loro posizione e accederci più velocemente.

```
/*
    L'idea di questa struttura dati e' di velocizzare lo svluppo del gioco, in quanto per trovare
    un pezzo nella scacchiera si dovrebbe ricorrere ad una ricerca lineare ogni volta, mentre cosi'
    facendo se si vuole prendere un pezzo bastera' accedere alla sua struttura dati per poterlo spostare
    e/o rimuovere dalla scacchiera.

    $occupiedSquares contiene gli indici 0-63 che rappresentano le caselle occupate dai pezzi
    contenuti nella PieceList che si considera, solo gli elementi fino a $numPieces sono validi,
    gli altri sono inutilizzati o garbage.

    $map permette di andare dall'indice di una casella,
    all'indice in cui quel pezzo e' salvato in $occupiedSquares
*/
typedef struct {
      int occupiedSquares[16];
      int map[64];
      int numPieces;
} PieceList;

void PieceListInit(PieceList*);

int Count(PieceList*);

void AddPieceAtSquare (int, PieceList*);

void RemovePieceAtSquare (int, PieceList*);

void MovePiece (int, int, PieceList*);
```

Il file PieceList.c contiene le funzioni utili per la gestione dei pezzi sulla scacchiera, permette la loro inizializzazione e il loro movimento.

```
/* Inizializzazione della PieceList
// IOP pL pieceList che rappresenta qualche pezzo della partita
// in questa funzione si pongono a zero tutti i valori per non avere
// residui di vecchie informazioni
*/
void PieceListInit(PieceList* pL) {
    memset(pL->map, 0, 64*sizeof(int));
    memset(pL->occupiedSquares, 0, 16*sizeof(int));
    pL->numPieces = 0;
} /* PieceListInit */

/* Restituisce il numero di pezzi contenuti in $pL*/
int Count(PieceList* pL) {
      return pL->numPieces;
} /* Count */

/* Aggiunge un nuovo pezzo ad una casella usato all'inizio della partita per
inizializzare i pezzi
// IP square casella in cui aggiungere il pezzo
// IOP pL pieceList in cui il pezzo viene aggiunto
*/
void AddPieceAtSquare (int square, PieceList* pL) {
      pL->occupiedSquares[pL->numPieces] = square; /* Lo aggiungo ai pezzi
che occupano una casella */
      pL->map[square] = pL->numPieces; /* Lo aggiungo nella mappa della
scacchiera */
    pL->numPieces++; /* Incremento il numero di pezzi */
} /* AddPieceAtSquare */

/* Rimuove un pezzo da una casella
// IP square casella in cui rimuovere il pezzo
// IOP pL pieceList in cui il pezzo viene rimosso
*/
void RemovePieceAtSquare (int square, PieceList* pL) {
      /* Prende l'indice in cui e' salvato in $occupiedSquares */
    int pieceIndex = pL->map[square];
      /* Sposta l'ultimo elemente nell'indice di quello da rimuovere */
      pL->occupiedSquares[pieceIndex] = pL->occupiedSquares[pL->numPieces -
1];
      /* Aggiorna la mappa alla nuova posizione dell'elemento spostato
nell'array*/
      pL->map[pL->occupiedSquares[pieceIndex]] = pieceIndex;
      pL->numPieces--;
} /* RemovePieceAtSquare */

/* Sposta un pezzo da una casella ad un'altra
// IP square casella in cui spostare il pezzo
// IOP pL pieceList in cui il pezzo viene spostato
*/
```

```
void MovePiece (int startSquare, int targetSquare, PieceList* pL) {
      int pieceIndex = pL->map[startSquare]; /* Prende l'indice che rappre-
senta il pezzo in occupiedSquares */
      pL->occupiedSquares[pieceIndex] = targetSquare; /* Inserisce la nuova
posizione */
      pL->map[targetSquare] = pieceIndex; /* Inserisce nella nuova posizione
l'indice */
} /* MovePiece */
```

Il file GameState.h contiene la struttura dati utile a tener conto delle mosse speciali possibili sulla scacchiera, attraverso dei valori che indicano la possibilità o meno di compierle.

```
/* Questa struttura serve a tener conto delle mosse speciali
    Per il $castlingRights uso i valori:
    0: nessun arrocco concesso
    1: arrocco corto del bianco concesso
    2: arrocco lungo del bianco concesso
    4: arrocco corto del nero concesso
    8: arrocco lungo del bianco concesso
    Se tutti sono possbili il valore sara' 15 = 8+4+2+1
*/
typedef struct {
    int castlingRights;
    int epSquares;
    int capturedPiece;
} GameState;

void InitGameState(int, int, int, GameState*);

void InitGameStateFromGameState(GameState*, GameState*);

void PrintGameState(GameState*);
```

Il file GameState.c contiene e funzioni utili ad inizializzare i valori necessari alla gestione delle mosse speciali.

```
#include <stdio.h>
#include "GameState.h"

/*
// IP castlingRights rappresenta quali arrocchi sono possibili
// IP epSquares rappresenta la colonna sulla quale e' possibile effettuare
l'en passant
// IP capturedPiece rappresenta se un pezzo e' stato catturato
// IOP g puntatore ad un gamestate che conterra' le informazioni date in in-
put
*/
void InitGameState(int castlingRights, int epSquares, int capturedPiece,
GameState* g) {
    g->castlingRights = castlingRights;
    g->epSquares = epSquares;
    g->capturedPiece = capturedPiece;
} /* InitGameState */

/*
// IOP a puntatore ad un gamestate che conterra' le informazioni date in in-
put
// IP b puntatore ad un gamestate che contiene le informazioni da trasferire
*/
void InitGameStateFromGameState(GameState* a, GameState* b) {
    a->castlingRights = b->castlingRights;
    a->epSquares = b->epSquares;
    a->capturedPiece = b->capturedPiece;
} /* InitGameStateFromGameState */

/*
// IP g gamestate contenente le informazioni
// OV informazioni contenute in $g
*/
void PrintGameState(GameState* g) {
    printf("Print GameState: %d\t%d\t%d\n", g->castlingRights, g->epSquares,
g->capturedPiece);
} /* PrintGameState */
```

Il file Stack.h è una semplice struttura dati di tipo pila.

```
#include <stdbool.h>
#include "GameState.h"

typedef struct {
    int MAXSIZE;
    GameState* stack;
    int top;
} Stack;

void initStack(Stack*);

bool IsEmpty(Stack*);

bool IsFull(Stack*);

void peek(Stack*, GameState*);

void pop(Stack*, GameState*);

void doubleIfFull(Stack*);

void push(Stack*, GameState*);
```

Il file Stack.c contiene le classiche funzioni attuabili con una pila, come push e pop.

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "Stack.h"

/*
// IOP s stack da inizializzare
*/
void initStack(Stack* s) {
    s->MAXSIZE = 50;
    s->top = -1;
    s->stack = malloc(s->MAXSIZE * sizeof(GameState));
    assert(s->stack != NULL);
} /* initStack */

/*
// IP s stack da analizzare
// OR true se la pila e' vuota
*/
bool IsEmpty(Stack* s) {
      return s->top == -1;
} /* IsEmpty */

/*
// IP s stack da analizzare
// OR true se la pila e' piena
*/
bool IsFull(Stack* s) {
      return s->top + 1 == s->MAXSIZE;
} /* IsFull */

/*
// IP s stack da analizzare
// IOP g gamestate che conterra' il gamestate sulla cima di $s
*/
void peek(Stack* s, GameState* g) {
   InitGameStateFromGameState(g, &s->stack[s->top]);
} /* peek */

/*
// IP s stack da cui rimuovere un elemento
// IOP data gamestate che conterra' il gamestate sulla cima di $s apppena
rimosso
*/
void pop(Stack* s, GameState* data) {

   if(!IsEmpty(s)) {
      peek(s, data);
      s->top = s->top - 1;
```

```
   } /* if */
   else {
      printf("Could not retrieve data, Stack is empty.\n");
   } /* else */
} /* pop */

/* Raddoppia la capienza dell'array $stack quando questo e` pieno.
IOP s Stack di cui raddoppiare la capienza.
*/
void doubleIfFull(Stack* s) {
    if (IsFull(s)) {
        int i;
        GameState* newStack = malloc(2 * s->MAXSIZE * sizeof(GameState));
        assert(newStack != NULL);
        for (i = 0; i < s->MAXSIZE; i++)
            InitGameStateFromGameState(&newStack[i], &s->stack[i]);
        s->MAXSIZE *= 2;
        free(s->stack);
        s->stack = newStack;
    } /* if */
} /* doubleIfFull */

/*
// IOP s stack in cui aggiungere un elemento
// IP data gamestate che viene aggiunto in cima ad $s
*/
void push(Stack* s, GameState* data) {
    if(IsFull(s)) {
        doubleIfFull(s);
    } /* if */
    s->top = s->top + 1;
    InitGameStateFromGameState(&s->stack[s->top], data);

} /* push */
```

Il file Board.h contiene la struttura dati utile a gestire la posizione in una partita e ad attuare operazioni come eseguire una mossa, ripristinare una mossa oppure stampare il codice FEN che rappresenta la posizione.

```
#include "Move.h"
#include "PieceList.h"
#include "FenUtility.h"
#include "Stack.h"

/* Definisco un indice per bianco e nero al fine di agevolare la lettura del codice */
#define WhiteIndex 0
#define  BlackIndex 1

typedef struct {
    /* Array che mappa le 64 caselle e per ciascuna
    scrive il codice del pezzo che c'e', se non c'e' niente
    il valore e' quello di none cioe' zero */
    int Square[64];

    bool WhiteToMove;
    int ColourToMove;
    int OpponentColour;
    int ColourToMoveIndex;

    int plyCount; /* Mosse compiute in totale dai due giocatori */

    /* Indice della casella occupata dal re bianco e nero sono messi a parte in quanto se il re
    // e' sotto scacco molte mosse non sono attuabili, dunque e' molto importante conoscere la posizione
    // del re il piu' velocemente possibile
    */
    int KingSquare[2];

    /* Indicano quali sono le posizioni dei vari pezzi*/
    PieceList rooks[2];
    PieceList bishops[2];
    PieceList queens[2];
    PieceList knights[2];
    PieceList pawns[2];
    PieceList emptyList;

    /* Conterra' tutte le $PieceList di ciascun giocatore, per agevolare le operazioni si preferisce
    // tenere alcune piecelist volutamente vuote */
    PieceList* allPieceLists[16];

    /* Tiene conto della possibilita' di compiere alcune mosse speciali */
    GameState currentGameState;
    Stack gameStateHistory;
```

```
} Board;

PieceList* GetPieceList (int, int, PieceList**);

void Initialize(Board*);

void MakeMove(Board*, Move*);

void UnmakeMove (Board*, Move*);

void LoadStartPosition(Board*);

void LoadPosition(Board*, char* fen);

void CurrentFen(Board*, char*);

void drawChessBoard(Board*);
```

Il file Board.c contiene le funzioni utile ad attuare operazioni come eseguire una mossa, ripristinare una mossa oppure stampare il codice FEN che rappresenta la posizione.

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include "Board.h"

/* Ogni pezzo ha una sua PieceList, in $allPieceLists le memorizzo tutte e poi seleziono quella
che mi interessa, scegliendo il pezzo e il colore
// IP pieceType tipologia di pezzo di interesse
// IP colourIndex colore del pezzo di interesse
// IOP allPieceLists puntatore a tutte le piecelist, si ritorna il puntatore a quella associata ai dati in input
*/
PieceList* GetPieceList (int pieceType, int colourIndex, PieceList** allPieceLists) {
    return allPieceLists[colourIndex * 8 + pieceType];
} /* GetPieceList */

void Initialize (Board* b) {
    b->plyCount = 0;

    /* Creo una scacchiera vuota */
    memset(b->Square, 0, 64*sizeof(int));
    memset(b->KingSquare, 0, 2*sizeof(int));

    /* Inizializzo tutte le PieceList */
    PieceListInit(&b->pawns[WhiteIndex]);
    PieceListInit(&b->knights[WhiteIndex]);
    PieceListInit(&b->bishops[WhiteIndex]);
    PieceListInit(&b->rooks[WhiteIndex]);
    PieceListInit(&b->queens[WhiteIndex]);
    PieceListInit(&b->pawns[BlackIndex]);
    PieceListInit(&b->knights[BlackIndex]);
    PieceListInit(&b->bishops[BlackIndex]);
    PieceListInit(&b->rooks[BlackIndex]);
    PieceListInit(&b->queens[BlackIndex]);
    PieceListInit(&b->emptyList);

    /* Inserisco le varie PieceList in $allPieceLists*/
    b->allPieceLists[0] = &b->emptyList;
    b->allPieceLists[1] = &b->emptyList;
    b->allPieceLists[2] = &b->pawns[WhiteIndex];
    b->allPieceLists[3] = &b->knights[WhiteIndex];
    b->allPieceLists[4] = &b->emptyList;
    b->allPieceLists[5] = &b->bishops[WhiteIndex];
    b->allPieceLists[6] = &b->rooks[WhiteIndex];
    b->allPieceLists[7] = &b->queens[WhiteIndex];
    b->allPieceLists[8] = &b->emptyList;
    b->allPieceLists[9] = &b->emptyList;
```

```
    b->allPieceLists[10] = &b->pawns[BlackIndex];
    b->allPieceLists[11] = &b->knights[BlackIndex];
    b->allPieceLists[12] = &b->emptyList;
    b->allPieceLists[13] = &b->bishops[BlackIndex];
    b->allPieceLists[14] = &b->rooks[BlackIndex];
    b->allPieceLists[15] = &b->queens[BlackIndex];

    initStack(&b->gameStateHistory);

} /* Initialize */

/* Fa una mossa sulla scacchiera
// IP move mossa da attuare nella scacchiera
// IOP b scacchiera sulla quale effettuare la mossa $move
*/
void MakeMove (Board* b, Move* move) {

    /* Salvo i dati sull'arrocco prima della mossa */
      int originalCastleState = b->currentGameState.castlingRights;
      int newCastleState = originalCastleState; /* Per il momento inizializ-
zo che non ci sono cambiamenti */
    int opponentColourIndex = 1 - b->ColourToMoveIndex;
    int moveFrom = StartSquare(move);
    int moveTo = TargetSquare(move);
    int capturedPieceType = PieceType(b->Square[moveTo]); /* Tipologia di
pezzo in $moveTo */
    int movePiece = b->Square[moveFrom]; /* Pezzo presente nella casella
$moveFrom */
    int movePieceType = PieceType(movePiece); /* Tipologia di pezzo presente
nella casella $moveFrom */
    int moveFlag = MoveFlag(move); /* $moveFlag contiene informazioni su
eventuali mosse speciali*/
    bool isPromotion = IsPromotion(move); /* Controllo se e' una promozione
di pedone */
    bool isEnPassant = moveFlag == EnPassantCapture; /* Controllo se e' una
cattura en passant */
    int pieceOnTargetSquare, epPawnSquare, castlingRookFromIndex, castling-
RookToIndex, file, promoteType;
    bool kingside;

    /* Cancello ora le informazioni in gamestate in quanto sono cambiate */
      InitGameState(0,0,0, &b->currentGameState);
    /* Inserisco la tipologia di pezzo che viene catturato nel currentGame-
State (anche nessun pezzo) */
    b->currentGameState.capturedPiece = capturedPieceType;
    /* Se e' un pezzo e non e' en passant*/
    if(capturedPieceType != 0 && !isEnPassant) {
        /* Rimuovo il pezzo dell'avversario dalla relativa piecelist*/
        RemovePieceAtSquare(moveTo,  GetPieceList(capturedPieceType,  oppo-
nentColourIndex, b->allPieceLists));
    } /* if */

    /* Sposto i pezzi nelle piecelist, re a parte per via dell'arrocco */
```

```
    if (movePieceType == King) {
      b->KingSquare[b->ColourToMoveIndex] = moveTo; /* Inserisco la nuova
posizione del re */
      if((b->WhiteToMove)) { /* Aggiorno le condizioni di arrocco */
            if((newCastleState % 4) == 3) {
                newCastleState = newCastleState - 3;
            }
            else if((newCastleState % 4) == 2)
                newCastleState = newCastleState - 2;
            else if((newCastleState % 4) == 1)
                newCastleState = newCastleState - 1;
        } /* if */
        else {
            if( (newCastleState - (newCastleState % 4)) == 12)
                newCastleState = newCastleState - 12;
            else if((newCastleState - (newCastleState % 4)) == 8)
                newCastleState = newCastleState - 8;
            else if((newCastleState - (newCastleState % 4)) == 4)
                newCastleState = newCastleState - 4;
        } /* else */
    } /* if */
    else {
      MovePiece(moveFrom,     moveTo,     GetPieceList(movePieceType,     b-
>ColourToMoveIndex, b->allPieceLists));
    } /* else */

    pieceOnTargetSquare = movePiece;

    /* Gestisce la promozione */
    if (isPromotion) {
      promoteType = 0;
      switch (moveFlag) {
            case PromoteToQueen:
                  promoteType = Queen;
                  AddPieceAtSquare(moveTo,                        &b->queens[b-
>ColourToMoveIndex]);
                  break;
            case PromoteToRook:
                  promoteType = Rook;
                  AddPieceAtSquare(moveTo,    &b->rooks[b->ColourToMoveIndex]
);
                  break;
            case PromoteToBishop:
                  promoteType = Bishop;
                  AddPieceAtSquare(moveTo,  &b->bishops[b->ColourToMoveIndex]
);
                  break;
            case PromoteToKnight:
                  promoteType = Knight;
                  AddPieceAtSquare(moveTo,  &b->knights[b->ColourToMoveIndex]
);
                  break;
```

```
      } /* switch */
      pieceOnTargetSquare = promoteType + b->ColourToMove; /* Nuovo pezzo
nella casella di destinazione */
      RemovePieceAtSquare(moveTo,  &b->pawns[b->ColourToMoveIndex]);  /*  Il
pedone viene tolto*/
    } /* if */
    else {
            /* Gestisce le altre mosse speciali, come l'arrocco e la presa
en passant*/
            switch (moveFlag) {
                  case EnPassantCapture:
                        epPawnSquare = moveTo + ((b->ColourToMove == White)
? -8 : 8); /* Casella in cui stava il pedone prima di venir catturato */
                        b->currentGameState.epSquares     =     PieceType(b-
>Square[epPawnSquare]); /* Aggiunge il pedone come tipo di cattura */
                        b->Square[epPawnSquare] = 0; /* Elimina il pedone
che e' stato catturato dalla scacchiera */
                        RemovePieceAtSquare(epPawnSquare,                &b-
>pawns[opponentColourIndex]); /* E dalla relativa PieceList*/
                        break;
                  case Castling:
                        kingside = moveTo == f1 || moveTo == f8; /* Determi-
no se e' un arrocco corto */
                        castlingRookFromIndex = (kingside) ? moveTo + 2 :
moveTo - 2; /* Posizione di partenza della torre */
                        castlingRookToIndex = (kingside) ? moveTo : moveTo +
1; /* Posizione di arrivo della torre */

                b->Square[castlingRookFromIndex] = None; /* Elimino la torre
da dov'e' nella scacchiera */
                        b->Square[castlingRookToIndex]    =    Rook    +    b-
>ColourToMove; /* La aggiungo nella nuova destinazione */

                /* Aggiorno posizione del re*/
                if(kingside) {
                    b->Square[moveTo + 1] = pieceOnTargetSquare;
                    b->KingSquare[b->ColourToMoveIndex] = moveTo + 1; /* In-
serisco la nuova posizione del re */
                } /* if */

                else {
                    b->Square[moveTo] = pieceOnTargetSquare;
                }
                b->Square[moveFrom] = 0;

                        MovePiece(castlingRookFromIndex,    castlingRookToIn-
dex, &b->rooks[b->ColourToMoveIndex]); /* Faccio lo stesso nella PieceList
*/
                        break;
            } /* switch */
      } /* else */

    /* Aggiorno la scacchiera spostando il pezzo mosso */
```

```
    if(moveFlag != 2) {
        b->Square[moveTo] = pieceOnTargetSquare;
        b->Square[moveFrom] = 0;
    }


    /* Se ho mosso di due un pedone attivo la possibilita' della presa al
passo */
      if (moveFlag == PawnTwoForward) {
            file = FileIndex(moveFrom) + 1;
            b->currentGameState.epSquares = file;
      }

      /* Se il pezzo si sta muovendo verso o dalla casella della torre ri-
muovo i diritti di arrocco per quel lato */
      if (originalCastleState != 0) {
            if (moveTo == h1 || moveFrom == h1) {
                  if(newCastleState % 2 == 1)
                      newCastleState = newCastleState - 1;
            } else if (moveTo == a1 || moveFrom == a1) {
                  if((newCastleState % 4) / 2 == 1)
                      newCastleState = newCastleState - 2 ;
            }
            if (moveTo == h8 || moveFrom == h8) {
                  if((newCastleState % 8) / 4 == 1)
                      newCastleState = newCastleState - 4;
            } else if (moveTo == a8 || moveFrom == a8) {
                  if(newCastleState / 8 == 1)
                      newCastleState = newCastleState - 8;
            }
      }
      /* Aggiorno quest'ultimo valore e inserisco il nuovo stato nella pila
*/
    b->currentGameState.castlingRights = newCastleState;
      push(&b->gameStateHistory, &b->currentGameState);

    /* Cambio il giocatore a cui tocca la mossa */
    b->WhiteToMove = !b->WhiteToMove;
    b->ColourToMove = (b->WhiteToMove) ? White : Black;
    b->OpponentColour = (b->WhiteToMove) ? Black : White;
    b->ColourToMoveIndex = 1 - b->ColourToMoveIndex;
    b->plyCount++;
} /* MakeMove */

/* Torna indietro di una mossa sulla scacchiera
// IP move mossa da eliminare dalla scacchiera
// IOP b scacchiera sulla quale eliminare la mossa $move
*/
void UnmakeMove (Board* b, Move* move) {

    int opponentColourIndex = b->ColourToMoveIndex;
```

```
    bool undoingWhiteMove = b->OpponentColour == White;
    int capturedPiece, capturedPieceType, movedFrom, movedTo, moveFlags;
    bool isEnPassant, isPromotion;
    int toSquarePieceType, movedPieceType;
    bool kingside;
    int epIndex, castlingRookFromIndex, castlingRookToIndex;
    b->ColourToMove = b->OpponentColour; /* Lato che ha fatto la mossa che
sto eliminando */
    b->OpponentColour = (undoingWhiteMove) ? Black : White;
    b->ColourToMoveIndex = 1 - b->ColourToMoveIndex;
    b->WhiteToMove = !b->WhiteToMove;

    /* Scopro la tipologia del pezzo che ho catturato*/
    capturedPieceType = b->currentGameState.capturedPiece;
    capturedPiece = (capturedPieceType == 0) ? 0 : capturedPieceType + b-
>OpponentColour;

    movedFrom = move->fromSquare;
    movedTo = move->toSquare;
    moveFlags = move->flag;
    isEnPassant = moveFlags == EnPassantCapture;
    isPromotion = IsPromotion(move);

    toSquarePieceType = PieceType(b->Square[movedTo]);
    movedPieceType = (isPromotion) ? Pawn : toSquarePieceType;



    /* Ignoro le catture en passant che vengono gestite piu' avanti */
    if (capturedPieceType != 0 && !isEnPassant) {
      AddPieceAtSquare(movedTo, GetPieceList(capturedPieceType, opponentCol-
ourIndex, b->allPieceLists));
    } /* if */

    /* Aggiorno la posizione del re */
    if (movedPieceType == King) {
      b->KingSquare[b->ColourToMoveIndex] = movedFrom;
    } else if (!isPromotion) {
      MovePiece(movedTo,    movedFrom,    GetPieceList(movedPieceType,    b-
>ColourToMoveIndex, b->allPieceLists));
    } /* else if */

    /* Rimetto il pezzo dov'era */
    b->Square[movedFrom] = movedPieceType + b->ColourToMove; /* ***Nota se
la mossa fosse stata una promozione, questo rimettera il pezzo promosso in-
vece del pedone. Gestito come mossa speciale nello switch */
    b->Square[movedTo] = capturedPiece; /* Sara' 0 se non e' stato catturato
nessun pezzo */

    if (isPromotion) {
      AddPieceAtSquare (movedFrom, &b->pawns[b->ColourToMoveIndex]);
      switch (moveFlags) {
            case PromoteToQueen:
```

```
                RemovePieceAtSquare(movedTo,                    &b->queens[b-
>ColourToMoveIndex]);
                break;
            case PromoteToKnight:
                RemovePieceAtSquare(movedTo,                  &b->knights[b-
>ColourToMoveIndex]);
                break;
            case PromoteToRook:
                RemovePieceAtSquare(movedTo,                    &b->rooks[b-
>ColourToMoveIndex]);
                break;
            case PromoteToBishop:
                RemovePieceAtSquare(movedTo,                  &b->bishops[b-
>ColourToMoveIndex]);
                break;
      } /* switch */
    } else if (isEnPassant) { /* cattura en passant: rimetto il pedone cat-
turato nella casella giusta */
      epIndex = movedTo + ((b->ColourToMove == White) ? -8 : 8);
      b->Square[movedTo] = 0;
      b->Square[epIndex] = (int) capturedPiece;
      AddPieceAtSquare(epIndex, &b->pawns[opponentColourIndex]);
    } else if (moveFlags == Castling) { /* Arrocco: rimetto la torre dov'era
prima della mossa */

      kingside = movedTo == 6 || movedTo == 62;
      castlingRookFromIndex = (kingside) ? movedTo + 1 : movedTo - 2;
      castlingRookToIndex = (kingside) ? movedTo - 1 : movedTo + 1;

      b->Square[castlingRookToIndex] = 0;
      b->Square[castlingRookFromIndex] = Rook + b->ColourToMove;

      MovePiece(castlingRookToIndex,   castlingRookFromIndex,   &b->rooks[b-
>ColourToMoveIndex]);

    } /* else if */

    pop(&b->gameStateHistory, &b->currentGameState); /* Rimuovo il gamestate
dalla pila */
      peek(&b->gameStateHistory, &b->currentGameState); /* Rimetto il game-
state che adesso e' in cima alla pila */
    b->plyCount--;
} /* UnmakeMove */

/* Carica la posizione di partenza classica
// IOP b scacchiera su cui carciare la posizione di partenza
*/
void LoadStartPosition(Board* b) {
    char str[] = startFen;
    LoadPosition(b, str);
} /* LoadStartPosition */
```

```
/* Carica una posizione custom da un FEN
// IOP b scacchiera su cui carciare la posizione custom
// IP fen codice FEN da cui ricavare la posizione
*/
void LoadPosition(Board* b, char* fen) {
    int squareIndex, whiteCastle, blackCastle, epState;
    LoadedPositionInfo loadedPosition;
    GameState initialGameState;
    Initialize(b);
    PositionFromFen(&loadedPosition, fen);
    /* Carico i pezzi nella scacchiera e nelle pieceList */
    for(squareIndex = 0; squareIndex < 64; squareIndex++) {
      int piece = loadedPosition.squares[squareIndex];
      b->Square[squareIndex] = piece;
      if (piece != None) {
            int pieceType = PieceType(piece);
            int pieceColourIndex = (IsColour(piece, White)) ? WhiteIndex :
BlackIndex;
            if (IsSlidingPiece(piece)) {
                  if (pieceType == Queen) {
                        AddPieceAtSquare(squareIndex,                    &b-
>queens[pieceColourIndex]) ;
                  } else if (pieceType == Rook) {
                        AddPieceAtSquare(squareIndex,                    &b-
>rooks[pieceColourIndex]);
                  } else if (pieceType == Bishop) {
                        AddPieceAtSquare          (squareIndex,          &b-
>bishops[pieceColourIndex]);
                  } /* else if */
            } else if (pieceType == Knight) {
                  AddPieceAtSquare             (squareIndex,             &b-
>knights[pieceColourIndex]);
            } else if (pieceType == Pawn) {
                  AddPieceAtSquare(squareIndex,                          &b-
>pawns[pieceColourIndex]);
            } else if (pieceType == King) {
                  b->KingSquare[pieceColourIndex] = squareIndex;
            }  /* else if */
      } /* if*/
    } /* for */
    /* Lato a cui tocca muovere */
    b->WhiteToMove = loadedPosition.whiteToMove;
    b->ColourToMove = (b->WhiteToMove) ? White : Black;
    b->OpponentColour = (b->WhiteToMove) ? Black : White;
    b->ColourToMoveIndex = (b->WhiteToMove) ? 0 : 1;

    /* Creo il gamestate */
      whiteCastle = ((loadedPosition.whiteCastleKingside) ? 1 : 0) + ((load-
edPosition.whiteCastleQueenside) ? 2 : 0);
      blackCastle = ((loadedPosition.blackCastleKingside) ? 4 : 0) + ((load-
edPosition.blackCastleQueenside) ? 8 : 0);
      epState = loadedPosition.epFile;
```

```
      InitGameState(whiteCastle     +     blackCastle,     epState,     0,
&initialGameState);
      push(&b->gameStateHistory, &initialGameState);
      InitGameStateFromGameState(&b->currentGameState, &initialGameState);
    b->plyCount = loadedPosition.plyCount;
} /* LoadPosition */

/* Restituisce il FEN dalla posizione sulla scacchiera
// IP board scacchiera da cui estrapolare il codice FEN
// IOP fen stringa che conterra' il FEN generato
*/
void CurrentFen(Board* board, char* fen) {
      int rank, file, i, piece, pieceType, epFile;
      bool isBlack;
      char pieceChar = ' ';
      char c[2] = {' ', '\0'};
      int numEmptyFiles = 0, epRank;
      bool whiteKingside, whiteQueenside, blackKingside, blackQueenside;
      /* Riprendo le lettere dal file BoardRepresentation.h */
      char fileNamesInBoard[] = fileNames;
      /* Come di consueto con il FEN la lettura avviene dall'ultima riga
prima colonna a scalare*/
    for (rank = 7; rank >= 0; rank--) {
            for (file = 0; file < 8; file++) {
                  i = rank * 8 + file;
                  /* Estraggo il pezzo in posizione i-esima*/
            piece = board->Square[i];
                  if (piece != 0) {
                        /*  Se  avevo  saltato  delle  posizioni  senza  pezzi
scrivo quante ne ho saltate */
                if (numEmptyFiles != 0) {
                    sprintf(c,"%d", numEmptyFiles);
                    strcat(fen, c);
                              numEmptyFiles = 0;
                    }
                    isBlack = IsColour(piece, Black);
                    pieceType = PieceType(piece);
                    /* Estraggo la lettera di interesse */
                switch (pieceType) {
                              case Rook:
                                    pieceChar = 'R';
                                    break;
                              case Knight:
                                    pieceChar = 'N';
                                    break;
                              case Bishop:
                                    pieceChar = 'B';
                                    break;
                              case Queen:
                                    pieceChar = 'Q';
                                    break;
```

```
                              case King:
                                    pieceChar = 'K';
                                    break;
                              case Pawn:
                                    pieceChar = 'P';
                                    break;
                        } /* switch */
                        /* Aggiungo la lettera facendo attenzione al maiu-
scolo/minuscolo*/
                c[0] = (isBlack) ? tolower(pieceChar) : pieceChar;
                        strcat(fen, c);
                  } /* if */
            else {
                        numEmptyFiles++;
                  } /* else */

            } /* for interno */
            /* Se avevo saltato delle posizioni scrivo quante ne ho saltate
*/
        if (numEmptyFiles != 0) {
            sprintf(c,"%d", numEmptyFiles);
            strcat(fen, c);
            numEmptyFiles = 0;
            } /* if */
            /* Se e' finita la riga */
            if (rank != 0) {
            c[0] = '/';
            strcat(fen, c);
            } /* if */
      } /* for esterno */

    /* Fine prima stringa ora spazio per separare e inserisco a chi tocca */
    c[0] = ' ';
      strcat(fen, c);
      c[0] = (board->WhiteToMove) ? 'w' : 'b';
      strcat(fen, c);

      /* Gestisco l'arrocco */
      whiteKingside = (board->currentGameState.castlingRights % 2) == 1;
      whiteQueenside = ( (board->currentGameState.castlingRights % 4) / 2)
== 1;
      blackKingside = ( (board->currentGameState.castlingRights % 8) / 4) ==
1;
      blackQueenside = (board->currentGameState.castlingRights / 8) == 1;
      c[0] = ' ';
      strcat(fen, c);

      if(whiteKingside) {
        c[0] = 'K';
          strcat(fen, c);
    }
```

```
    if(whiteQueenside) {
        c[0] = 'Q';
          strcat(fen, c);
    }
    if(blackKingside) {
        c[0] = 'k';
          strcat(fen, c);
    }
      if(blackQueenside) {
        c[0] = 'q';
          strcat(fen, c);
    }

    /* Se nessun arrocco e' disponibile scrivo un "-" */
    if(board->currentGameState.castlingRights == 0) {
        c[0] = '-';
          strcat(fen, c);
    }

      /* En-passant */
      c[0] = ' ';
      strcat(fen, c);
      epFile = board->currentGameState.epSquares;
      if (epFile == 0) {
            c[0] = '-';
          strcat(fen, c);
      }
    else {
        c[0] = fileNamesInBoard[epFile - 1];
            epRank = (board->WhiteToMove) ? 6 : 3;
            strcat(fen, c);
        sprintf(c,"%d", epRank);
        strcat(fen, c);
      }
      /* Contatore del turno di gioco (dovrebbe essere 1 all'inizio, e au-
mentare ogni mossa del nero) */
      c[0] = ' ';
      strcat(fen, c);
      sprintf(c,"%d", board->plyCount / 2);
    strcat(fen, c);
} /* CurrentFen */

/* Disegna la scacchiera sul terminale
// IP b scacchiera da disegnare
// OV scacchiera
*/
void drawChessBoard(Board* b) {
    int i, j, k;
    printf("\n\t\t   CHESSBOARD\n");
    for(i = 7; i >= 0; i--) {
        printf("\n\t");
```

```
        for(k = 0; k < 8; k++)
            printf("+---");
        printf("+\n\t|");

        for(j = 0; j < 8; j++) {
            if(b->Square[i*8+j] <= 23 && b->Square[i*8+j] > 0)
            printf(" %c |", pieceFromCode(b->Square[i*8+j]));
            else printf(" %c |", 0);
        } /* for */
        printf(" %d", i+1);
    } /* for */
    printf("\n\t");
    for(k = 0; k < 8; k++)
        printf("+---");
    printf("+\n\t");

    for(k = 0; k < 8; k++)
        printf("  %c ", k+97);
    printf("\n");
}
```

Il file ListInt.h è una semplice lista di interi.

```
/* Lista di interi */
typedef struct nodo{
  int data;
  struct nodo *next;
} Nodo;

void Add(Nodo**, int);

void RemoveAt(Nodo**, int);

void Clear(Nodo* r);

int* ToArray(Nodo*);
```

Il file ListInt.c contiene le funzioni classiche di una lista.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "ListInt.h"

/* Aggiunge un elemento alla lista
// IP data elemento da aggiungere
// IOP radice della lista a cui aggiungere $data
*/
void Add(Nodo ** r, int data) {
    /* Costruzione nuovo nodo */
    Nodo* n = malloc(sizeof(Nodo));
    assert(n != NULL);
    n->data = data;
    n->next = NULL;
    /* Scorro la lista */
    while(*r != NULL) {
        r = &((*r)->next);
    } /* while */
    *r = n;
} /* Add */

/* Elimina un elemento dalla lista
// IP index indice elemento da rimuovere
// IOP radice della lista a cui rimuovere l'elemento $index
*/
void RemoveAt(Nodo ** r, int index) {
    int i;
    Nodo* s;
    for(i = 0; i < index - 1; i++) {
        r = &((*r)->next);
    }
    s = ((*r)->next)->next;
    (*r)->next = s;
} /* RemoveAt */

/* Pulisce la lista eliminando tutto
// IOP r puntatore alla lista da rimuovere
*/
void Clear(Nodo* r) {
    Nodo* s;
    while(r != NULL) {
        s = r->next;
        free(r);
        r = s;
    } /* while */
} /* Clear */

/*
```

```
// IP r puntatore alla lista
// OR puntatore al nuovo array creato
*/
int* ToArray(Nodo* r) {
    int i = 0, j;
    int* a = NULL;
    Nodo* s = r;
    /* Conto il numero di nodi presenti nella lista scorrendola */
    while(s != NULL) {
        i++;
        s = s->next;
    } /* while */
    /* Inizializzo l'array e lo riempio con i valori della lista il primo
valore e' la dimensione dell'array stesso */
    a = malloc((sizeof(int) * i) + 1);
    assert(a != NULL);
    a[0] = i+1;
    for(j = 1; j <= i; j++) {
        a[j] = r->data;
        r = r->next;
    } /* for */
    return a;
} /* ToArray */
```

Il file ListInt.h è una semplice lista di dato strutturato Move.

```
#include "PrecomputedMoveData.h"

typedef struct Mnodo{
  Move data;
  struct Mnodo *next;
} MNodo;

void MAdd(MNodo**, Move*);

void MRemoveAt(MNodo**, int);

void MClear(MNodo* r);

Move* MToArray(MNodo*);
```

Il file ListMove.c contiene le funzioni classiche di una lista.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "ListMove.h"

/* Aggiunge un elemento alla lista
// IP data elemento da aggiungere
// IOP radice della lista a cui aggiungere $data
*/
void MAdd(MNodo ** r, Move* data) {
    /* Costruzione nuovo MNodo */
    MNodo* n = malloc(sizeof(MNodo));
    assert(n != NULL);
    n->data.fromSquare = StartSquare(data);
    n->data.toSquare = TargetSquare(data);
    n->data.flag = MoveFlag(data);
    n->next = NULL;
    /* Scorro la lista */
    while(*r != NULL) {
        r = &((*r)->next);
    } /* while */
    *r = n;
} /* MAdd */

/* Elimina un elemento dalla lista
// IP index indice elemento da rimuovere
// IOP radice della lista a cui rimuovere l'elemento $index
*/
void MRemoveAt(MNodo ** r, int index) {
    int i;
    MNodo* s;
    for(i = 0; i < index - 1; i++) {
        r = &((*r)->next);
    } /* for */
    s = ((*r)->next)->next;
    (*r)->next = s;
} /* MRemoveAt */

/* Pulisce la lista eliminando tutto
// IOP r puntatore alla lista da rimuovere
*/
void MClear(MNodo* r) {
    MNodo* s;
    while(r != NULL) {
        s = r->next;
        free(r);
        r = s;
    } /* while */
} /* MClear */
```

```
/*
// IP r puntatore alla lista
// OR puntatore al nuovo array creato
*/
Move* MToArray(MNodo* r) {
    int i = 0, j;
    Move* a;
    MNodo* s = r;
    /* Conto il numero di nodi presenti nella lista scorrendola */
    while(s != NULL) {
        i++;
        s = s->next;
    } /* while */
    /* Inizializzo l'array e lo riempio con i valori della lista */
    a = malloc((sizeof(Move) * i) + sizeof(Move));
    assert(a != NULL);
    a[0].fromSquare = i;
    a[0].toSquare = i;
    a[0].flag = i;
    for(j = 1; j <= i; j++) {
        a[j].fromSquare = StartSquare(&(r->data));
        a[j].toSquare = TargetSquare(&(r->data));
        a[j].flag = MoveFlag(&(r->data));
        r = r->next;
    } /* for */
    return a;
} /* MToArray */
```

Il file PrecomputedMoveData.h contiene una struttura dati con le informazioni su come si muovono i vari pezzi data la loro casella di partenza.

```
#include "Board.h"
#include "ListInt.h"

/* L'idea di questa struttura e' di avere un supporto per decretare se una mossa e' legale,
   il contenuto sono tutte informazioni che riguardano come si muovono i vari pezzi sulla scacchiera
   data una qualunque posizione.
*/
typedef struct {
    /* Contiene le direzioni di movimento, le prime 4 sono ortogonali
       le ultime 4 sono diagonali (N, S, W, E, NW, SE, NE, SW)
    */
    int directionOffsets[8];

    /* Salva il numero di mosse disponibili in ogni direzione per ognuna delle caselle della scacchiera
       L'ordine delle direzioni e': N, S, W, E, NW, SE, NE, SW
       Ad esempio, se numSquaresToEdge[0][1] == 7...
       significa che ci sono 7 caselle a nord di b1 (la casella con indice 1 nella scacchiera sotto forma di array)
    */
    int numSquaresToEdge[64][8];

    /* Salva gli array di indici per ogni casella in cui un cavallo puo' atterrare per ogni casella della scacchiera
       Dunque ad esempio, knightMoves[0] e' uguale a {10, 17}, significa che un cavallo in a1 puo' saltare in c2 e b3
    */
    int* knightMoves[64];
    int* kingMoves[64];

    /* Direzioni in cui attaccano i pedoni per bianco e nero (NW, NE; SW SE) */
    int pawnAttackDirections[2][2];

    int* pawnAttacksWhite[64];
    int* pawnAttacksBlack[64];
    int directionLookup[127];

    int kingAttackBitboards[64][64];
    int knightAttackBitboards[64][64];
    int pawnAttackBitboards[64][2][64];

    int rookMoves[64][64];
    int bishopMoves[64][64];
    int queenMoves[64][64];

    /* Aka manhattan distance (salva quante mosse sono necessarie ad una torre per raggiungere la casella b dalla casella a) */
```

```
    int orthogonalDistance[64][64];
    /* Aka chebyshev distance (salva quante mosse sono necessarie ad un re
per raggiungere la casella b dalla casella a) */
    int kingDistance[64][64];
    int centreManhattanDistance[64];

} PreComputedMoveData;

int min(int, int);

int max(int, int);

int sign(int);

int NumRookMovesToReachSquare(PreComputedMoveData*, int, int);

int NumKingMovesToReachSquare(PreComputedMoveData*, int, int);

void PrecomputedMoveData(PreComputedMoveData*);
```

Il file PrecomputedMoveData.c contiene le funzioni che determinano dove si può muovere un pezzo data la sua casella di partenza.

```
#include <Math.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include "PrecomputedMoveData.h"

/*
// IP a intero
// IP b intero
// OP minore tra $a e $b
*/
int min(int a, int b) {
    return (a < b) ? a : b;
} /* min */

/*
// IP a intero
// IP b intero
// OP maggiore tra $a e $b
*/
int max(int a, int b) {
    return (a > b) ? a : b;
} /* max */

/*
// IP a intero
// OP segno di $a
*/
int sign(int a) {
```

```
    if(a > 0) {
        return 1;
    } else if(a < 0) {
        return -1;
    } else return 0;
} /* sign */

/* Calcola il numero di mosse che la torre deve compiere per raggiungere
$targetSquare
// IP pcd calcoli precompilati
// IP startSquare casella di partenza
// IP targetSquare casella di arrivo
// OP numero di mosse necessarie alla torre per raggiungere $targetSquare
partendo da $startSquare
*/
int NumRookMovesToReachSquare (PreComputedMoveData* pcd, int startSquare,
int targetSquare) {
      return pcd->orthogonalDistance[startSquare][targetSquare];
} /* NumRookMovesToReachSquare */

/* Calcola il numero di mosse che il re deve compiere per raggiungere $tar-
getSquare
// IP pcd calcoli precompilati
// IP startSquare casella di partenza
// IP targetSquare casella di arrivo
// OP numero di mosse necessarie al re per raggiungere $targetSquare parten-
do da $startSquare
*/
int NumKingMovesToReachSquare (PreComputedMoveData* pcd, int startSquare,
int targetSquare) {
      return pcd->kingDistance[startSquare][targetSquare];
} /* NumKingMovesToReachSquare */

/* Inizializza i dati per il lookup durante la partita */
void PrecomputedMoveData(PreComputedMoveData* pcd) {
      /* Calcola i salti del cavallo e le caselle disponibili per ogni ca-
sella sulla scacchiera. */
      int allKnightJumps[] = { 15, 17, -17, -15, 10, -6, 6, -10 }; /* Salti
che il cavallo puo' effettuare*/
      int squareIndex, y, x, north, south, west, east, i, knightJumpDelta,
knightJumpSquare;
    int knightSquareY, knightSquareX, maxCoordMoveDst, kingMoveDelta, king-
MoveSquare, kingSquareY, kingSquareX;
    int directionIndex, currentDirOffset, n, targetSquare, offset, absOff-
set, absDir, squareA, fileDstFromCentre;
    int rankDstFromCentre, squareB, rankDistance, fileDistance;
    Nodo* legalKnightJumps;
    Nodo* legalKingMoves;
    Nodo* pawnCapturesWhite;
    Nodo* pawnCapturesBlack;
    int knightBitboard[64];

    /* Direzioni lungo le quali si possono muovere alfiere torre e regina */
```

```
    pcd->directionOffsets[0] = 8;
    pcd->directionOffsets[1] = -8;
    pcd->directionOffsets[2] = -1;
    pcd->directionOffsets[3] = 1;
    pcd->directionOffsets[4] = 7;
    pcd->directionOffsets[5] = -7;
    pcd->directionOffsets[6] = 9;
    pcd->directionOffsets[7] = -9;

    /* Direzioni lungo le quali catturano i pedoni (NW, NE, SW, SE) */
    pcd->pawnAttackDirections[0][0] = 4;
    pcd->pawnAttackDirections[0][1] = 6;
    pcd->pawnAttackDirections[1][0] = 7;
    pcd->pawnAttackDirections[1][1] = 5;

      for(squareIndex = 0; squareIndex < 64; squareIndex++) {
            y = squareIndex / 8;
            x = squareIndex - y * 8;
        /* Calcolo quante sono le caselle disponibili in ogni direzione */
            north = 7 - y;
            south = y;
            west = x;
            east = 7 - x;
            /* Le inserisco nell'array insieme alle diagonali */
            pcd->numSquaresToEdge[squareIndex][0] = north;
            pcd->numSquaresToEdge[squareIndex][1] = south;
            pcd->numSquaresToEdge[squareIndex][2] = west;
            pcd->numSquaresToEdge[squareIndex][3] = east;
            pcd->numSquaresToEdge[squareIndex][4] = min(north, west);
            pcd->numSquaresToEdge[squareIndex][5] = min(south, east);
            pcd->numSquaresToEdge[squareIndex][6] = min(north, east);
            pcd->numSquaresToEdge[squareIndex][7] = min(south, west);
            /* Calcolo le caselle sulle quali il cavallo puo' andare parten-
do da una determinata casella */
            legalKnightJumps = NULL;
            memset(knightBitboard, 0, 64*sizeof(int));
            for(i = 0; i < 8; i++) {
                knightJumpDelta = allKnightJumps[i]; /* Prendo l'offset che
costituisce il salto*/
                  knightJumpSquare = squareIndex + knightJumpDelta; /* Creo
la mossa */
                  if (knightJumpSquare >= 0 && knightJumpSquare < 64) { /*
Scarto quelle che finiscono fuori dalla scacchiera */
                        knightSquareY = knightJumpSquare / 8;
                        knightSquareX = knightJumpSquare - knightSquareY *
8;
                        /* Mi assicuro che il cavallo si sia mosso di solo 2
caselle sugli assi x/y (per evitare indici che hanno fatto saltare il caval-
lo da una parte all'altra della scacchiera ) */
                        maxCoordMoveDst = max(abs(x - knightSquareX), abs(y
- knightSquareY));
                        if (maxCoordMoveDst == 2) {
```

```
                    Add(&legalKnightJumps, (int) knightJumpSquare);
                            knightBitboard[knightJumpSquare] = 1;
                        } /* if */
                    } /* if */
            } /* for */
            /* Aggiungo le mosse ad una lista, in quanto non ne conosco il
numero, in tal modo miglioro lo spazio occupato */
            pcd->knightMoves[squareIndex] = ToArray(legalKnightJumps);
            /* Elimino la lista dopo aver convertito in array*/
        Clear(legalKnightJumps);
        /* Collego l'array ad un array di puntatori a formare una matrice
non rettangolare */
            for(i = 0; i < 64; i++) {
                pcd->knightAttackBitboards[squareIndex][i]    =    knightBit-
board[i];
        }
            /* Calcolo tutte le caselle che il re puo' raggiungere (senza
considerare l'arrocco) */
            legalKingMoves = NULL;
            for(i = 0; i < 64; i++) {
                pcd->kingAttackBitboards[squareIndex][i] = 0;
        } /* for */
            for(i = 0; i < 8; i++) {
                /* Prendo la direzione*/
                  kingMoveDelta = pcd->directionOffsets[i];
                  /* Creo la mossa*/
             kingMoveSquare = squareIndex + kingMoveDelta;
             /* Controllo di non essere finito fuori dalla scacchiera*/
                  if (kingMoveSquare >= 0 && kingMoveSquare < 64) {
                        kingSquareY = kingMoveSquare / 8;
                        kingSquareX = kingMoveSquare - kingSquareY * 8;
                        /* E di non aver fatto salti da un punto all'altro
della scacchiera */
                        maxCoordMoveDst = max(abs (x - kingSquareX), abs (y
- kingSquareY));
                        if (maxCoordMoveDst == 1) {
                              Add(&legalKingMoves, (int) kingMoveSquare);
                              pcd-
>kingAttackBitboards[squareIndex][kingMoveSquare] = 1;
                        } /* if */
                  } /* if */
            } /* for */
            /* Come prima metto le mosse in una lista e poi converto in ar-
ray liberando la lista */
            pcd->kingMoves[squareIndex] = ToArray(legalKingMoves);
        Clear(legalKingMoves);



            /* Calcolo le catture dei pedoni che si possono effettuare */
            pawnCapturesWhite = NULL;
            pawnCapturesBlack = NULL;
            /* Inizializzazione */
```

```
            for(i = 0; i < 64; i++) {
                pcd->pawnAttackBitboards[squareIndex][WhiteIndex][i] = 0;
        }
        for(i = 0; i < 64; i++) {
                pcd->pawnAttackBitboards[squareIndex][BlackIndex][i] = 0;
        }
        /* Aggiornamento*/
            if (x > 0) {
                  if (y < 7) {
                        Add(&pawnCapturesWhite,squareIndex + 7);
                        pcd-
>pawnAttackBitboards[squareIndex][WhiteIndex][squareIndex + 7] = 1;
                  } /* if */
                  if (y > 0) {
                        Add (&pawnCapturesBlack, squareIndex - 9);
                        pcd-
>pawnAttackBitboards[squareIndex][BlackIndex][squareIndex - 9] = 1;
                  } /* if */
            } /* if */

            if (x < 7) {
                  if (y < 7) {
                        Add(&pawnCapturesWhite, squareIndex + 9);
                        pcd-
>pawnAttackBitboards[squareIndex][WhiteIndex][squareIndex + 9] = 1;
                  } /* if */
                  if (y > 0) {
                        Add(&pawnCapturesBlack, squareIndex - 7);
                        pcd-
>pawnAttackBitboards[squareIndex][BlackIndex][squareIndex - 7] = 1;
                  } /* if */
            } /* if */
            pcd->pawnAttacksWhite[squareIndex] = ToArray(pawnCapturesWhite);
            Clear(pawnCapturesWhite);
            pcd->pawnAttacksBlack[squareIndex] = ToArray(pawnCapturesBlack);
        Clear(pawnCapturesBlack);
        /* Inizializzazione */
        for(i = 0; i < 64; i++) {
                pcd->rookMoves[squareIndex][i] = 0;
            }
            /* Mosse di torre */
            for (directionIndex = 0; directionIndex < 4; directionIndex++) {
                  currentDirOffset = pcd->directionOffsets[directionIndex];
                  for       (n       =       0;       n       <       pcd-
>numSquaresToEdge[squareIndex][directionIndex]; n++) {
                        targetSquare = squareIndex + currentDirOffset * (n +
1);
                        pcd->rookMoves[squareIndex][targetSquare] = 1;
                  } /* for */
            } /* for */
            /* Inizializzazione */
        for(i = 0; i < 64; i++) {
```

```
                pcd->bishopMoves[squareIndex][i] = 0;
          }
            /* Mosse di alfiere */
            for (directionIndex = 4; directionIndex < 8; directionIndex++) {
                  currentDirOffset = pcd->directionOffsets[directionIndex];
                  for (n = 0; n < pcd->numSquaresToEdge[squareIndex][directionIndex]; n++) {
                        targetSquare = squareIndex + currentDirOffset * (n + 1);
                        pcd->bishopMoves[squareIndex][targetSquare] = 1;
                  } /* for */
            } /* for */
            /* Mosse di regina */
        for(i = 0; i < 64; i++) {
                if(pcd->rookMoves[squareIndex][i] == 1)
                pcd->queenMoves[squareIndex][i] = pcd->rookMoves[squareIndex][i];
            else  pcd->queenMoves[squareIndex][i] = pcd->bishopMoves[squareIndex][i];
          } /* for */

      } /* for */

    /* Restituisce la direzione che collega due caselle con segno*/
      for (i = 0; i < 127; i++) {
            offset = i - 63;
            absOffset = abs(offset);
            absDir = 1;
            if (absOffset % 9 == 0) {
                  absDir = 9;
            } else if (absOffset % 8 == 0) {
                  absDir = 8;
            } else if (absOffset % 7 == 0) {
                  absDir = 7;
            }
        if(offset == 0)
            pcd->directionLookup[i] = 0;
        else
                pcd->directionLookup[i] = absDir * sign(offset);
      } /* for */

      /* Lookup per la distanza */
      for (squareA = 0; squareA < 64; squareA++) {
            Coord coordA;
        CoordFromIndex(squareA, &coordA);
            fileDstFromCentre = max (3 - coordA.fileIndex, coordA.fileIndex - 4);
            rankDstFromCentre = max (3 - coordA.rankIndex, coordA.rankIndex - 4);
            pcd->centreManhattanDistance[squareA] = fileDstFromCentre + rankDstFromCentre;
```

```
            for(squareB = 0; squareB < 64; squareB++) {

                  Coord coordB;
            CoordFromIndex(squareB, &coordB);
                  rankDistance = abs(coordA.rankIndex - coordB.rankIndex);
                  fileDistance = abs(coordA.fileIndex - coordB.fileIndex);
                  pcd->orthogonalDistance[squareA][squareB] = fileDistance +
rankDistance;
                  pcd->kingDistance[squareA][squareB]   =   max(fileDistance,
rankDistance);
            } /* for */
      } /* for */
} /* PrecomputedMoveData */
```

Il file MoveGenerator.h contiene la struttura dati che consente di salvare tutte le possibili mosse legali presenti in una data posizione.

```
#include <stdlib.h>
#include "ListMove.h"

typedef struct {
    /* Lista di mosse presenti nella posizione */
    MNodo* moves;
    bool isWhiteToMove;
    int friendlyColour;
    int opponentColour;
    int friendlyKingSquare;
    int friendlyColourIndex;
    int opponentColourIndex;

    bool inCheck;
    bool inDoubleCheck;
    /* Se c'e' un pezzo inchiodato */
    bool pinsExistInPosition;
    /* Direzione in cui avviene uno scacco */
    int checkRayBitmask[64];
    /* Direzione in cui avviene un'inchiodatura */
    int pinRayBitmask[64];
    /* Caselle attaccate dai cavalli dell'avversario */
    int opponentKnightAttacks[64];
    /* Caselle attaccate da tutti i pezzi dell'avversario tranne i pedoni */
    int opponentAttackMapNoPawns[64];
    /* Caselle attaccate da tutti i pezzi dell'avversario */
    int opponentAttackMap[64];
    /* Caselle attaccate dai pedoni dell'avversario */
    int opponentPawnAttackMap[64];
    /* Caselle attaccate da alfieri torri e regine dell'avversario */
    int opponentSlidingAttackMap[64];

    Board* board;
} MoveGenerator;

bool InCheck (MoveGenerator*);

bool HasKingsideCastleRight(MoveGenerator*);

bool HasQueensideCastleRight(MoveGenerator*);

bool ContainsSquare (int[], int);

bool SquareIsAttacked (MoveGenerator*, int);

void Init(MoveGenerator*);
```

```
bool SquareIsInCheckRay(MoveGenerator*, int);

void GenerateKingMoves(MoveGenerator*, PreComputedMoveData*);

bool IsPinned (MoveGenerator*, int);

bool IsMovingAlongRay (MoveGenerator*, PreComputedMoveData*, int, int, int);

void GenerateSlidingPieceMoves (MoveGenerator*, PreComputedMoveData*, int,
int, int);

void GenerateSlidingMoves(MoveGenerator*, PreComputedMoveData*);

void GenerateKnightMoves(MoveGenerator*, PreComputedMoveData*);

void MakePromotionMoves (MoveGenerator*, int, int);

bool SquareAttackedAfterEPCapture (MoveGenerator*, PreComputedMoveData*,
int, int);

bool InCheckAfterEnPassant (MoveGenerator*, PreComputedMoveData*, int, int,
int);

void GeneratePawnMoves (MoveGenerator*, PreComputedMoveData*);

void UpdateSlidingAttackPiece (MoveGenerator*, PreComputedMoveData*, int,
int, int);

void GenSlidingAttackMap (MoveGenerator*, PreComputedMoveData*);

void CalculateAttackData (MoveGenerator*, PreComputedMoveData*);

MNodo* GenerateMoves(MoveGenerator*, PreComputedMoveData*, Board*);
```

Il file MoveGenerator.c contiene le funzioni che permettono di generare tutte le mosse legali in una posizione sulla scacchiera.

```
#include <stdio.h>
#include "MoveGenerator.h"

/* Questa funzione e' corretta solo dopo l'invocazione di GenerateMove()
// IP mg puntatori ai dati per le mosse generate
// OR true se il re e' sotto scacco
*/
bool InCheck (MoveGenerator* mg) {
      return mg->inCheck;
} /* InCheck */

/* La funzione restituisce vero quando il giocatore può effettuare l'arrocco
corto
// il valore dell'arrocco corto e' 1 se il giocatore e' il bianco e 4 se il
giocatore
// e' il nero dunque vado a vedere se nella rappresentazione binaria del va-
lore che
// rappresenta l'arrocco sono presenti i bit relativi a tali valori con un
and logico
// IP mg puntatore ai dati per le mosse generate
// OR true se il re puo' effettuare l'arrocco corto
*/
bool HasKingsideCastleRight(MoveGenerator* mg) {
      int mask = (mg->board->WhiteToMove) ? 1 : 4;
      return (mg->board->currentGameState.castlingRights & mask) != 0;
} /* HasKingsideCastleRight */

/* La funzione restituisce vero quando il giocatore può effettuare l'arrocco
lungo
// il valore dell'arrocco lungo e' 2 se il giocatore e' il bianco e 8 se il
giocatore
// e' il nero dunque vado a vedere se nella rappresentazione binaria del va-
lore che
// rappresenta l'arrocco sono presenti i bit relativi a tali valori con un
and logico
// IP mg puntatore ai dati per le mosse generate
// OR true se il re puo' effettuare l'arrocco lungo
*/
bool HasQueensideCastleRight(MoveGenerator* mg) {
      int mask = (mg->board->WhiteToMove) ? 2 : 8;
      return (mg->board->currentGameState.castlingRights & mask) != 0;
} /* HasQueensideCastleRight */

/* La funzione restituisce vero quando il bit della bitboard in posizione
Square e' 1
// IP bitboard scacchiera che contiene 1 nelle caselle che soddisfano un re-
quisito
// IP square casella da verificare
// OR true se la casella soddisfa il requisito della $bitboard
*/
```

```
bool ContainsSquare (int bitboard[], int square) {
    return bitboard[square]!= 0;
} /* ContainsSquare */

/* La funzione restituisce vero se esiste un pezzo dell'avversario che rie-
sce in una mossa a
// raggiungere la casella $square
// IP mg puntatore ai dati per le mosse generate
// IP square casella da verificare
// OR true se la casella e' controllata dall'avversario
*/
bool SquareIsAttacked (MoveGenerator* mg, int square) {
    return ContainsSquare(mg->opponentAttackMap, square);
} /* SquareIsAttacked */

/* La funzione inizializza lo struct MoveGenerator che serve a calcolare
tutte le possibili mosse
// attuabili dal giocatore nel suo turno
// IOP mg puntatore ai dati per le mosse generate da inizializzare
*/
void Init(MoveGenerator* mg) {
      int i = 0;
    mg->inCheck = false;
      mg->inDoubleCheck = false;
      mg->pinsExistInPosition = false;
      for(i = 0; i < 64; i++) {
         mg->checkRayBitmask[i]= 0;
         mg->pinRayBitmask[i] = 0;
    }
      /*
    mg->opponentKnightAttacks = 0;
      mg->opponentAttackMapNoPawns = 0;
      mg->opponentAttackMap = 0;
      mg->opponentPawnAttackMap = 0;
      mg->opponentSlidingAttackMap = 0;
    */

      mg->isWhiteToMove = mg->board->ColourToMove == White;
      mg->friendlyColour = mg->board->ColourToMove;
      mg->opponentColour = mg->board->OpponentColour;
      mg->friendlyKingSquare        =        mg->board->KingSquare[mg->board-
>ColourToMoveIndex];
      mg->friendlyColourIndex  =  (mg->board->WhiteToMove)  ?  WhiteIndex  :
BlackIndex;
    mg->opponentColourIndex = 1 - mg->friendlyColourIndex;

} /* Init */

/* La funzione restituisce vero se la casella $square e' sotto il raggio di
uno scacco
// ovvero la casella si interpone tra la linea che collega il pezzo dell'av-
versario che minaccia
```

```
// il re e il re del giocatore
// IP mg puntatore ai dati per le mosse generate
// IP square casella da verificare
// OR true se la casella e' nel raggio di uno scacco al re
*/
bool SquareIsInCheckRay(MoveGenerator* mg, int square) {
    return (mg->inCheck && (mg->checkRayBitmask[square] != 0));
} /* SquareIsInCheckRay*/

/* Questa funzione si occupa di generare tutte le possibili mosse effettuabili dal re durante il turno
// del giocatore
// IOP mg puntatore ai dati per le mosse generate a cui aggiungere le mosse di re
// IP pcm puntatore ai dati di appoggio per il calcolo delle mosse
*/
void GenerateKingMoves(MoveGenerator* mg, PreComputedMoveData* pcm) {
      int i, targetSquare, pieceOnTargetSquare, castleKingsideSquare, castleQueensideSquare;
      bool isCapture;
      Move m;

      /* $pcm->kingMoves[mg->friendlyKingSquare][0] contiene il numero di mosse possibili che il re puo'
    // effettuare in questo turno, bisogna verificare se sono valide*/
    for (i = 1; i < pcm->kingMoves[mg->friendlyKingSquare][0]; i++) {
            targetSquare = pcm->kingMoves[mg->friendlyKingSquare][i]; /* possibile mossa i-esima */
            /* controllo il pezzo presente nella casella di destinazione del re*/
            pieceOnTargetSquare = mg->board->Square[targetSquare];

            /* Salto quest'iterazione se il pezzo nella casella di destinazione e' del giocatore,
        // in quanto non puo' catturare i suoi pezzi e dunque la mossa e' illegale */
            if (IsColour(pieceOnTargetSquare, mg->friendlyColour)) {
            continue;
            } /* if */

        /* Controllo se c'e' un pezzo dell'avversario nella casella di destinazione del re*/
            isCapture = IsColour(pieceOnTargetSquare, mg->opponentColour);
            if (!isCapture) {
                  /* Se non e' una cattura del pezzo avversario, ma la casella e' sotto l'attacco diretto
            // di un pezzo avversario la mossa e' illegale e salto questa iterazione */
                  if (SquareIsInCheckRay(mg, targetSquare)) {
                continue;
                  } /* if */
            } /* if */
```

```
            /* Se la casella non e' sotto l'attacco diretto di un pezzo ne-
mico allora il re puo' effettuare la mossa
        // che viene quindi aggiunta alla lista di mosse realizzabili */
            if (!SquareIsAttacked(mg, targetSquare)) {
                MoveInit(mg->friendlyKingSquare, targetSquare, 0, &m);
                  MAdd(&mg->moves, &m);

                  /* Gestione arrocco: se il re non e' sotto scacco e non ha
catturato un pezzo avversario */
                  if (!mg->inCheck && !isCapture) {
                        /* Arrocco corto: la casella di destinazione deve
essere f1 per il bianco e f8 per il nero e
                // si devono avere i diritti di arroccare*/
                        if ((targetSquare == f1 || targetSquare == f8) &&
HasKingsideCastleRight(mg)) {
                              castleKingsideSquare = targetSquare + 1;
                              /* La casella dopo f1 o f8 dev'essere anch'essa
libera */
                              if (mg->board->Square[castleKingsideSquare] ==
None) {
                                    /* E non deve essere attaccata da pezzi
avversari*/
                        if (!SquareIsAttacked(mg, castleKingsideSquare)) {
                                          /* Aggiungo la mossa alla lista*/
                              MoveInit(mg->friendlyKingSquare,   targetSquare,
Castling, &m);
                              MAdd(&mg->moves, &m);
                                    } /* if */
                              } /* if */
                        } /* if */
                        /* Arrocco lungo, la casella deve essere d1 o d8 e
si procede come nel caso precedente, solo che
                // questa volta sono due le caselle successive a quella di
arrocco a dover essere libere */
                        else if ((targetSquare == d1 || targetSquare == d8)
&& HasQueensideCastleRight(mg)) {
                              castleQueensideSquare = targetSquare - 1;
                              if (mg->board->Square[castleQueensideSquare] ==
None && mg->board->Square[castleQueensideSquare - 1] == None) {
                                    if      (!SquareIsAttacked(mg,      cas-
tleQueensideSquare)) {
                                          MoveInit(mg->friendlyKingSquare,
castleQueensideSquare, Castling, &m);
                              MAdd(&mg->moves, &m);
                                    } /* if */
                              } /* if */
                        } /* else if */
                  } /* if */
            } /* if */
      } /* for */
} /*  GenerateKingMoves */

/* La funzione ritorna vero se il pezzo non si puo' muovere in quanto libe-
rerebbe un attacco di un pezzo
```

```
// avversario verso il re permettendo di catturarlo
// IP mg puntatore ai dati per le mosse generate
// IP square casella da verificare
// OR true se la casella $square e' nel raggio di una inchiodatura al re
*/
bool IsPinned (MoveGenerator* mg, int square) {
      return mg->pinsExistInPosition && (mg->pinRayBitmask[square] != 0);
} /* IsPinned */

/* La funzione ritorna vero se la mossa che si effettua avviene lungo la direzione descritta da $rayDir
// IP mg puntatore ai dati per le mosse generate
// IP pcm puntatore ai dati di appoggio
// IP rayDir direzione lungo la quale ci si aspetta che la mossa stia avvenendo
// IP startSquare casella di partenza
// IP targetSquare casella di arrivo
// OR true se la direzione individuata da $startSquare/$targetSquare coincide con quella di $rayDir
*/
bool IsMovingAlongRay (MoveGenerator* mg, PreComputedMoveData* pcm, int rayDir, int startSquare, int targetSquare) {
      int moveDir = pcm->directionLookup[targetSquare - startSquare + 63]; /* Calcolo la direzione */
      return (rayDir == moveDir || -rayDir == moveDir); /* Controllo se corrisponde */
} /* IsMovingAlongRay */

/* La funzione genera tutte le mosse effettuabili da una torre, alfiere o regina
// IOP mg puntatore ai dati per le mosse generate che conterra' le nuove mosse
// IP pcm puntatore ai dati di appoggio
// IP startSquare casella da cui generare le mosse
// IP startDirIndex indice di partenza per le direzioni in cui ci si muove prese dal directionOffsets di PreComputedMoveData.h
// IP endDirIndex indice finale per le direzioni in cui ci si muove prese dal directionOffsets di PreComputedMoveData.h
*/
void GenerateSlidingPieceMoves (MoveGenerator* mg, PreComputedMoveData* pcm, int startSquare, int startDirIndex, int endDirIndex) {
      int directionIndex, currentDirOffset, n, targetSquare, targetSquarePiece;
    bool isPinned = IsPinned(mg, startSquare);
    Move m;

      /* Se questo pezzo e' inchiodato, e il re e' sotto scacco, questo pezzo non si puo' muovere */
      if (mg->inCheck && isPinned) {
            return;
      } /* if */

    /* directionIndex indica l'indice di directionOffsets da cui prelevare
```

```
le direzioni di movimento
    // per gli alfieri sono le diagonali, per le torri le verticali e le orizzontali, per le regine tutte */
      for (directionIndex = startDirIndex; directionIndex < endDirIndex; directionIndex++) {
            currentDirOffset = pcm->directionOffsets[directionIndex];

            /* Se inchiodato, questo pezzo puo' solo muoversi lungo il raggio che va verso o si allontana dal
        // re, dunque evito di considerare altre direzioni saltando le esecuzioni */
            if (isPinned && !IsMovingAlongRay(mg, pcm, currentDirOffset, mg->friendlyKingSquare, startSquare)) {
                  continue;
            } /* if */

        /* Finche' non si arriva al bordo della scacchiera */
            for (n = 0; n < pcm->numSquaresToEdge[startSquare][directionIndex]; n++) {
                  targetSquare = startSquare + currentDirOffset * (n + 1); /* Mi sposto lungo la direzione */
                targetSquarePiece = mg->board->Square[targetSquare]; /* Controllo se ci sono pezzi */

                  /* Se vengo bloccato da un pezzo del giocatore, smetto di proseguire in questa direzione */
                  if (IsColour(targetSquarePiece, mg->friendlyColour)) {
                        break;
                  } /* if */
                  /* Controllo se e' una cattura */
                  bool isCapture = targetSquarePiece != None;

            /* Controllo se il pezzo si interpone ad un attacco al re*/
                  bool movePreventsCheck = SquareIsInCheckRay(mg, targetSquare);

                  /* Se blocco uno scacco o non sono sotto scacco la mossa e' valida */
                  if (movePreventsCheck || !mg->inCheck) {
                              MoveInit(startSquare, targetSquare, 0, &m);
                    MAdd (&mg->moves, &m);
                  } /* if */
                  /* Se la casella non e' vuota, non posso proseguire oltre in questa direzione
                  // Inoltre, se questa mossa blocca uno scacco, le altre mosse non lo bloccheranno dunque posso uscire */
                  if (isCapture || movePreventsCheck) {
                        break;
                  } /* if */
            } /* for */
      } /* for */
} /* GenerateSlidingPieceMoves */
```

```
/* La funzione genera e raggruppa tutte le mosse che possono effettuare tor-
ri, regine e alfieri insieme
// IOP mg puntatore ai dati per le mosse generate che conterra' le nuove
mosse
// IP pcm puntatore ai dati di appoggio
*/
void GenerateSlidingMoves(MoveGenerator* mg, PreComputedMoveData* pcm) {
      int i;
      /* Estraggo i pezzi in considerazione che sono a disposizione del gio-
catore */
    PieceList rooks = mg->board->rooks[mg->friendlyColourIndex];
    PieceList bishops = mg->board->bishops[mg->friendlyColourIndex];
    PieceList queens = mg->board->queens[mg->friendlyColourIndex];

    /* Per ciascuno genero le mosse stando attento a selezionare le direzio-
ni lungo le quali muovono */
      for (i = 0; i < Count(&rooks); i++) {
            GenerateSlidingPieceMoves(mg, pcm, rooks.occupiedSquares[i], 0,
4);
      } /* for */

      for (i = 0; i < Count(&bishops); i++) {
        GenerateSlidingPieceMoves(mg, pcm, bishops.occupiedSquares[i], 4,
8);
      } /* for */

      for (i = 0; i < Count(&queens); i++) {
            GenerateSlidingPieceMoves(mg, pcm, queens.occupiedSquares[i], 0,
8);
      } /* for */

} /* GenerateSlidingMoves */

/* La funzione genera tutte le mosse effettuabili dai cavalli del giocatore
// IOP mg puntatore ai dati per le mosse generate che conterra' le nuove
mosse
// IP pcm puntatore ai dati di appoggio
*/
void GenerateKnightMoves(MoveGenerator* mg, PreComputedMoveData* pcm) {
      int i, startSquare, knightMoveIndex, targetSquare, targetSquarePiece;
      /* Estraggo i pezzi di interesse */
    PieceList* myKnights = &mg->board->knights[mg->friendlyColourIndex];
    Move m;

    /* Finche' i pezzi non finiscono genero le mosse */
      for (i = 0; i < Count(myKnights); i++) {
            startSquare = myKnights->occupiedSquares[i];
            /* Se il cavallo e' inchiodato non puo' muovere */
            if (IsPinned(mg, startSquare)) {
                  continue;
            } /* for */
```

```
            /* $knightMoveIndex < pcm->knightMoves[startSquare][0] contiene
il numero di mosse effettuabili dal cavallo
        // dalla posizione $startSquare */
            for    (knightMoveIndex    =    1;    knightMoveIndex    <    pcm-
>knightMoves[startSquare][0]; knightMoveIndex++) {
            /* Estraggo la casella di destinazione*/
            targetSquare = pcm->knightMoves[startSquare][knightMoveIndex];
                  /* Controllo se ci sono pezzi*/
            targetSquarePiece = mg->board->Square[targetSquare];
                  /* Se c'e' un pezzo del giocatore salto l'iterazione, o se
si e' sotto scacco e il cavallo non cattura ne' si interpone ad esso */
                  if   (IsColour(targetSquarePiece,   mg->friendlyColour)   ||
(mg->inCheck && !SquareIsInCheckRay(mg, targetSquare))) {
                continue;
                  } /* if */
                  /* Se nessuna delle ipotesi sopra si verifica la mossa e'
valida e si aggiunge alla lista */
                  MoveInit(startSquare, targetSquare, 0, &m);
                  MAdd (&mg->moves, &m);
            } /* for */
      } /* for */
} /* GenerateKnightMoves */

/* Gestisce l'aggiunta delle mosse di promozione
// IOP mg puntatore ai dati per le mosse generate che conterra' le nuove
mosse (promozione)
// IP fromSquare casella di partenza
// IP toSquare casella di arrivo
*/
void MakePromotionMoves(MoveGenerator* mg, int fromSquare, int toSquare) {
      Move m;
    enum PromotionMode { All, QueenOnly, QueenAndKnight };
    int promotionsToGenerate = All;
    /* inserisco la promozione a regina nella lista*/
    MoveInit(fromSquare, toSquare, PromoteToQueen, &m);
    MAdd(&mg->moves, &m);
    /* Se la promozione e' impostata a tutte inserisco tutte le mosse altri-
menti solo quella a cavallo */
      if (promotionsToGenerate == All) {
            MoveInit(fromSquare, toSquare, PromoteToKnight, &m);
        MAdd(&mg->moves, &m);
            MoveInit(fromSquare, toSquare, PromoteToRook, &m);
        MAdd(&mg->moves, &m);
            MoveInit(fromSquare, toSquare, PromoteToBishop, &m);
        MAdd(&mg->moves, &m);
      } else if (promotionsToGenerate == QueenAndKnight) {
            MoveInit(fromSquare, toSquare, PromoteToKnight, &m);
        MAdd (&mg->moves, &m);
      } /* else if */

} /* MakePromotionMoves */
```

```
/* Restituisce vero se il pedone dopo la cattura en passant genera uno scacco al re
// IP mg puntatore ai dati per le mosse generate
// IP pcm puntatore ai dati di appoggio
// IP epCaptureSquare casella in cui e' stato catturato il pedone con la presa en passant
// IP capturingPawnStartSquare casella da cui partiva il pedone che ha catturato en passant
// OR true se dopo la mossa speciale en passant si e' generato uno scacco al re
*/
bool SquareAttackedAfterEPCapture (MoveGenerator* mg, PreComputedMoveData* pcm, int epCaptureSquare, int capturingPawnStartSquare) {
	int i, squareIndex, piece, dirIndex;

	/* Se il re e' attaccato da un pezzo che non e' un pedone dell'avversario si'*/
    if (ContainsSquare(mg->opponentAttackMapNoPawns, mg->friendlyKingSquare)) {
		return true;
	} /* if*/

	/* Ciclo for attraverso le direzioni orizzontali verso le catture ep per vedere se qualche pezzo nemico ora attacca il re */
	dirIndex = (epCaptureSquare < mg->friendlyKingSquare) ? 2 : 3;
	for(i = 0; i < pcm->numSquaresToEdge[mg->friendlyKingSquare][dirIndex]; i++) {
		squareIndex = mg->friendlyKingSquare + pcm->directionOffsets[dirIndex] * (i + 1);
		piece = mg->board->Square[squareIndex];
		if (piece != None) {
			/* Un pezzo del giocatore sta bloccando la vista di questa casella al nemico */
			if (IsColour (piece, mg->friendlyColour)) {
				break;
			} /* if */
			/* Se contiene un pezzo nemico: */
			else {
				if (IsRookOrQueen (piece)) {
					return true;
				} else {
					/* Questo pezzo non e' in grado di muoversi nella direzione attuale, e dunque sta bloccando tutti gli scacchi lungo la linea */
					break;
				} /* else */
			} /* else */
		} /* if */
	} /* for */

	/* Controlla se un pezzo nemico sta controllando questa casella (Non posso usare la bitboard del pedone, perche' e' stato catturato ) */
	for (i = 0; i < 2; i++) {
```

```
            /* Controlla se esiste una diagonale al re da cui il pedone nemico potrebbe starla attaccando */
            if            (pcm->numSquaresToEdge[mg->friendlyKingSquare][pcm->pawnAttackDirections[mg->friendlyColourIndex][i]] > 0) {
                  /* muovo lungo la direzione in cui attaccano i pedoni del giocatore per trovare la casella da cui il pedone nemico attaccherebbe */
                  piece  =  mg->board->Square[mg->friendlyKingSquare  +  pcm->directionOffsets[pcm->pawnAttackDirections[mg->friendlyColourIndex][i]]];
                  if (piece == (Pawn + (mg->opponentColour)))
                        return true;  /* e' un pedone nemico */
            } /* if */
      } /* for */

      return false;
} /* SquareAttackedAfterEPCapture */

/*
Restituisce vero se il pedone dopo la cattura en passant genera uno scacco al re
// IOP mg puntatore ai dati per le mosse generate da cui viene eliminata la mossa eseguita in quanto illegale
// IP pcm puntatore ai dati di appoggio
// IP epCaptureSquare casella in cui e' stato catturato il pedone con la presa en passant
// IP capturingPawnStartSquare casella da cui partiva il pedone che ha catturato en passant
// OR true se dopo la mossa speciale en passant si e' generato uno scacco al re
*/
bool InCheckAfterEnPassant (MoveGenerator* mg, PreComputedMoveData* pcm, int startSquare, int targetSquare, int epCapturedPawnSquare) {
      /* Effettuo la mossa per vedere se si e' aperto uno scacco */
      mg->board->Square[targetSquare] = mg->board->Square[startSquare];
      mg->board->Square[startSquare] = None;
      mg->board->Square[epCapturedPawnSquare] = None;

      bool inCheckAfterEpCapture = false;
      if   (SquareAttackedAfterEPCapture   (mg,   pcm,   epCapturedPawnSquare, startSquare)) {
            inCheckAfterEpCapture = true;
      } /* if */

      /* Rimuovere il cambiamento nella scacchiera */
      mg->board->Square[targetSquare] = None;
      mg->board->Square[startSquare] = (Pawn + mg->friendlyColour);
      mg->board->Square[epCapturedPawnSquare] = (Pawn + mg->opponentColour);
      return inCheckAfterEpCapture;
} /* InCheckAfterEnPassant */

/* Genera le mosse effettuabili dai pedoni durante la mossa
// IOP mg puntatore ai dati per le mosse generate che conterra' le nuove mosse (pedoni)
// IP pcm puntatore ai dati di appoggio
```

```
*/
void GeneratePawnMoves (MoveGenerator* mg, PreComputedMoveData* pcm) {
      int i, startSquare, rank, squareOneForward, j, pawnCaptureDir, tar-
getSquare, targetPiece, squareTwoForward, epCapturedPawnSquare;
      bool oneStepFromPromotion;
    PieceList myPawns = mg->board->pawns[mg->friendlyColourIndex];
      int pawnOffset = (mg->friendlyColour == White) ? 8 : -8;
      int startRank = (mg->board->WhiteToMove) ? 1 : 6;
      int finalRankBeforePromotion = (mg->board->WhiteToMove) ? 6 : 1;

      int enPassantFile = ((int) mg->board->currentGameState.epSquares - 1);
      int enPassantSquare = -1;

      Move m;


      if (enPassantFile != -1) {
            enPassantSquare = 8 * ((mg->board->WhiteToMove) ? 5 : 2) +
enPassantFile;
      } /* if */

      for (i = 0; i < Count(&myPawns); i++) {
            startSquare = myPawns.occupiedSquares[i];
            rank = RankIndex(startSquare);
            oneStepFromPromotion = rank == finalRankBeforePromotion;

            squareOneForward = startSquare + pawnOffset;

            /* Se la casella davanti al pedone e' libera: spinte di pedone
*/
            if (mg->board->Square[squareOneForward] == None) {
                  /* Il pedone non e' inchiodato, o si sta muovendo lungo la
linea di un'inchiodatura */
                  if (!IsPinned(mg, startSquare) || IsMovingAlongRay(mg,
pcm, pawnOffset, startSquare, mg->friendlyKingSquare)) {
                        /* Non si e' sotto scacco, o sta interponendo un
pezzo che da' scacco */
                        if (!mg->inCheck || SquareIsInCheckRay(mg, square-
OneForward)) {
                              if (oneStepFromPromotion) {
                                    MakePromotionMoves(mg,       startSquare,
squareOneForward);
                              } else {
                                    MoveInit(startSquare,   squareOneForward,
0, &m);
                        MAdd(&mg->moves, &m);
                              } /* else */
                        } /* if */

                        /* Se e' nella casella iniziale (dunque puo' muover-
si di due se non e' bloccato) */
                        if (rank == startRank) {
                              squareTwoForward = squareOneForward + pawnOff-
```

```
set;
                              if    (mg->board->Square[squareTwoForward]    ==
None) {
                        /* Non c'e' scacco, o il pedone sta interponendo in
una linea di scacco */
                              if   (!mg->inCheck   ||   SquareIsInCheck-
Ray(mg, squareTwoForward)) {
                              MoveInit(startSquare,          squareTwoForward,
PawnTwoForward, &m);
                              MAdd(&mg->moves, &m);
                                    } /* if */
                              } /* if */
                        } /* if */
                  } /* if */
            } /* if */

            /* Catture di pedone */
            for (j = 0; j < 2; j++) {
                  /* Controlla se esiste la diagonale rispetto al pedone */
                  if                (pcm->numSquaresToEdge[startSquare][pcm-
>pawnAttackDirections[mg->friendlyColourIndex][j]] > 0) {
                        /* Sposto il pedone nella direzione di cattura dei
pedoni del giocatore per ottenere la casella da cui il pedone nemico attac-
cherebbe */
                        pawnCaptureDir      =      pcm->directionOffsets[pcm-
>pawnAttackDirections[mg->friendlyColourIndex][j]];
                        targetSquare = startSquare + pawnCaptureDir;
                        targetPiece = mg->board->Square[targetSquare];

                        /* Se il pedone e' inchiodato, e la casella su cui
si vuole muovere non e' nella stessa linea dell'inchiodatura, salta la dire-
zione */
                        if  (IsPinned(mg,  startSquare)  &&  !IsMovingAlong-
Ray(mg, pcm, pawnCaptureDir, mg->friendlyKingSquare, startSquare)) {
                              continue;
                        } /* if */

                        /* Cattura regolare */
                        if (IsColour(targetPiece, mg->opponentColour)) {
                              /* Se e' in scacco, e il pezzo non sta cattu-
rando/interponendo il pezzo che da' scacco, salto alla casella successiva */
                              if (mg->inCheck && !SquareIsInCheckRay(mg, tar-
getSquare)) {
                                    continue;
                              } /* if */
                              if (oneStepFromPromotion) {
                                    MakePromotionMoves(mg,  startSquare,  tar-
getSquare);
                              } else {
                                    MoveInit(startSquare,   targetSquare,   0,
&m);
                        MAdd(&mg->moves, &m);
                              } /* else */
                        } /* if */
```

```
                        /* Catttura en-passant */
                        if (targetSquare == enPassantSquare) {
                              epCapturedPawnSquare = targetSquare + ((mg-
>board->WhiteToMove) ? -8 : 8);
                              if       (!InCheckAfterEnPassant(mg,       pcm,
startSquare, targetSquare, epCapturedPawnSquare)) {
                                    MoveInit(startSquare,       targetSquare,
EnPassantCapture, &m);
                        MAdd (&mg->moves, &m);
                              } /* if */
                        } /* if */
                  } /* if */
            } /* for */
      } /* for */
} /* GeneratePawnMoves */

/* Mossa di uno sliding piece: torre alfiere regina
// IOP mg puntatore ai dati per le mosse generate che conterra' gli aggior-
namenti delle informazioni sulle caselle controllate dopo la mossa
// IP pcm puntatore ai dati di appoggio
// IP startSquare casella di partenza
// IP startDirIndex direzione di partenza dal directionOffsets del PreCompu-
tedMoveData.h
// IP endDirIndex direzione finale dal directionOffsets del PreComputedMove-
Data.h
*/
void UpdateSlidingAttackPiece (MoveGenerator* mg, PreComputedMoveData* pcm,
int startSquare, int startDirIndex, int endDirIndex) {
    int  directionIndex,  currentDirOffset,  n,  targetSquare,  targetSquare-
Piece;
      for(directionIndex = startDirIndex; directionIndex < endDirIndex; di-
rectionIndex++) {
            currentDirOffset = pcm->directionOffsets[directionIndex];
            for        (n        =        0;        n        <        pcm-
>numSquaresToEdge[startSquare][directionIndex]; n++) {
                  targetSquare = startSquare + currentDirOffset * (n + 1);
                  targetSquarePiece = mg->board->Square[targetSquare];
                  mg->opponentSlidingAttackMap[targetSquare] = 1;
                  if (targetSquare != mg->friendlyKingSquare) {
                        if (targetSquarePiece != None) {
                              break;
                        } /* if */
                  } /* if */
            } /* for */
      } /* for */
} /* UpdateSlidingAttackPiece */

/* Mossa di uno sliding piece: torre alfiere regina
// IOP mg puntatore ai dati per le mosse generate che conterra' le informa-
zioni sulle caselle controllate dai pezzi avversari
// IP pcm puntatore ai dati di appoggio
*/
```

```
void GenSlidingAttackMap (MoveGenerator* mg, PreComputedMoveData* pcm) {
      int i;

      PieceList* enemyRooks = &mg->board->rooks[mg->opponentColourIndex];
      PieceList* enemyQueens = &mg->board->queens[mg->opponentColourIndex];
      PieceList*        enemyBishops        =        &mg->board->bishops[mg-
>opponentColourIndex];
      for(i = 0; i < 64; i++){
         mg->opponentSlidingAttackMap[i] = 0;
    }

      for (i = 0; i < Count(enemyRooks); i++) {
            UpdateSlidingAttackPiece(mg,           pcm,           enemyRooks-
>occupiedSquares[i], 0, 4);
      } /* for */

      for (i = 0; i < Count(enemyQueens); i++) {
            UpdateSlidingAttackPiece(mg,           pcm,           enemyQueens-
>occupiedSquares[i], 0, 8);
      } /* for */

      for (i = 0; i < Count(enemyBishops); i++) {
            UpdateSlidingAttackPiece(mg,          pcm,          enemyBishops-
>occupiedSquares[i], 4, 8);
      } /* for */
} /* GenSlidingAttackMap */

/* Tutte le mosse
// IOP mg puntatore ai dati per le mosse generate che conterra' le informa-
zioni sulle caselle attaccate dall'avversario
// IP pcm puntatore ai dati di appoggio
*/
void CalculateAttackData (MoveGenerator* mg, PreComputedMoveData* pcm) {
      int dir, n, directionOffset, i, j, squareIndex, piece, knightIndex,
pawnIndex, pieceType, startSquare, pawnSquare, enemyKingSquare;
      bool isDiagonal, isFriendlyPieceAlongRay, isKnightCheck;
      int rayMask[64], pawnAttacks[64];

      /* Cerca caselle in tutte le direzioni intorno al re del giocatore per
scacchi/inchiodature dai pezzi dell'avversario (queen, rook, bishop) */
      int startDirIndex = 0;
      int endDirIndex = 8;
      PieceList opponentKnights, opponentPawns;

    GenSlidingAttackMap(mg, pcm);

      if (Count(&(mg->board->queens[mg->opponentColourIndex])) == 0) {
        startDirIndex = (Count(&(mg->board->rooks[mg->opponentColourIndex]))
> 0) ? 0 : 4;
            endDirIndex          =          (Count(&(mg->board->bishops[mg-
>opponentColourIndex])) > 0) ? 8 : 4;
      } /* if */
```

```
      for (dir = startDirIndex; dir < endDirIndex; dir++) {
            isDiagonal = dir > 3;

            n = pcm->numSquaresToEdge[mg->friendlyKingSquare][dir];
            directionOffset = pcm->directionOffsets[dir];
            isFriendlyPieceAlongRay = false;
            for (i = 0; i < 64; i++) {
                rayMask[i] = 0;
        }

            for (i = 0; i < n; i++) {
                  squareIndex = mg->friendlyKingSquare + directionOffset *
(i + 1);
                  /* Aggiungo la casella ai controllati */
                rayMask[squareIndex] = 1;
                  piece = mg->board->Square[squareIndex];

                  /* Questa casella contiene un pezzo */
                  if (piece != None) {
                        if (IsColour (piece, mg->friendlyColour)) {
                              /* Prima un pezzo del giocatore, dobbiamo pro-
seguire perche' potrebbe esserci un pin al re in questa direzione, quindi
essere inchiodato */
                              if (!isFriendlyPieceAlongRay) {
                                    isFriendlyPieceAlongRay = true;
                              } /* if */
                              /* Questo e' il secondo pezzo che si trova dun-
que non puo' essere un pin */
                              else {
                                    break;
                              } /* else */
                        } /* if */
                        /* Contiene un pezzo nemico */
                        else {
                              pieceType = PieceType (piece);

                              /* Controlla se il pezzo e' nella bitmask dei
pezzi capaci di muoversi nella posizione corrente */
                              if ((isDiagonal && IsBishopOrQueen(pieceType))
|| (!isDiagonal && IsRookOrQueen(pieceType))) {
                                    /* Un pezzo del giocatore blocca lo scac-
co, quindi e' un pin */
                                    if (isFriendlyPieceAlongRay) {
                                          mg->pinsExistInPosition = true;
                                          for(j = 0; j < 64; j++){
                                mg->pinRayBitmask[j] = rayMask[j];
                                }
                                    } /* if */
                                    /* Nessun pezzo a bloccarlo, dunque uno
scacco */
                                    else {
```

```
                                        for(j = 0; j < 64; j++) {
                                            if(rayMask[j] == 1)
                                        mg->checkRayBitmask[j] = rayMask[j];
                                    }
                                        mg->inDoubleCheck = mg->inCheck; /*
Se era gia' sotto scacco allora e' uno scacco doppio */
                                        mg->inCheck = true;
                                    } /* else */
                                    break;
                                } else {
                                    /* Questo pezzo nemico non puo' muoversi
nella direzione corrente, e quindi sta bloccando scacchi/inchiodature */
                                    break;
                                } /* else */
                            } /* else */
                        } /* if */
                    } /* for */
                    /* Smette di cercare inchiodature nel caso di doppio scacco, in
quanto il re e' l'unico pezzo in grado di muoversi in questo caso */
                    if (mg->inDoubleCheck) {
                        break;
                    } /* if */

            } /* for */

            /* attacco dei cavalli */
            opponentKnights = mg->board->knights[mg->opponentColourIndex];
            for(j = 0; j < 64; j++) {
                mg->opponentKnightAttacks[j] = 0;
        }
            isKnightCheck = false;

            for (knightIndex = 0; knightIndex < Count(&opponentKnights); knightIn-
dex++) {
                    startSquare = opponentKnights.occupiedSquares[knightIndex];
                    for(j = 0; j < 64; j++) {
                    if((((pcm->knightAttackBitboards)[startSquare])[j]) == 1)
                        mg->opponentKnightAttacks[j]             =             pcm-
>knightAttackBitboards[startSquare][j];
                }
                    if (!isKnightCheck && ContainsSquare(mg->opponentKnightAttacks,
mg->friendlyKingSquare)) {
                        isKnightCheck = true;
                        mg->inDoubleCheck = mg->inCheck; /* Se gia' sotto scacco,
diventa doppio scacco */
                        mg->inCheck = true;
                        mg->checkRayBitmask[startSquare] = 1;
                    } /* if */
            } /* for */

            /* Attacco di pedoni */
            opponentPawns = mg->board->pawns[mg->opponentColourIndex];
```

```
      for(j = 0; j < 64; j++) {
          mg->opponentPawnAttackMap[j] = 0;
    }
      bool isPawnCheck = false;

      for (pawnIndex = 0; pawnIndex < Count(&opponentPawns); pawnIndex++) {
            pawnSquare = opponentPawns.occupiedSquares[pawnIndex];
            for(j = 0; j < 64; j++) {
                pawnAttacks[j]   =   pcm->pawnAttackBitboards[pawnSquare][mg-
>opponentColourIndex][j];
        }
            for(j = 0; j < 64; j++) {
              if(pawnAttacks[j] == 1)
                mg->opponentPawnAttackMap[j] = pawnAttacks[j];
        }

            if     (!isPawnCheck     &&     ContainsSquare(pawnAttacks,     mg-
>friendlyKingSquare)) {
                 isPawnCheck = true;
                 mg->inDoubleCheck = mg->inCheck;
                 mg->inCheck = true;
                 for(j = 0; j < 64; j++){
                     mg->checkRayBitmask[pawnSquare] = 1;
            }
            } /* if */
      } /* for */

      enemyKingSquare = mg->board->KingSquare[mg->opponentColourIndex];
    for(j = 0; j < 64; j++){
      if(mg->opponentSlidingAttackMap[j] == 1)
            mg->opponentAttackMapNoPawns[j]              =              mg-
>opponentSlidingAttackMap[j];
        else if(mg->opponentKnightAttacks[j] == 1)
            mg->opponentAttackMapNoPawns[j] = mg->opponentKnightAttacks[j];
        else
            mg->opponentAttackMapNoPawns[j]              =              pcm-
>kingAttackBitboards[enemyKingSquare][j];

        if(mg->opponentAttackMapNoPawns[j] == 1)
         mg->opponentAttackMap[j] = mg->opponentAttackMapNoPawns[j];
      else
         mg->opponentAttackMap[j] = mg->opponentPawnAttackMap[j];
      } /* for */
} /* CalculateAttackData */

/* Genera una lista di tutte le possibili mosse nella corrente posizione.
// IOP mg puntatore ai dati per le mosse generate che conterra' le informa-
zioni sulle caselle attaccate dall'avversario
// IP pcm puntatore ai dati di appoggio
// IP board scacchiera che da' la posizione da cui calcolare la posizione */
MNodo* GenerateMoves(MoveGenerator* mg, PreComputedMoveData* pcd,   Board*
board) {
```

```
    mg->moves = NULL;
    mg->board = board;
      Init(mg);
      CalculateAttackData(mg, pcd);
      GenerateKingMoves(mg, pcd);

      /* Solo le mosse di re sono possibili in uno scacco doppio dunque pos-
so ritornare prima. */
      if (mg->inDoubleCheck) {
            return mg->moves;
      } /* if */

      GenerateSlidingMoves(mg, pcd);
      GenerateKnightMoves(mg, pcd);
      GeneratePawnMoves(mg, pcd);
      return mg->moves;
} /* GenerateMoves */
```

Il file Tester.c contiene le funzioni che permettono di giocare contro il motore o contro un altro giocatore.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "MoveGenerator.h"

int main(int args, char **argv) {
    Board b;
    Coord coord;
    MoveGenerator mg;
    PreComputedMoveData pcm;
    MNodo* moveList;
    Move* moveArray;
    /*char movenList[3]; */
    int i, n;
    int startSquare = 0, toSquare = 0;
    char fen[88] = "";
    char move[8];
    bool endGame = false;
    Move m;
    srand(time(NULL));
    PrecomputedMoveData(&pcm);
    LoadStartPosition(&b);
    drawChessBoard(&b);
    while(true) { /* esce con la fine della partita */
        startSquare = 0;
        toSquare = 0;
        CurrentFen(&b, fen);
        moveList = GenerateMoves(&mg, &pcm, &b);
        if(moveList == NULL && mg.inCheck) {
            printf("\nLa partita e' finita per scacco matto!");
            break;
        } else if (moveList == NULL) {
            printf("\nLa partita e' finita in stallo!");
            break;
        } /* if */
        moveArray = MToArray(moveList);
        moveList = NULL;
        MClear(moveList);

        if(b.plyCount % 2 == 0) {
            printf("\nIl FEN e': %s\n", fen);
            /*printf("\n-----Mosse Legali:");
            for(i = 1; i <= moveArray[0].fromSquare; i++) {
                if(i % 4 == 1) printf("\n");
                SquareNameFromTwoCoordi-
nate(FileIndex(moveArray[i].fromSquare), RankIndex(moveArray[i].fromSquare),
movenList);
```

```
                printf("%s-", movenList);
                SquareNameFromTwoCoordi-
nate(FileIndex(moveArray[i].toSquare),      RankIndex(moveArray[i].toSquare),
movenList);
                printf("%s-%d\t\t", movenList, moveArray[i].flag);
            } for */
            printf("\nInserire una mossa nel formato casella-casella-flag:
");
            gets(move);

            coord.fileIndex = move[0]-97;
            coord.rankIndex = (move[1]-'0') - 1;
            SquareNameFromCoordinate(&coord, move);
            startSquare          =          IndexFromTwoCoord(coord.fileIndex,
coord.rankIndex);
            coord.fileIndex = move[3]-97;
            coord.rankIndex = (move[4]-'0') - 1;
            SquareNameFromCoordinate (&coord, move);
            toSquare = IndexFromTwoCoord(coord.fileIndex, coord.rankIndex);

            MoveInit(startSquare, toSquare, (move[6]-'0'), &m);
            endGame = true;
            for(i = 1; i <= moveArray[0].fromSquare; i++) {
                if(SameMove(&moveArray[i], &m)) {
                    endGame = false;
                    break;
                }
            } /* for */
            if(endGame) {
                printf("\nHai realizzato una mossa illegale, quindi hai per-
so la partita");
                break;
            } /* if */
            MakeMove(&b, &m);
            drawChessBoard(&b);
        } /* if */
        else {
            n = (rand() % moveArray[0].fromSquare) + 1;
            printf("\nIl motore gioca la mossa: ");
            Name(&moveArray[n]);
            MakeMove(&b, &moveArray[n]);
            drawChessBoard(&b);
        }
        free(moveArray);
        moveArray = NULL;
        strcpy(fen, "");
    }/* while */
    free(b.gameStateHistory.stack);
    return 0;
} /* main */
```