```
////////////////////////////////////////////////////////////////////////////
// C++ source code of GreenChessEngine.h Version 5.1.2 at greenchess.com //
////////////////////////////////////////////////////////////////////////////

// incorporate needed system files //
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>
#include <bitset>
#include <cmath>

// class representing a transposition table element //
class ChessTranspositionTableElement
{
   // the age of the transposition table element //
   int Age;

   // the score of the best move //
   int Score;

   // the game tree depth to which the position was searched //
   int Depth;

   // the best move for this particular board position //
   int BestMove;

   // whether this was a score from an alpha-beta cutoff or not //
   bool CutoffOccurred;

   // the zobrist hash value //
   unsigned long long ZobristHash;

   // public functions //
   public:

      // get the age of the transposition table element //
      int GetAge();

      // get the score from the transposition data //
      int GetScore();

      // get the search depth from the transposition table //
      int GetDepth();

      // get the best move for the specified board position //
      int GetBestMove();

      // get whether this score was from a cutoff or not //
      bool WasCutoff();

      // store the transposition table data //
      void StoreTransData(unsigned long long, int, int, bool, int);

      // age the transposition table element //
      void AgeElement(int AgeChange);

      // get the zobrish hash value //
      unsigned long long GetZobristHash();
};

// class representing a chess opening book element //
class ChessOpeningBookElement
{
   // the next move for this position in the opening book //
   std::string NextMove;

   // the zobrist hash value for the specified board position //
   unsigned long long ZobristHash;

   // public functions //
   public:

      // store the next move data and zobrist hash into an opening book element //
      void StoreOpeningBookData(unsigned long long, std::string);

      // get the next move in the opening book for the specified board position //
      std::string GetNextMove();

      // get the zobrist hash represent the specified board position //
      unsigned long long GetZobristHash();
};

// a class representing a chess opening book //
class ChessOpeningBook
{
   // whether or not a next move already exists in the chess opening book //
   bool DataAlreadyExists;

   // an index in a list //
   long Index;

   // a chess opening book element //
   ChessOpeningBookElement OpeningBookElement;

   // a vector of opening book elements //
   std::vector<ChessOpeningBookElement> OpeningBookElements;

   // an iterator to a list of opening book elements //
   std::vector<ChessOpeningBookElement>::iterator ElementIter;

   // a chess opening book //
   std::vector<std::vector<ChessOpeningBookElement> > OpeningBook;

   // an iterator to a chess opening book //
   std::vector<std::vector<ChessOpeningBookElement> >::iterator BookIter;

   // public functions //
   public:

      // whether a next move exists for the specified board position //
      bool MoveExists(unsigned long long);

      // initialize the attributes for the chess opening book //
      void InitializeAttributes();

      // store the next move and zobrist hash into the chess opening book //
      void StoreOpeningBookData(unsigned long long, std::string);

      // get the next move for the specified board position in the opening book //
      std::string GetNextMove();
};

// class representing a transposition table //
class ChessTranspositionTable
{
   // the count of the size of the transposition table //
   int Count;

   // whether found stored data in the transposition table or not //
   bool FoundStoredData;

   // an index in a list //
   long Index;

   // a transposition table element //
   ChessTranspositionTableElement TableElement;

   // a list of elements for a location in the transposition table //
   std::vector<ChessTranspositionTableElement> ElementList;

   // an iterator to a chess transposition element list //
   std::vector<ChessTranspositionTableElement>::iterator ElementIter;

   // the transposition table //
   std::vector<std::vector<ChessTranspositionTableElement> > Table;

   // an iterator to a transposition table //
   std::vector<std::vector<ChessTranspositionTableElement> >::iterator TableIter;

   // remove the old elements from the transposition table //
   void RemoveOldElements();

   // age the transposition table //
   void AgeTable(int AgeChange);

   // public functions //
   public:

      // get the score for the corresponding zobrist hash //
      int GetScore();

      // get the best move for the specified board position //
      int GetBestMove();

      // get the size of the transposition table //
      int TableSize();

      // if score caused an alpha-beta cutoff //
      bool WasCutoff();

      // if score didn't cause an alpha-beta cutoff //
      bool NotCutoff();

      // get whether there is a saved score corresponding to the zobrist hash or not //
      bool BoardPositionWasPreviouslyStored(unsigned long long);

      // determine if data stored in transposition table was searched deep enough //
      bool DataSearchedDeepEnough(int, int);

      // initialize the transposition table attributes //
      void InitializeAttributes();

      // process the transpostion table data //
      void ProcessTransData(unsigned long long, int, int, bool, int);

      // compact the transposition table //
      void CompactTransTable(bool AgeForward);
};

// class representing a chess piece //
class ChessPiece
{
   // a zobrist key index //
   short ZobristKeyIndex;

   // index location of piece in piece list //
   short Index;

   // rank coordinate of piece on board //
   short Rank;

   // new rank coordinate of piece on board //
   short NewRank;

   // file coordinate of piece on board //
   short File;

   // new file coordinate of piece on board //
   short NewFile;

   // 0=pawn, 1=knight, 2=bishop, 3=rook, 4=queen, 5=king, 6=pseudo king //
   short Type;

   // the mobility of the piece //
   short Mobility;

   // material value of piece //
   short Value;

   // boolean values representing piece attributes //
   std::bitset<12> Booleans;

   // directions along which pinned piece may move //
   std::bitset<16> Pins;

   // directions along which piece is attacked //
   std::bitset<16> Attacks;

   // set the zobrist key index for the piece //
   void SetZobristKeyIndex();

   // set rank and file coordinates of piece on board //
   void SetCoordinates(short);

   // assign material value of piece relevant to the piece type //
   void AssignMaterial();

   // initialize piece attributes //
   void InitializeAttributes(short, bool, short, short, bool);

   // if piece is guarded by a pawn //
   bool GuardedByPawn();

   // if piece is not attacked by a pawn //
   bool NotAttackedByPawn();

   // if piece has not moved valid number of squares from starting position //
   bool IllegalLength(short, short);

   // the passed pawn multiplier for increasing the value of a passed pawn //
   short PassedPawnMultiplier();

   // the centeredness of a piece on the board for determining development //
   short Centeredness();

   // return positive one if piece is white and negative one if black //
   short ColorSign();

   // descriptive upper case symbol for piece //
   std::string UpperPieceSymbol();

   // convert integer to string //
   std::string IntToString(int);

   // public functions //
   public:

      // increase the mobility of the piece //
      void IncreaseMobility();

      // mark piece as having already moved //
      void MarkMoved();

      // set doubled pawn status //
      void SetDoubledPawnStatus(bool);

      // set tripled pawn status //
      void SetTripledPawnStatus(bool);

      // set passed pawn status //
      void SetPassedPawnStatus(bool);

      // set isolated pawn status //
      void SetIsolatedPawnStatus(bool);

      // mark piece as being guarded by a pawn //
      void MarkGuardedByPawn();

      // mark piece as being attacked by a pawn //
      void MarkAttackedByPawn();

      // mark that piece causes a check //
      void CauseCheck();

      // promote pawn to a knight //
      void PromotePawnToKnight();

      // promote pawn to a bishop //
      void PromotePawnToBishop();

      // promote pawn to a rook //
      void PromotePawnToRook();

      // promote pawn to a queen //
      void PromotePawnToQueen();

      // deactivate piece //
      void DeactivatePiece();

      // create a white pawn //
      void CreateWhitePawn(short, short);

      // create a white knight //
      void CreateWhiteKnight(short, short);

      // create a white bishop //
      void CreateWhiteBishop(short, short);

      // create a white rook //
      void CreateWhiteRook(short, short);

      // create a white queen //
      void CreateWhiteQueen(short, short);

      // create a white king //
      void CreateWhiteKing(short, short);

      // create a white pseudo king for determining which moves can cause a check //
      void CreateWhitePseudoKing(short, short);

      // create a black pawn //
      void CreateBlackPawn(short, short);

      // create a black knight //
      void CreateBlackKnight(short, short);

      // create a black bishop //
      void CreateBlackBishop(short, short);

      // create a black rook //
      void CreateBlackRook(short, short);

      // create a black queen //
      void CreateBlackQueen(short, short);

      // create a black king //
      void CreateBlackKing(short, short);

      // create a black pseudo king for determining which moves can cause a check //
      void CreateBlackPseudoKing(short, short);

      // reinitialize piece attributes //
      void ReinitializeAttributes(bool);

      // make piece eligible to be captured en passant //
      void MakeEnpassantEligible();

      // make piece ineligible to be captured en passant //
      void MakeEnpassantIneligible();

      // update position of piece on board //
      void UpdatePosition(short);

      // shift file coordinate of piece on board //
      void ShiftFile(short);

      // mark that piece is attacked along this direction //
      void DirectionallyAttack(short);

      // mark directions along which pinned piece can move //
      void DirectionallyPin(short);

      // whether piece is currently the side to move //
      bool SideMoving();

      // whether piece is not currently the side to move //
      bool NotSideMoving();

      // if piece is white //
      bool IsWhite();

      // if piece is black //
      bool IsBlack();

      // if is a pawn eligible to promote //
      bool PawnCanPromote();

      // if piece is available to be captured en passant //
      bool IsEnpassant();

      // determine if piece causes a check //
      bool IsChecker();

      // if piece is active //
      bool IsActive();

      // if piece is not active //
      bool NotActive();

      // if piece is being attacked along this direction of movement //
      bool DirectionallyAttacked(short);

      // determine if piece moves and violates its pinned status //
      bool ViolatesPin(short);

      // if piece has never moved //
      bool NeverMoved();

      // determine if piece is moving along an allowed direction //
      bool AllowedDirection(short);

      // if piece is a pseudo king for determining which moves cause a check //
      bool IsPseudoKing();

      // determine if piece is a pawn //
      bool IsPawn();

      // determine if piece is a knight //
      bool IsKnight();

      // determine if piece is a bishop //
      bool IsBishop();

      // determine if piece is a rook //
      bool IsRook();

      // determine if piece is a queen //
      bool IsQueen();

      // determine if piece is a king //
      bool IsKing();

      // get the piece color //
      bool PieceColor();

      // whether piece is on a white square or not //
      bool OnWhiteSquare();

      // the material value of the piece //
      short PieceValue();

      // get a zobrist key index //
      short GetZobristKeyIndex();

      // rank coordinate of piece //
      short PieceRank();

      // file coordinate of piece //
      short PieceFile();

      // get type of piece //
      short PieceType();

      // get next position of piece during move simulation //
      short NextPosition(short, short);

      // get position of piece on board //
      short PiecePosition();

      // get position of square one rank behind an en passant eligible pawn //
      short EnpassantPiecePosition();

      // get index of piece in piece list //
      short PieceIndex();

      // points for pieces on outposts //
      short GetOutpostPoints();

      // points for isolated pawns //
      short GetIsolatedPawnPoints();

      // points for passed pawns //
      short GetPassedPawnPoints();

      // points for doubled pawns //
      short GetDoubledPawnPoints();

      // points for tripled pawns //
      short GetTripledPawnPoints();

      // points given for piece mobility //
      short GetMobilityPoints(short PiecesLeft);

      // points given for piece development //
      short GetDevelopedPoints(short PiecesLeft);

      // get points given for material value of piece //
      short GetMaterialPoints();

      // pgn symbol for piece //
      std::string PGNSymbol();

      // descriptive board location of piece //
      std::string BoardLocation();

      // the descriptive board location of the enpassant target square //
      std::string EnpassantBoardLocation();

      // name of file //
      std::string FileName();

      // name of rank //
      std::string RankName();

      // descriptive lower case symbol for piece //
      std::string LowerPieceSymbol();
};

// class representing a move by a chess piece //
class ChessMove
{
   // identification label for move //
   int Label;

   // whether move can cause a check or not //
   bool CanCheck;

   // whether move does an en passant pawn capture //
   bool Enpassant;

   // the specific type of pgn label to use for the move due to ambiguities //
   short PGNSelection;

   // the original value of the pgn label to use for the move due to ambiguities //
   short OriginalPGNSelection;

   // whether move castles (-1 = no castle, 0 = kingside, 1 = queenside) //
   short Castle;

   // if causes a check (-1 = none, 0 = check, 1 = checkmate, 2 = stalemate) //
   short Check;

   // the distance the piece has traveled for this move along a given path //
   short Length;

   // the direction along which the piece is moving (0 to 15) //
   short Direction;

   // the piece that is moving //
   ChessPiece Piece;

   // updated piece at new position //
   ChessPiece UpdatedPiece;

   // the captured piece if a capture occurs //
   ChessPiece CapturedPiece;

   // a pgn label from a string //
   std::string PGNLabelString;

   // if a check has occurred //
   bool IsCheck();

   // if a checkmate has occurred //
   bool IsCheckmate();

   // pgn description of capture //
   std::string PGNCapture();

   // pgn description of whether in check or checkmate //
   std::string PGNCheck();

   // pgn description of pawn promotion //
   std::string PGNPromotion();

   // increase the detail of the pgn description //
   std::string PGNDetail();

   // promotion symbol of piece //
   std::string PromotionSymbol();

   // public functions //
   public:

      // the square the captured piece is on if there was a captured piece //
      int CapturedPieceSquare();

      // the material value of the piece //
      int PieceValue();

      // the material value of the piece captured //
      int CapturedPieceValue();

      // get the move identification label //
      int MoveLabel();

      // get direction piece is moving in //
      int MoveDirection();

      // get index of moving piece in piece list //
      int PieceIndex();

      // get index of captured piece in piece list //
      int CapturedPieceIndex();

      // get the updated position of piece after it has moved //
      int UpdatedPosition();

      // mark that move can cause a check if moves to certain square //
      void MarkCanCheck();

      // initialize pgn information in move //
      void InitializePGN();

      // set the level of detail will need in the pgn label for the move //
      void SetPGNDetail(int);

      // mark that move causes a check //
      void CauseCheck();

      // mark that move causes a checkmate //
      void CauseCheckmate();

      // mark that move causes a stalemate //
      void CauseStalemate();

      // record that captures a piece en passant //
      void CaptureEnpassant();

      // promote pawn to a knight //
      void PromotePawnToKnight();

      // promote pawn to a bishop //
      void PromotePawnToBishop();

      // promote pawn to a rook //
      void PromotePawnToRook();

      // promote pawn to a queen //
      void PromotePawnToQueen();

      // make identification label for move //
      void MakeLabel();

      // store the captured piece //
      void StoreCapturedPiece(ChessPiece);

      // initialize move attributes //
      void InitializeAttributes(ChessPiece, int, int, int);

      // mark piece as being eligible to be captured en passant //
      void MakeEnpassantEligible();

      // mark move as being a kingside castle //
      void CastleKingside();

      // mark move as being a queenside castle //
      void CastleQueenside();

      // whether a king has been potentially captured or not //
      bool KingIsCaptured();

      // whether move is the side currently to move //
      bool SideMoving();

      // whether move is not the side currently to move //
      bool NotSideMoving();

      // if is a pawn eligible to promote //
      bool PawnCanPromote();

      // determine if move made is an en passant pawn capture //
      bool IsEnpassant();

      // whether move is a kingside castle //
      bool IsKingsideCastle();

      // whether move is a queenside castle //
      bool IsQueensideCastle();

      // determine if piece captures another piece in this move //
      bool IsCapture();

      // whether move can cause a check or not //
      bool CanCauseCheck();

      // determine if piece does not capture another piece //
      bool NotCapture();

      // determine if move attacks square that piece moves to //
      bool IsAttack();

      // determine if move results in a pawn promotion //
      bool IsPawnPromotion();

      // get color of piece //
      bool PieceColor();

      // determine if piece moving is a pawn //
      bool IsPawn();

      // determine if piece moving is a bishop //
      bool IsBishop();

      // determine if piece moving is a rook //
      bool IsRook();

      // determine if piece moving is a queen //
      bool IsQueen();

      // determine if piece moving is a king //
      bool IsKing();

      // determing if piece moving is a knight //
      bool IsKnight();

      // determine if move is white //
      bool IsWhite();

      // determine if move is black //
      bool IsBlack();

      // retrieve the updated piece for the move //
      ChessPiece RetrieveUpdatedPiece();

      // get uci label of move //
      std::string UCILabel();

      // pgn label in portable game notation //
      std::string PGNLabel();

      // the version of the pgn label to use for the move due to pgn ambiguities //
      std::string PGNLabelVersion(int PGNVersion);
};

// class representing a square on the board //
class ChessSquare
{
   // whether piece needs to move to this square to block a check //
   bool Block;

   // the number of attacks by white on this square //
   short AttacksByWhite;

   // the number of attacks by black on this square //
   short AttacksByBlack;

   // the piece on this square //
   ChessPiece Piece;

   // the en passant pseudo piece on this square //
   ChessPiece EnpassantPiece;

   // the pieces that can cause a check on this square //
   std::bitset<6> PiecesThatCauseCheck;

   // public functions //
   public:

      // the file that the piece that can be captured en passant is on //
      int EnpassantFile();

      // get a zobrist key index for active piece on the square //
      int GetZobristKeyIndex();

      // the position of the piece on square //
      int PiecePosition();

      // the piece list index of the piece //
      int PieceIndex();

      // get the number of attacks on this square by the specified color //
      int NumberOfAttacksByColor(bool AttackColor);

      // add an attack on this square by the specified color //
      void AddAttackToSquare(bool PieceColor);

      // mark square as requiring a piece to move here to block a check //
      void MarkBlock();

      // initialize square attributes //
      void InitializeAttributes();

      // deactivate piece on square //
      void DeactivatePiece();

      // store occupying piece on square //
      void StorePiece(ChessPiece);

      // store the pseudo en passant piece on square //
      void StoreEnpassantPiece(ChessPiece);

      // mark that a pawn can cause a check on this square //
      void MarkPawnCanCauseCheck();

      // mark that a rook can cause a check on this square //
      void MarkRookCanCauseCheck();

      // mark that a knight can cause a check on this square //
      void MarkKnightCanCauseCheck();

      // mark that a queen can cause a check on this square //
      void MarkQueenCanCauseCheck();

      // mark that a bishop can cause a check on this square //
      void MarkBishopCanCauseCheck();

      // determine whether a piece causes a check on this square or not //
      bool PieceCanCheckHere(int Type);

      // if piece on square is a pawn //
      bool IsPawn();

      // if piece on square is a rook //
      bool IsRook();

      // if piece on square is a king //
      bool IsKing();

      // if piece on square is black //
      bool IsBlack();

      // if piece on square is white //
      bool IsWhite();

      // whether piece does not need to block a check here //
      bool NotBlock();

      // determine if square is occupied by a piece //
      bool IsOccupied();

      // determine if square is occupied by a pseudo en passant piece //
      bool IsEnpassantOccupied();

      // determine if square is not occupied by a piece //
      bool IsUnoccupied();

      // determine if occupying piece has never moved //
      bool NeverMoved();

      // determine if piece on square is attacked //
      bool IsAttacked(bool PieceColor);

      // determine if piece on square is not attacked //
      bool NotAttacked(bool PieceColor);

      // get the color of the piece on the square //
      bool PieceColor();

      // get the color of the pseudo en passant piece on the square //
      bool EnpassantPieceColor();

      // retrieve the piece occupying the square //
      ChessPiece RetrievePieceOnSquare();

      // retrieve the pseudo en passant piece occupying the square //
      ChessPiece RetrieveEnpassantPiece();
};

// class representing the chessboard //
class ChessBoard
{
   // the number of occurrences of a given condition //
   int Occurrences;

   // the points for having castled //
   int CastlingPoints;

   // the total points for having shielding pawns in front of the castled king //
   int TotalKingPawnShieldPoints;

   // the points for having shielding pawns in front of the castled king //
   int KingPawnShieldPoints;

   // the total points for attacks in the king zone //
   int TotalAttacksInKingZonePoints;

   // the points for attacks in the king zone //
   int AttacksInKingZonePoints;

   // the total points for friendly guards of squares in the king zone //
   int TotalGuardsInKingZonePoints;

   // the points for friendly guards of squares in the king zone //
   int GuardsInKingZonePoints;

   // the total points for enemy pieces occupying the king zone //
   int TotalEnemyPiecesInKingZonePoints;

   // the points for enemy pieces occupying the king zone //
   int EnemyPiecesInKingZonePoints;

   // the total points for friendly pieces occupying the king zone //
   int TotalFriendlyPiecesInKingZonePoints;

   // the points for friendly pieces occupying the king zone //
   int FriendlyPiecesInKingZonePoints;

   // points for the bishop and rook pairs //
   int BishopRookPairPoints;

   // points for pieces on outpost squares in enemy territory //
   int OutpostPoints;

   // points for material on the board //
   int MaterialPoints;

   // points for mobility of the kings //
   int KingMobilityPoints;

   // points for mobility of the queens //
   int QueenMobilityPoints;

   // points for mobility of the rooks //
   int RookMobilityPoints;

   // points for mobility of the knights //
   int KnightMobilityPoints;

   // points for mobility of the bishops //
   int BishopMobilityPoints;

   // points for development of the kings //
   int KingDevelopmentPoints;

   // points for development of the queens //
   int QueenDevelopmentPoints;

   // points for development of the rooks //
   int RookDevelopmentPoints;

   // points for development of the knights //
   int KnightDevelopmentPoints;

   // points for development of the bishops //
   int BishopDevelopmentPoints;

   // points for development of the pawns //
   int PawnDevelopmentPoints;

   // points for passed pawns //
   int PassedPawnPoints;

   // points for isolated pawns //
   int IsolatedPawnPoints;

   // points for doubled pawns //
   int DoubledPawnPoints;

   // points for tripled pawns //
   int TripledPawnPoints;

   // the number of white bishops on the board //
   int NumberOfWhiteBishops;

   // the number of black bishops on the board //
   int NumberOfBlackBishops;

   // the number of white bishops on white squares //
   int NumberOfWhiteBishopsOnWhite;

   // the number of black bishops on white squares //
   int NumberOfBlackBishopsOnWhite;

   // the number of white bishops on black squares //
   int NumberOfWhiteBishopsOnBlack;
   // the number of black bishops on black squares //
   int NumberOfBlackBishopsOnBlack;

   // the number of white rooks on the board //
   int NumberOfWhiteRooks;

   // the number of black rooks on the board //
   int NumberOfBlackRooks;

   // the number of white knights on the board //
   int NumberOfWhiteKnights;

   // the number of black knights on the board //
   int NumberOfBlackKnights;

   // the number of white queens on the board //
   int NumberOfWhiteQueens;

   // the number of black queens on the board //
   int NumberOfBlackQueens;

   // the number of white pawns on the board //
   int NumberOfWhitePawns;

   // the number of black pawns on the board //
   int NumberOfBlackPawns;

   // the number of white pawns on each file //
   int WhitePawnFileCounts[8];

   // the number of black pawns on each file //
   int BlackPawnFileCounts[8];

   // the number of pieces left on the board //
   int PiecesLeft;

   // the pawn count on a specified file //
   int PawnCount;

   // a minimum rank on the board to use in a calculation //
   int MinRank;

   // a maximum rank on the board to use in a calculation //
   int MaxRank;

   // a rank on the board //
   int Rank;

   // a file on the board //
   int File;

   // the file of a king on the board //
   int KingFile;

   // the rank of a king on the board //
   int KingRank;

   // the first rank in front of the king //
   int KingRankPlusOne;

   // second rank in front of the king //
   int KingRankPlusTwo;

   // a list index //
   int Index;

   // a list index //
   int Index2;

   // position of piece on board //
   int Position;

   // direction piece is moving along //
   int Direction;

   // length of path traveled by piece //
   int Length;

   // what state of check currently have //
   int Check;

   // the index of the white king in the piece list //
   int WhiteKingIndex;

   // the index of the black king in the piece list //
   int BlackKingIndex;

   // whether the king was potentially captured or not //
   bool KingCaptured;

   // whether found a previous zobrist hash for determining threefold repetition //
   bool FoundPreviousZobristHash;

   // whether has the option to kingside castle //
   bool KingsideCastlingOption;

   // whether has the option to queenside castle //
   bool QueensideCastlingOption;

   // whether white has the option to castle kingside //
   bool WhiteKingsideCastlingOption;

   // whether white has the option to castle queenside //
   bool WhiteQueensideCastlingOption;

   // whether black has the option to castle kingside //
   bool BlackKingsideCastlingOption;

   // whether black has the option to castle queenside //
   bool BlackQueensideCastlingOption;

   // whether white has castled kingside //
   bool WhiteCastledKingside;

   // whether black has castled kingside //
   bool BlackCastledKingside;

   // whether white has castled queenside //
   bool WhiteCastledQueenside;

   // whether black has castled queenside //
   bool BlackCastledQueenside;

   // determines if move captures a piece causing a check //
   bool CapturesChecker;

   // whether done moving a piece along the specified path to generate moves //
   bool DoneMoving;

   // whether have a special pin //
   bool SpecialPin;

   // color of side currently moving //
   bool Color;

   // a move made on the board //
   ChessMove MadeMove;

   // a move on the board by a piece //
   ChessMove Move;

   // a chess piece on the board //
   ChessPiece Piece;

   // an element in a FEN notation string //
   std::string FENElement;

   // a FEN notation string //
   std::string FEN;

   // the pgn label for the given pgn detail //
   std::string PGNLabelVersion;

   // list of squares where a capture can occur //
   std::string ListOfCaptureSquares;

   // list of square elements that are between king and attacker //
   std::vector<int> Path;

   // the score and path down the game tree vector //
   std::vector<int> Score;

   // the zobrist hash value //
   unsigned long long ZobristHash;

   // a string iterator //
   std::string::iterator StringIter;

   // list of moves //
   std::vector<ChessMove> MoveList;

   // a temporary list of moves //
   std::vector<ChessMove> TempMoveList;

   // list of pieces //
   std::vector<ChessPiece> PieceList;

   // list of pieces encountered along path piece is moving along //
   std::vector<ChessPiece> PathPieces;

   // list of squares on board //
   std::vector<ChessSquare> SquareList;

   // a list of all possible pgn labels used to resolve pgn ambiguities //
   std::vector<std::string> PGNLabels;

   // the zobrist hash keys //
   std::vector<unsigned long long> ZobristKeys;

   // a list of zobrist hashes for use in determining if have threefold repetition //
   std::vector<unsigned long long> ZobristHashes;

   // a list of the number of times each zobrist hash has occurred so far //
   std::vector<unsigned long long> ZobristHashCounts;

   // iterator to a move //
   std::vector<ChessMove>::iterator MoveIter;

   // iterator to a piece //
   std::vector<ChessPiece>::iterator PieceIter;

   // iterator to a square //
   std::vector<ChessSquare>::iterator SquareIter;

   // iterator to a square that the king is on //
   std::vector<ChessSquare>::iterator KingSquare;

   // iterator to a square that the rook is on //
   std::vector<ChessSquare>::iterator RookSquare;

   // iterator to a pgn label vector //
   std::vector<std::string>::iterator PGNIter;

   // get the piece list index of the king currently moving //
   int MovingKingIndex();

   // get the number of pawns on file in rank range for given color and file number //
   int PawnCountForFile(int PawnFile, int MinRank, int MaxRank, bool PawnColor);

   // get the number of pawns on file in rank range for given color and file number //
   int PawnCountForFile(int PawnFile, bool PawnColor);

   // update the zobrist hash counts used for determining threefold repetition //
   void UpdateZobristHashCounts();

   // compute the evaluation function components //
   void ComputeEvaluationComponents();

   // update the position of the pseduo kings to the current king positions //
   void UpdatePseudoKingPositions();

   // compute the points for castling //
   void ComputeCastlingPoints();

   // compute the safety of the king //
   void ComputeKingSafety();

   // compute the attacks and guards in the king zone //
   void ComputeAttacksAndGuardsInKingZone(bool IsWhite);

   // find the king and file rank of the side currently computing for //
   void FindKingFileAndRank(bool IsWhite);

   // compute the king pawn shield points //
   void ComputeKingPawnShield(bool IsWhite);

   // compute the points for having the bishop pair //
   void ComputeBishopRookPairPoints();

   // compute the numbers of each type of piece on the board //
   void ComputeNumbersOfEachPieceType();

   // mark moves that cause check on the board //
   void MarkMovesThatCauseCheck();

   // add the move to the move list if it is legal //
   void AddToMoveListIfLegal();

   // initialize the zobrist keys //
   void InitializeZobristKeys();

   // mark that rook on given square on the board has moved at least once //
   void MarkRookMoved(int, bool);

   // add up points of individual pieces for score calculation //
   void ComputeScore();

   // test if move made causes a check //
   void MadeMoveCheckTest();

   // determine if encountered a king along path piece moves along //
   void KingEncounterTest();

   // determine if have checkmate or stalemate //
   void StalemateCheckmateTest();

   // test legality of moves //
   void TestLegality();

   // build path that piece moves along on board //
   void BuildPath();

   // build paths that pieces move along on board //
   void BuildPaths();

   // make move labels //
   void MakeLabels();

   // mark the path on board where piece has to move onto to block a check //
   void BlockPath();

   // set that in a state of check //
   void SetCheck();

   // generate promotion moves //
   void GeneratePromotionMoves();

   // refresh the layout of the pieces on the board //
   void RefreshLayout();

   // record a move //
   void RecordMove();

   // directionally attack piece if possible //
   void DirectionallyAttack();

   // pin piece if have a pin //
   void PinPiece();

   // generate all possible moves //
   void GenerateMoves();

   // update the current castling rights //
   void UpdateCastlingRights();

   // generate castling moves //
   void GenerateCastlingMoves();

   // create pieces for the board //
   void CreatePieces(std::string, std::string, std::string);

   // update the pieces just created in terms of initial piece movement //
   void UpdateCreatedPieces(std::string);

   // reinitialize piece attributes //
   void ReinitializePieces();

   // determine if have special pin //
   void SpecialPinTest();

   // analyze the encounter of an en passant eligible enemy //
   void AnalyzeEnpassantEnemy();

   // analyze pawn move //
   void AnalyzePawn();

   // analyze move to empty square //
   void AnalyzeEmptySquare();

   // analyze encounter with friendly piece //
   void AnalyzeFriend();

   // analyze encounter with enemy piece //
   void AnalyzeEnemy();

   // analyze move //
   void AnalyzeMove();

   // reinitialize en passant eligibilities //
   void ReinitializeEnpassant();

   // resolve pgn label conflicts //
   void PGNResolve();

   // find pgn ambiguities //
   void PGNAmbiguities();

   // find pgn duplicates for the given pgn detail //
   void PGNDuplicates(int PGNDetail);

   // analyze the pawn structure on the board //
   void AnalyzePawnStructure();

   // compute a castling right for the specified color, king, and rook location //
   bool ComputeCastlingRight(bool PieceColor, int KingLocation, int RookLocation);

   // whether white forfeits a castle or not //
   bool WhiteForfeitsCastle();

   //whether black forfeits a castle or not //
   bool BlackForfeitsCastle();

   // determine if pin violated //
   bool ViolatesPin();

   // determine if move is legal //
   bool LegalMove();

   // determine if encountered a friendly piece //
   bool EncounteredFriend();

   // determine if encountered an enemy piece //
   bool EncounteredEnemy();

   // determine if encountered an enemey piece that is en passant eligible //
   bool EncounteredEnpassantEnemy();

   // test legality of king move //
   bool KingMoveLegality();

   // determine if piece can move to this square //
   bool CanMovePieceToThisSquare();

   // determine if do not currently have check //
   bool NoCheck();

   // determine if queenside castling is possible //
   bool QueensideCastlingPossible();

   // determine if kingside castling is possible //
   bool KingsideCastlingPossible();

   // convert integer to string //
   std::string IntToString(int);

   // public functions //
   public:

      // get the move label //
      int MoveLabel();

      // the number of pieces on the board //
      int NumberOfPiecesLeft();

      // compute the number of pieces left on the board //
      void ComputeNumberOfPiecesLeft();

      // sort the moves to make alpha-beta search more efficient //
      void SortMoves(int, std::vector<int>);

      // compute the zobrist hash for the transposition table //
      void ComputeZobristHash();

      // make a move on the board //
      void MakeMove(int);

      // condense move list to unstable moves only //
      void CondenseMoves();

      // initialize board attributes //
      void InitializeAttributes(std::string, std::string, std::string, std::string);

      // whether a king has potentially been captured or not //
      bool KingIsCaptured();

      // determine if currently have an en passant move //
      bool IsEnpassant();

      // whether the side to move is currently in check //
      bool IsInCheck();

      // whether the side to move is currently not in check //
      bool NotInCheck();

      // determine if currently have check //
      bool IsSingleCheck();

      // determine if currently have double check //
      bool IsDoubleCheck();

      // determine if currently have stalemate //
      bool IsStalemate();

      // determine if currently have checkmate //
      bool IsCheckmate();

      // whether have threefold repetition or not //
      bool IsThreefoldRepetition();

      // whether have insufficient material for a checkmate or not //
      bool InsufficientMaterialToMate();

      // return the current color moving on the board //
      bool ColorMoving();

      // find the move matching the given move string label //
      bool FindMove(std::string);

      // find the move for the given move label //
      bool FindMove(int);

      // determine if white is currently moving //
      bool IsWhite();

      // determine if black is currently moving //
      bool IsBlack();

      // determine if the game is over //
      bool GameOver();

      // retrieve the next move in the move list to simulate //
      bool RetrieveNextMove();

      // determine if the move list is empty //
      bool NoMoves();

      // determine if the move list only contains one possible move //
      bool OnlyOnePossibleMove();

      // pgn label in portable game notation //
      std::string PGNLabel();

      // uci label in universal chess interface notation //
      std::string UCILabel();

      // convert move label to pgn format //
      std::string LabelToPGN(long);

      // get the list of squares where a piece can be captured //
      std::string GetListOfCaptureSquares();

      // get the list of squares that are threatened //
      std::string GetListOfThreatenedSquares();

      // get the score at the game tree leaf //
      std::vector<int> EvaluateLeaf(int);

      // get the leaf score with the tree path in it //
      std::vector<int> GetScoreWithTreePath();

      // get the zobrist hash value representing the board position //
      unsigned long long GetZobristHash();
};

// class representing the computational chess engine //
class ChessEngine
{
   // the best more at the specified game tree depth to use for sorting moves //
   int BestMoveForSorting;

   // number of nodes per second being searched by the engine //
   int SearchRate;

   // the index in a for loop //
   int Index;

   // another index in a for loop //
   int Index2;

   // number of plies deep to search in game tree //
   int Plies;

   // time limit for the engine computation //
   int MaxTime;

   // maximum number of plies the engine will search down to //
   int MaxPlies;

   // the number of transposition table elements used //
   int TransUsed;

   // whether the graphical user interface wants to stop the engine move or not //
   bool StopEngine;

   // whether the engine is time limited or not //
   bool TimeLimited;

   // whether the engine is ply limited or not //
   bool PlyLimited;

   // number of moves computer has searched for current move //
   long Count;

   // starting time value //
   time_t Start;

   // a chessboard //
   ChessBoard Board;

   // the calculated best move in uci format //
   std::string BestMove;

   // the best score vector currently found from the alpha-beta search algorithm //
   std::vector<int> Score;

   // the alpha vector for alpha-beta search //
   std::vector<int> Alpha;

   // the beta vector for alpha-beta search //
   std::vector<int> Beta;

   // a list of moves //
   std::vector<ChessMove> MoveList;

   // a temporary list of moves //
   std::vector<ChessMove> TempMoveList;

   // a transposition table //
   ChessTranspositionTable TransTable;

   // a list of boards //
   std::vector<ChessBoard> BoardList;

   // list of a list of killer moves for alpha-beta move ordering //
   std::vector<std::vector<int> > KillerMoves;

   // an iterator to a move in a move list //
   std::vector<ChessMove>::iterator MoveIter;

   // update the killer moves with the newest move label value //
   void UpdateKillerMoves(int, int);

   // send information to standard output from the engine //
   void InfoToStandardOutput(int, long, int, std::vector<int>, ChessBoard Board, int,
                            int);

   // convert best move in game tree to uci notation //
   void ConvertBestMove(ChessBoard Board);

   // initialize chess engine attributes //
   void InitializeAttributes(ChessBoard, int, int);

   // whether need to stop searching the game tree or not //
   bool NeedToStopSearching();

   // whether can use a score stored in the transposition table or not //
   bool CanUseTranspositionTableScore(int, int, std::vector<ChessBoard>::iterator);

   // negate the score portion of an alpha or beta vector for alpha-beta search //
   std::vector<int> NegateScore(std::vector<int>);

   // alpha-beta search algorithm //
   std::vector<int> AlphaBetaSearch(std::vector<int>, std::vector<int>, std::vector
                                   <ChessBoard>::iterator, int);

   // public functions //
   public:

      // compact the transposition table //
      void CompactTransTable(bool AgeForward);

      // initialize the transposition table //
      void InitializeTransTable();

      // make the chess engine stop if receive stop command from gui //
      void MakeEngineStop();

      // perform a search to find the theoretical best move in uci format //
      std::string FindBestMove(ChessBoard, int, int);
};

// class representing chess game data //
class ChessData
{
   // the date represented as a single number //
   int DateNumber;

   // the total number of moves in a game //
   int TotalMoves;

   // the number of left parentheses in an annotation //
   int LeftParentheses;

   // the number of right parentheses in an annotation //
   int RightParentheses;

   // the number of left brackets in an annotation //
   int LeftBrackets;

   // the number of right brackets in an annotation //
   int RightBrackets;

   // an index in a list //
   int Index;

   // the year the game was played in //
   int Year;

   // the month the game was played in //
   int Month;

   // the day the game was played in //
   int Day;

   // number of move plies //
   int Plies;

   // beginning position in a string //
   int Start;

   // ending position in a string //
   int End;

   // whether black moves first as a result of initial fen data //
   bool BlackMovesFirst;

   // a chessboard //
   ChessBoard Board;

   // a chess engine //
   ChessEngine Engine;

   // substring of the game header //
   std::string HeaderSubstring;

   // game time control field //
   std::string TimeControl;

   // opening field //
   std::string Opening;

   // section field //
   std::string Section;

   // board number field //
   std::string BoardNumber;

   // the white elo field //
   std::string WhiteElo;

   // the black elo field //
   std::string BlackElo;

   // the white uscf field //
   std::string WhiteUSCF;

   // the black uscf field //
   std::string BlackUSCF;

   // the annotator field //
   std::string Annotator;

   // the eco field for encyclopedia of chess openings //
   std::string ECO;

   // the year as a string //
   std::string YearString;

   // the month as a string //
   std::string MonthString;

   // the day as a string //
   std::string DayString;

   // pgn representation of a game //
   std::string PGN;

   // uci representation of a game //
   std::string UCI;

   // all the annotations in a game //
   std::string Annotations;

   // a string to hold a game annotation //
   std::string AnnotationString;

   // the header portion of a game //
   std::string Header;

   // the notation portion of a game //
   std::string Notation;

   // the fen string //
   std::string FENString;

   // game event field //
   std::string Event;

   // game site field //
   std::string Site;

   // game date field //
   std::string Date;

   // long descriptive version of date //
   std::string LongDate;

   // game round field //
   std::string Round;

   // who played as white during the game //
   std::string White;

   // who played as black during the game //
   std::string Black;

   // the outcome of the game //
   std::string Result;

   // an individual move in portable game notation //
   std::string Move;

   // the fen castling rights //
   std::string FENCastlingRights;

   // the fen board layout //
   std::string FENLayout;

   // the fen starting color //
   std::string FENColor;

   // the fen en passant square //
   std::string FENEnpassantSquare;

   // the fen start move number //
   std::string FENStartMoveNumber;

   // the fen fifty move rule string //
   std::string FENFiftyMoveRule;

   // a string iterator //
   std::string::iterator StringIter;

   // a vector of moves in portable game notation //
   std::vector<std::string> MoveList;

   // a vector of annotations in a game //
   std::vector<std::string> AnnotationList;

   // iterator to move list //
   std::vector<std::string>::iterator MoveIter;

   // total number of plies in the game //
   int TotalPlies();

   // get the fields from the header of a game file //
   void GetFields();

   // extract the moves for a game from the game file //
   void ExtractMoves();

   // construct pgn representation of game //
   void ConstructPGN();

   // process date string into a user readable form //
   void ProcessDate();

   // process the fen string //
   void ProcessFENString();

   // remove all the non ascii characters from a string //
   std::string RemoveNonAscii(std::string);

   // convert integer to string //
   std::string IntToString(int);

   // extract a field from the game header //
   std::string HeaderField(std::string);

   // public functions //
   public:

      // the date as a single number which is used for sorting //
      int SortingDate();

      // initialize attributes //
      void InitializeAttributes();

      // store move into move list //
      void StoreMove(std::string, std::string);

      // store the game header attributes to the game data //
      void StoreGameHeader(std::string, std::string, std::string, std::string,
                          std::string, std::string, std::string, std::string,
                          std::string, std::string, std::string, std::string,
                          std::string, std::string, std::string, std::string,
                          std::string, std::string);

      // store the game header attributes to the game data //
      void StoreGameHeader(std::string);

      // store the game move notation to the game data //
      void StoreGameNotation(std::string);

      // write pgn game //
      void WritePGN(std::string, bool);

      // append a game to a currently existing pgn file //
      void AppendPGN(std::string, bool);

      // process a pgn game file //
      bool ProcessPGN();

      // replace all the occurrences of a substring in a string //
      std::string ReplaceSubstring(std::string, std::string, std::string);

      // contruct the game in uci notation //
      std::string ConstructUCI();

      // construct a string of all the move annotations in a game //
      std::string ConstructAnnotations();

      // create a game description //
      std::string GameDescription();

      // name of white player //
      std::string WhitePlayer();

      // name of black player //
      std::string BlackPlayer();

      // get the time control for the game //
      std::string GetTimeControl();

      // get the opening for the game //
      std::string GetOpening();

      // get the section for the game //
      std::string GetSection();

      // get the board number for the game //
      std::string GetBoardNumber();

      // get the white elo for the game //
      std::string GetWhiteElo();

      // get the black elo for the game //
      std::string GetBlackElo();

      // get the white uscf for the game //
      std::string GetWhiteUSCF();

      // get the black uscf for the game //
      std::string GetBlackUSCF();

      // get the annotator for the game //
      std::string GetAnnotator();

      // get the eco for the game //
      std::string GetECO();

      // get the fen string //
      std::string GetFENString();

      // get the site location of the game //
      std::string GetSite();

      // get the event title of the game //
      std::string GetEvent();

      // get the round number of the game //
      std::string GetRound();

      // return the round number of the game //
      std::string GetResult();

      // return the date the game was played on //
      std::string GetDate();
};

// class representing game database //
class ChessDatabase
{
   // a temporary data value //
   int TempValue;

   // loop index //
   int Index;

   // current game count when searching through game database //
   int Count;

   // selected game that user wants to review from game database archive //
   int Game;

   // whether had adjacent values in a list swapped //
   bool WasSwapped;

   // whether have gotten to the next game in the pgn collection data file //
   bool BeginningOfNextGame;

   // game data //
   ChessData Data;

   // amount of space padding needed in game description //
   std::string Pad;

   // a line of data read in from a data file  //
   std::string EntireLine;
   // a line of data read in from a data file //
   std::string Line;

   // the header portion of the game data file read in //
   std::string Header;

   // the move notation of the game data file read in //
   std::string Notation;

   // the game descriptions list string //
   std::string DescriptionsList;

   // input file stream //
   std::ifstream Read;

   // list of indices in the game data list //
   std::vector<int> Indices;

   // list of dates games played on from data list that will be used for sorting //
   std::vector<int> SortingDates;

   // list of game data //
   std::vector<ChessData> DataList;

   // the temporary list of game data //
   std::vector<ChessData> TempDataList;

   // iterator to game data list //
   std::vector<ChessData>::iterator DataIter;

   // create list of descriptions for the games in database archive //
   void GameDescriptions();

   // sort game database records by date //
   void DateSort();

   // remove the leading spaces from a string //
   void RemoveLeadingSpaces();

   // convert integer to string //
   std::string IntToString(int);

   // public functions //
   public:

      // read in a collection of pgn games //
      bool ReadGameCollection(std::string);

      // the size of the data list from the game collection //
      long GetDataListSize();

      // get the specified game data from the pgn game collection //
      ChessData GetGameData(int);

      // get the game descriptions from the game collection //
      std::string GetGameDescriptions();
};

// convert an integer to a string //
std::string ChessPiece::IntToString(int Number)
{
   // create a string stream //
   std::stringstream Stream;

   // send the number into the string stream //
   Stream << Number;

   // return the string stream as a string //
   return Stream.str();
}

// mark whether piece is an isolated pawn //
void ChessPiece::SetIsolatedPawnStatus(bool Status)
{
   // mark whether piece is an isolated pawn //
   Booleans.set(0, Status);
}

// mark whether piece is a passed pawn //
void ChessPiece::SetPassedPawnStatus(bool Status)
{
   // mark whether piece is a passed pawn //
   Booleans.set(1, Status);
}

// mark whether piece is a doubled pawn //
void ChessPiece::SetDoubledPawnStatus(bool Status)
{
   // mark whether piece is a doubled pawn //
   Booleans.set(2, Status);
}

// mark whether piece is a tripled pawn //
void ChessPiece::SetTripledPawnStatus(bool Status)
{
   // mark whether piece is a tripled pawn //
   Booleans.set(3, Status);
}

// mark that piece has already moved //
void ChessPiece::MarkMoved()
{
   // mark that piece has already moved //
   Booleans.set(7, true);
}

// return sign based on color of piece //
short ChessPiece::ColorSign()
{
   // if piece is white //
   if (IsWhite())
      // return positive one //
      return 1;
   // piece is black //
   else
      // return negative one //
      return -1;
}

// mark that piece causes a check //
void ChessPiece::CauseCheck()
{
   // mark that piece causes a check //
   Booleans.set(4, true);
}

// determine if piece causes a check //
bool ChessPiece::IsChecker()
{
   // determine if piece causes a check //
   return Booleans.test(4);
}

// whether piece is currently the side to move //
bool ChessPiece::SideMoving()
{
   // return whether the piece is currently the side to move //
   return Booleans.test(6) == Booleans.test(9);
}

// whether piece is not currently the side to move //
bool ChessPiece::NotSideMoving()
{
   // return whether the piece is not currently the side to move //
   return !SideMoving();
}

// promote pawn to knight //
void ChessPiece::PromotePawnToKnight()
{
   // set type to knight //
   Type = 1;

   // set material value //
   AssignMaterial();

   // set the zobrist key index //
   SetZobristKeyIndex();
}

// promote pawn to bishop //
void ChessPiece::PromotePawnToBishop()
{
   // set type to bishop //
   Type = 2;

   // set material value //
   AssignMaterial();

   // set the zobrist key index //
   SetZobristKeyIndex();
}

// promote pawn to rook //
void ChessPiece::PromotePawnToRook()
{
   // set type to rook //
   Type = 3;

   // set material value //
   AssignMaterial();

   // set the zobrist key index //
   SetZobristKeyIndex();
}

// promote pawn to queen //
void ChessPiece::PromotePawnToQueen()
{
   // set type to queen //
   Type = 4;

   // set material value //
   AssignMaterial();

   // set the zobrist key index //
   SetZobristKeyIndex();
}

// remove piece from the board //
void ChessPiece::DeactivatePiece()
{
   // mark that piece is no longer active on the board //
   Booleans.set(5, false);

   // set piece index to an invalid value //
   Index = -1;
}

// determine if piece can be captured en passant //
bool ChessPiece::IsEnpassant()
{
   // return whether piece can be captured en passant //
   return Booleans.test(8);
}

// mark that piece can potentially be captured en passant on the next move //
void ChessPiece::MakeEnpassantEligible()
{
   // mark that piece can be captured en passant on the next move //
   Booleans.set(8, true);
}

// mark that piece cannot be captured en passsant on the next move //
void ChessPiece::MakeEnpassantIneligible()
{
   // mark that piece cannot be captured en passant on the next move //
   Booleans.set(8, false);
}

// if piece is not active on the board //
bool ChessPiece::NotActive()
{
   // return whether piece is not active on the board //
   return !Booleans.test(5);
}

// if piece is active on the board //
bool ChessPiece::IsActive()
{
   // return whether piece is active on the board //
   return Booleans.test(5);
}

// get the index of the piece in the piece list //
short ChessPiece::PieceIndex()
{
   // return the index position of the piece in the piece list //
   return Index;
}

// determine if piece is attacked in the given direction //
bool ChessPiece::DirectionallyAttacked(short Direction)
{
   // return if piece is attacked in this direction //
   return Attacks.test(Direction);
}

// determine if piece moves and violates its pinned status //
bool ChessPiece::ViolatesPin(short Direction)
{
   // return whether piece violates its pinned status in this direction //
   return !Pins.test(Direction) && Pins.any();
}

// mark piece as being attacked in this direction //
void ChessPiece::DirectionallyAttack(short Direction)
{
   // mark piece as being attacked in this direction //
   Attacks.set(Direction);
}

// mark directions that pinned piece is restricted to moving along //
void ChessPiece::DirectionallyPin(short Direction)
{
   // allow piece to move in this direction //
   Pins.set(Direction);

   // allow piece to move in this direction //
   Pins.set((Direction + 8) % 16);
}

// get the centered of the piece //
short ChessPiece::Centeredness()
{
   // if the piece is in the center four squares of the board //
   if (Rank > 2 && Rank < 5 && File > 2 && File < 5)
      // return multiplier for being in the center four squares of the board //
      return 3;
   // if the piece is in the center sixteen squares of the board //
   else if (Rank > 1 && Rank < 6 && File > 1 && File < 6)
      // return multiplier for being in the center sixteen squares of the board //
      return 2;
   // if the piece is in the center thirty-six squares of the board //
   else if (Rank > 0 && Rank < 7 && File > 0 && File < 7)
      // return multiplier for being in the center thirty-six squares of the board //
      return 1;
   // if the piece is on the outer edge of the board //
   else
      // return multiplier for being on the outer edge of the board //
      return -5;
}

// compute points given for development of piece //
short ChessPiece::GetDevelopedPoints(short PiecesLeft)
{
   // if piece is a pawn //
   if (IsPawn())
   {
      // if piece is white //
      if (IsWhite())
         // return more points for more further advancement and on center files //
         return Rank - 1 + 3 * (File >= 2 && File <= 5);
      // piece is black //
      else
         // return more points for more further advancement and on center files //
         return Rank - 6 - 3 * (File >= 2 && File <= 5);
   }
   // if piece is a king //
   else if (IsKing())
   {
      // if there are sixteen or less pieces on the board //
      if (PiecesLeft <= 16)
         // return points for being centered on the board //
         return ColorSign() * Centeredness();
      // there are more than sixteen pieces on the board //
      else
      {
         // if is a white king and on rank 0 //
         if (IsWhite() && Rank == 0)
            // return points for white king being on rank 0 //
            return 2;
         // if is a black king and on rank 7 //
         else if (IsBlack() && Rank == 7)
            // return points for black king being on rank 7 //
            return -2;
         // king is not on home rank //
         else
            // return points not not being on home rank //
            return -2 * ColorSign();
      }
   }
   // if piece is a queen //
   else if (IsQueen())
   {
      // if there are 25 or less pieces on the board //
      if (PiecesLeft <= 25)
         // return points for being centered on the board //
         return ColorSign() * Centeredness();
      // there are more than 25 pieces on the board //
      else
         // return zero points for development //
         return 0;
   }
   // if piece is a rook //
   else if (IsRook())
   {
      // if rook is on the center files and ten or more pieces left //
      if (File > 2 && File < 5 && PiecesLeft >= 10)
         // return more points for being on the center files //
         return ColorSign();
      // rook is not on the center files //
      else
         // return zero points for development //
         return 0;
   }
   // if piece is a knight or bishop //
   else
      // return points for being centered on the board //
      return ColorSign() * Centeredness();
}

// get the passed pawn multiplier //
short ChessPiece::PassedPawnMultiplier()
{
   // if piece is white //
   if (IsWhite())
      // return the passed pawn multiplier for white //
      return Rank - 1;
   // piece is black //
   else
      // return the passed pawn multiplier for black //
      return 6 - Rank;
}

// the points for being a doubled points //
short ChessPiece::GetDoubledPawnPoints()
{
   // return the points for being a doubled pawn //
   return -ColorSign() * Booleans.test(2);
}

// the points for being a tripled pawn //
short ChessPiece::GetTripledPawnPoints()
{
   // return the points for being a tripled pawn //
   return -ColorSign() * Booleans.test(3);
}

// the points for being a passed pawn //
short ChessPiece::GetPassedPawnPoints()
{
   // return the points for being a passed pawn //
   return ColorSign() * PassedPawnMultiplier() * Booleans.test(1);
}

// the points for being an isolated pawn //
short ChessPiece::GetIsolatedPawnPoints()
{
   // return the points for being an isolated pawn //
   return -ColorSign() * Booleans.test(0);
}

// return the points for being a piece on an outpost square //
short ChessPiece::GetOutpostPoints()
{
   // if the piece is a major piece guarded by a pawn and not attacked by a pawn //
   if (!(GuardedByPawn() && !IsPawn() && NotAttackedByPawn()))
      // return zero points since not a piece on an outpost square //
      return 0;

   // if piece is on the enemy side of the board //
   if ((IsWhite() && Rank > 3) || (IsBlack() && Rank < 4))
      // return points for a major piece on an outpost square //
      return ColorSign();
   // piece is not a major piece on an outpost square //
   else
      // return zero points since not a major piece on an output square //
      return 0;
}

// compute points to give for mobility //
short ChessPiece::GetMobilityPoints(short PiecesLeft)
{
   // if a nonpawn piece has limited mobility //
   if (!IsPawn() && Mobility <= 2)
      // return points for limited mobility //
      return -5 * ColorSign();

   // if piece is a knight or a bishop //
   if (IsKnight() || IsBishop())
      // return the mobility points for a knight //
      return ColorSign() * Mobility;
   // if piece is a rook or a queen //
   else if (IsRook() || IsQueen())
   {
      // if there are 25 or more pieces on the board //
      if (PiecesLeft >= 25)
         // return points for mobility //
         return ColorSign() * Mobility;
      // there are less than 25 pieces on the board //
      else
         // return points for mobility //
         return 2 * ColorSign() * Mobility;
   }
   // if piece is a king //
   else if (IsKing())
   {
      // if there are 25 or more pieces on the board //
      if (PiecesLeft >= 25)
         // return no points for mobility //
         return 0;
      // if there are 16 or more and less than 25 pieces on the board //
      else if (PiecesLeft >= 16)
         // return points for mobility //
         return ColorSign() * Mobility;
      // there are less than 16 pieces on the board //
      else
         // return points for mobility //
         return 4 * ColorSign() * Mobility;
   }
   // piece is a pawn //
   else
      // return zero points for mobility //
      return 0;
}

// get points for material value of piece //
short ChessPiece::GetMaterialPoints()
{
   // return points for material value of piece //
   return ColorSign() * Value;
}

// the value of the piece //
short ChessPiece::PieceValue()
{
   // return the value of the piece //
   return Value;
}

// if piece has not moved a valid number of squares from starting position //
bool ChessPiece::IllegalLength(short Direction, short Length)
{
   // if piece hasn't moved a valid number of squares from starting position //
   return !(IsQueen() || IsRook() || IsBishop() || (Length == 1 && (IsPawn() ||
          IsKing() || IsKnight())) || (Length == 2 && IsPawn() && Direction % 8 == 0
          && NeverMoved()) || (IsPseudoKing() && ((Length == 1 && Direction % 2 == 1)
          || (Direction % 2 == 0))));
}

// return new board position for a possible move //
short ChessPiece::NextPosition(short Direction, short Length)
{
   // if piece has not moved an allowed length along the board //
   if (IllegalLength(Direction, Length))
      // return value for no next valid position found //
      return -1;

   // assign rank coordinate origin //
   NewRank = Rank;

   // assign file coordinate origin //
   NewFile = File;

   // process each possible direction piece is moving in //
   switch(Direction)
   {
      // if moving to 12:00 //
      case 0:

         // increment rank by path length //
         NewRank += Length;

         // exit the switch statement //
         break;

      // if moving to 1:00 //
      case 1:

         // increment rank by two //
         NewRank += 2;

         // increment file by one //
         NewFile++;

         // exit the switch statement //
         break;

      // if moving to 1:30 //
      case 2:

         // increment rank by path length //
         NewRank += Length;

         // increment file by path length //
         NewFile += Length;

         // exit the switch statement //
         break;

      // if moving to 2:00 //
      case 3:

         // increment rank by one //
         NewRank++;

         // increment file by two //
         NewFile += 2;

         // exit the switch statement //
         break;

      // if moving to 3:00 //
      case 4:

         // increment file by path length //
         NewFile += Length;

         // exit the switch statement //
         break;

      // if moving to 4:00 //
      case 5:

         // decrement rank by one //
         NewRank--;

         // increment file by two //
         NewFile += 2;

         // exit the switch statement //
         break;

      // if moving to 4:30 //
      case 6:

         // decrement rank by path length //
         NewRank -= Length;

         // increment file by path length //
         NewFile += Length;

         // exit the switch statement //
         break;

      // if moving to 5:00 //
      case 7:

         // decrement rank by two //
         NewRank -= 2;

         // increment file by one //
         NewFile++;

         // exit the switch statement //
         break;

      // if moving to 6:00 //
      case 8:

         // decrement rank by path length //
         NewRank -= Length;

         // exit the switch statement //
         break;

      // if moving to 7:00 //
      case 9:

         // decrement rank by two //
         NewRank -= 2;

         // decrement file by one //
         NewFile--;

         // exit the switch statement //
         break;

      // if moving to 7:30 //
      case 10:

         // decrement rank by path length //
         NewRank -= Length;

         // decrement file by path length //
         NewFile -= Length;

         // exit the switch statement //
         break;

      // if moving to 8:00 //
      case 11:

         // decrement rank by one //
         NewRank--;

         // decrement file by two //
         NewFile -= 2;

         // exit the switch statement //
         break;

      // if moving to 9:00 //
      case 12:

         // decrement file by path length //
         NewFile -= Length;

         // exit the switch statement //
         break;

      // if moving to 10:00 //
      case 13:

         // increment rank by one //
         NewRank++;

         // decrement file by two //
         NewFile -= 2;

         // exit the switch statement //
         break;

      // if moving to 10:30 //
      case 14:

         // increment rank by path length //
         NewRank += Length;

         // decrement file by path length //
         NewFile -= Length;

         // exit the switch statement //
         break;

      // if moving to 11:00 //
      case 15:

         // increment rank by two //
         NewRank += 2;

         // decrement file by one //
         NewFile--;

         // exit the switch statement //
         break;
   }

   // if piece has moved off of board //
   if (NewRank < 0 || NewRank > 7 || NewFile < 0 || NewFile > 7)
      // return value for no valid next position found //
      return -1;
   // piece has not moved off of board //
   else
      // return new position for move //
      return NewRank * 8 + NewFile;
}

// get rank coordinate of piece //
short ChessPiece::PieceRank()
{
   // return rank coordinate of piece //
   return Rank;
}

// get file coordinate of piece //
short ChessPiece::PieceFile()
{
   // return file coordinate of piece //
   return File;
}

// calculate the position from the rank and file coordinates //
short ChessPiece::PiecePosition()
{
   // return position of piece on board //
   return Rank * 8 + File;
}

// calculate the position one rank behind an en passant eligible piece //
short ChessPiece::EnpassantPiecePosition()
{
   // if piece is white //
   if (IsWhite())
      // return the position one rank behind an en passant eligible piece //
      return (Rank - 1) * 8 + File;
   // if piece is black //
   else
      // return the position one rank behind an en passant eligible piece //
      return (Rank + 1) * 8 + File;
}

// set coordinates of piece on board //
void ChessPiece::SetCoordinates(short Position)
{
   // get rank coordinate from board position //
   Rank = Position / 8;

   // get file coordinate from board position and rank //
   File = Position - Rank * 8;
}

// if is a pawn eligible to promote //
bool ChessPiece::PawnCanPromote()
{
   // return whether is a pawn eligible to promote //
   return IsPawn() && (IsWhite() && Rank == 7 || IsBlack() && Rank == 0);
}

// shift file coordinate of piece on board //
void ChessPiece::ShiftFile(short Shift)
{
   // mark that piece has moved //
   Booleans.set(7, true);

   // shift file coordinate of piece //
   File += Shift;
}

// update position of piece on board //
void ChessPiece::UpdatePosition(short Position)
{
   // mark that piece has moved //
   Booleans.set(7, true);

   // set rank and file of piece on board //
   SetCoordinates(Position);
}

// determine if piece is allowed to move in this direction //
bool ChessPiece::AllowedDirection(short Direction)
{
   // return if piece is moving in an allowed direction //
   return (IsPawn() && IsWhite() && (Direction == 0 || Direction == 2 || Direction ==
          14)) || (IsPawn() && IsBlack() && (Direction == 6 || Direction == 8 ||
          Direction == 10)) || (IsKnight() && Direction % 2 == 1) || (IsBishop() &&
          (Direction + 2) % 4 == 0) || (IsRook() && Direction % 4 == 0) || ((IsQueen()
          || IsKing()) && Direction % 2 == 0) || IsPseudoKing();
}

// if piece is white //
bool ChessPiece::IsWhite()
{
   // return if piece is white //
   return Booleans.test(6);
}

// if piece is black //
bool ChessPiece::IsBlack()
{
   // return if piece is black //
   return !Booleans.test(6);
}

// if piece has never moved //
bool ChessPiece::NeverMoved()
{
   // return whether piece has never moved //
   return !Booleans.test(7);
}

// increase the mobility of a piece //
void ChessPiece::IncreaseMobility()
{
   // increase the mobility of the piece //
   Mobility++;
}

// reinitialize piece attributes //
void ChessPiece::ReinitializeAttributes(bool Color)
{
   // initialize piece mobility to zero //
   Mobility = 0;

   // set that pawn is not initially marked as isolated //
   Booleans.set(0, false);

   // set that pawn is not initially marked as doubled //
   Booleans.set(2, false);

   // set that pawn is not initially marked as tripled //
   Booleans.set(3, false);

   // set that pawn is not initially marked as passed //
   Booleans.set(1, false);

   // clear directions along which piece is attacked //
   Attacks = 0;

   // clear directions along which piece can move if pinned //
   Pins = 0;

   // assign color of the side currently moving //
   Booleans.set(9, Color);

   // mark that piece does not currently cause a check //
   Booleans.set(4, false);
}

// initialize piece attributes //
void ChessPiece::InitializeAttributes(short AssignedType, bool AssignedColor,
                                      short AssignedPosition, short AssignedIndex,
                                      bool ActiveStatus)
{
   // initialize the boolean attributes of the piece to zero //
   Booleans = 0;

   // set assigned type //
   Type = AssignedType;

   // set the assigned color of the piece //
   Booleans.set(6, AssignedColor);

   // set index location of piece in piece list //
   Index = AssignedIndex;

   // mark whether piece is active on the board //
   Booleans.set(5, ActiveStatus);

   // mark that piece has not initially moved //
   Booleans.set(7, false);

   // piece is not eligible to be captured en passant //
   Booleans.set(8, false);

   // set coordinates of piece on board //
   SetCoordinates(AssignedPosition);

   // assign material value relevant to the piece type //
   AssignMaterial();

   // set the zobrist key index for the piece //
   SetZobristKeyIndex();
}

// set the zobrist key index for the piece//
void ChessPiece::SetZobristKeyIndex()
{
   // if piece is white //
   if (Booleans.test(6))
      // create a zobrist key index to use in finding the zobrist hash //
      ZobristKeyIndex = Type + 1;
   // piece is black //
   else
      // create a zobrist key index to use in finding the zobrist hash //
      ZobristKeyIndex = Type + 7;
}

// get a zobrist key index //
short ChessPiece::GetZobristKeyIndex()
{
   // return a zobrist key index //
   return ZobristKeyIndex;
}

// assign material value relevant to the piece type //
void ChessPiece::AssignMaterial()
{
   // if piece is a pawn //
   if (IsPawn())
      // set material value of pawn //
      Value = 100;
   // if piece is a knight //
   else if (IsKnight())
      // set material value of knight //
      Value = 300;
   // if piece is a bishop //
   else if (IsBishop())
      // set material value of bishop //
      Value = 325;
   // if piece is a rook //
   else if (IsRook())
      // set material value of rook //
      Value = 500;
   // if piece is a queen //
   else if (IsQueen())
      // set material value of queen //
      Value = 900;
   // piece is a king //
   else if (IsKing())
      // set material value to one since king cannot be captured //
      Value = 1;
}

// create a white pawn //
void ChessPiece::CreateWhitePawn(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(0, true, Position, ListIndex, true);
}

// create a white knight //
void ChessPiece::CreateWhiteKnight(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(1, true, Position, ListIndex, true);
}

// create a white bishop //
void ChessPiece::CreateWhiteBishop(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(2, true, Position, ListIndex, true);
}

// create a white rook //
void ChessPiece::CreateWhiteRook(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(3, true, Position, ListIndex, true);
}

// create a white queen //
void ChessPiece::CreateWhiteQueen(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(4, true, Position, ListIndex, true);
}

// create a white king //
void ChessPiece::CreateWhiteKing(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(5, true, Position, ListIndex, true);
}

// create a white pseudo king for determining which moves cause check //
void ChessPiece::CreateWhitePseudoKing(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(6, true, Position, ListIndex, false);
}

// create a black pawn //
void ChessPiece::CreateBlackPawn(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(0, false, Position, ListIndex, true);
}

// create a black knight //
void ChessPiece::CreateBlackKnight(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(1, false, Position, ListIndex, true);
}

// create a black bishop //
void ChessPiece::CreateBlackBishop(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(2, false, Position, ListIndex, true);
}

// create a black rook //
void ChessPiece::CreateBlackRook(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(3, false, Position, ListIndex, true);
}

// create a black queen //
void ChessPiece::CreateBlackQueen(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(4, false, Position, ListIndex, true);
}

// create a black king //
void ChessPiece::CreateBlackKing(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(5, false, Position, ListIndex, true);
}

// create a black pseudo king for determining which moves cause check //
void ChessPiece::CreateBlackPseudoKing(short Position, short ListIndex)
{
   // initialize type, color, position, list index, and whether active //
   InitializeAttributes(6, false, Position, ListIndex, false);
}

// whether piece is a pseduo king used for determining which moves cause check //
bool ChessPiece::IsPseudoKing()
{
   // return whether is a pseudo king used for determining which moves cause check //
   return Type == 6;
}

// determine if piece is a pawn //
bool ChessPiece::IsPawn()
{
   // return if piece is a pawn //
   return Type == 0;
}

// determine if piece is a knight //
bool ChessPiece::IsKnight()
{
   // return if piece is a knight //
   return Type == 1;
}

// determine if piece is a bishop //
bool ChessPiece::IsBishop()
{
   // return if piece is a bishop //
   return Type == 2;
}

// determine if piece is a rook //
bool ChessPiece::IsRook()
{
   // return if piece is a rook //
   return Type == 3;
}

// determine if piece is a queen //
bool ChessPiece::IsQueen()
{
   // return if piece is a queen //
   return Type == 4;
}

// determine if piece is a king //
bool ChessPiece::IsKing()
{
   // return if type is a king //
   return Type == 5;
}

// descriptive board location of piece //
std::string ChessPiece::BoardLocation()
{
   // return descriptive board location of piece //
   return FileName() + RankName();
}

// descriptive board location of the FEN enpassant target square for position editor //
std::string ChessPiece::EnpassantBoardLocation()
{
   // if location of piece on board is on the fourth rank //
   if (Rank == 3)
      // return the corresponding FEN enpassant target square location //
      return FileName() + IntToString(3);
   // if location of piece on board is on the fifth rank //
   else if (Rank == 4)
      // return the corresponding FEN enpassant target square location //
      return FileName() + IntToString(6);
   // piece is not on the fourth or fifth rank where enpassant pieces would be //
   else
      // return an empty string //
      return "";
}

// get description of rank coordinate //
std::string ChessPiece::RankName()
{
   // return the rank number as a string //
   return IntToString(Rank + 1);
}

// get description of file coordinate //
std::string ChessPiece::FileName()
{
   // if first file //
   if (File == 0)
      // return file of a //
      return "a";
   // if second file //
   else if (File == 1)
      // return file of b //
      return "b";
   // if third file //
   else if (File == 2)
      // return file of c //
      return "c";
   // if fourth file //
   else if (File == 3)
      // return file of d //
      return "d";
   // if fifth file //
   else if (File == 4)
      // return file of e //
      return "e";
   // if sixth file //
   else if (File == 5)
      // return file of f //
      return "f";
   // if seventh file //
   else if (File == 6)
      // return file of g //
      return "g";
   // if eighth file //
   else
      // return file of h //
      return "h";
}

// lower case symbol representation of piece //
std::string ChessPiece::LowerPieceSymbol()
{
   // if piece is a pawn //
   if (IsPawn())
      // return symbol for pawn //
      return "p";
   // if piece is a knight //
   else if (IsKnight())
      // return symbol for knight //
      return "n";
   // if piece is a bishop //
   else if (IsBishop())
      // return symbol for bishop
      return "b";
   // if piece is a rook //
   else if (IsRook())
      // return symbol for rook //
      return "r";
   // if piece is a queen //
   else if (IsQueen())
      // return symbol for queen //
      return "q";
   // piece is a king //
   else
      // return symbol for king //
      return "k";
}

// upper case symbol representation of piece //
std::string ChessPiece::UpperPieceSymbol()
{
   // if piece is a pawn //
   if (IsPawn())
      // return symbol for pawn //
      return "P";
   // if piece is a knight //
   else if (IsKnight())
      // return symbol for knight //
      return "N";
   // if piece is a bishop //
   else if (IsBishop())
      // return symbol for bishop //
      return "B";
   // if piece is a rook //
   else if (IsRook())
      // return symbol for rook //
      return "R";
   // if piece is a queen //
   else if (IsQueen())
      // return symbol for queen //
      return "Q";
   // piece is a king //
   else
      // return symbol for king //
      return "K";
}

// pgn representation of piece symbol //
std::string ChessPiece::PGNSymbol()
{
   // if piece is a pawn //
   if (IsPawn())
      // return empty string since in pgn the pawn symbol is not used //
      return "";
   // piece is not a pawn //
   else
      // return the symbol for the appropriate piece //
      return UpperPieceSymbol();
}

// get type of piece //
short ChessPiece::PieceType()
{
   // return type of piece //
   return Type;
}

// get color of piece //
bool ChessPiece::PieceColor()
{
   // return color of piece //
   return Booleans.test(6);
}

// whether piece is on a white square or not //
bool ChessPiece::OnWhiteSquare()
{
   // return whether piece is on a white square or not //
   return (File + Rank) % 2;
}

// mark that piece is guarded by a pawn //
void ChessPiece::MarkGuardedByPawn()
{
   // mark that piece is guarded by a pawn //
   Booleans.set(10);
}

// mark that piece is attacked by a pawn //
void ChessPiece::MarkAttackedByPawn()
{
   // mark that piece is attacked by a pawn //
   Booleans.set(11);
}

// determine if piece is guarded by a pawn //
bool ChessPiece::GuardedByPawn()
{
   // return whether piece is guarded by a pawn //
   return Booleans.test(10);
}

// determine if piece is not attacked by a pawn //
bool ChessPiece::NotAttackedByPawn()
{
   // return whether piece is not attacked by a pawn //
   return !Booleans.test(11);
}

// get the piece position of the captured piece //
int ChessMove::CapturedPieceSquare()
{
   // return the piece position of the captured piece //
   return CapturedPiece.PiecePosition();
}

// get whether a king can potentially be captured which is illegal to have //
bool ChessMove::KingIsCaptured()
{
   // return whether a king can be potentially captured which is illegal to have //
   return CapturedPiece.IsKing();
}

// set the level of detail that will need in the detail of the pgn label //
void ChessMove::SetPGNDetail(int Detail)
{
   // if the detail for the pgn label is greater than the previous detail //
   if (Detail > PGNSelection)
      // update the type of pgn detail to use for the pgn label //
      PGNSelection = Detail;
}

// get a version of the pgn label for a move due to pgn label ambiguities //
std::string ChessMove::PGNLabelVersion(int PGNVersion)
{
   // save the original selection of the pgn move label version //
   OriginalPGNSelection = PGNSelection;

   // set the pgn label detail selection to the chosen value //
   PGNSelection = PGNVersion;

   // get the pgn label string //
   PGNLabelString = PGNLabel();

   // reset the pgn label detail selection to the original value //
   PGNSelection = OriginalPGNSelection;

   // return the pgn label string based on the pgn label version chosen //
   return PGNLabelString;
}

// pgn label of chess move in portable game notation //
std::string ChessMove::PGNLabel()
{
   // if have a kingside castle //
   if (IsKingsideCastle())
      // return pgn label for a kingside castle //
      return "O-O" + PGNCheck();
   // if have a queenside castle //
   else if (IsQueensideCastle())
      // return pgn label for a queenside castle //
      return "O-O-O" + PGNCheck();
   // is not a castling move //
   else
      // return pgn label for a noncastling move //
      return Piece.PGNSymbol() + PGNDetail() + PGNCapture() + UpdatedPiece.
             BoardLocation() + PGNPromotion() + PGNCheck();
}

// whether the piece is not currently the side to move //
bool ChessMove::NotSideMoving()
{
   // whether the piece is not currently the side to move //
   return Piece.NotSideMoving();
}

// whether the piece is currently the side to move //
bool ChessMove::SideMoving()
{
   // return whether the piece is currently the side to move //
   return Piece.SideMoving();
}

// whether a pawn is eligible to promote //
bool ChessMove::PawnCanPromote()
{
   // return whether a pawn is eligible to promote //
   return UpdatedPiece.PawnCanPromote();
}

// retrieve piece after has moved to new position //
ChessPiece ChessMove::RetrieveUpdatedPiece()
{
   // return the updated piece //
   return UpdatedPiece;
}

// the value of the piece moving //
int ChessMove::PieceValue()
{
   // return the value of the piece moving //
   return Piece.PieceValue();
}

// the value of the captured piece //
int ChessMove::CapturedPieceValue()
{
   // return the value of the captured piece //
   return CapturedPiece.PieceValue();
}

// get uci label of move //
std::string ChessMove::UCILabel()
{
   // if move was a kingside castle for white //
   if (IsKingsideCastle() && IsWhite())
      // return uci label for a kingside castle for white //
      return "e1g1";
   // if move was a kingside castle for black //
   else if (IsKingsideCastle() && IsBlack())
      // return uci label for a kingside castle for black //
      return "e8g8";
   // if move was a queenside castle for white //
   else if (IsQueensideCastle() && IsWhite())
      // return uci label for a queenside castle for white //
      return "e1c1";
   // if move was a queenside castle for black //
   else if (IsQueensideCastle() && IsBlack())
      // return uci label for a queenside castle for black //
      return "e8c8";
   // move was not a castling move //
   else
      // return the description for this move in terms of movement //
      return Piece.BoardLocation() + UpdatedPiece.BoardLocation() + PromotionSymbol();
}

// determine if move causes a check //
bool ChessMove::IsCheck()
{
   // return whether move causes a check //
   return Check == 0;
}

// determine if move causes a checkmate //
bool ChessMove::IsCheckmate()
{
   // return whether move causes a checkmate //
   return Check == 1;
}

// mark move as causing a check //
void ChessMove::CauseCheck()
{
   // mark move as causing a check //
   Check = 0;
}

// mark move as causing a checkmate //
void ChessMove::CauseCheckmate()
{
   // mark move as causing a checkmate //
   Check = 1;
}

// mark move as causing a stalemate //
void ChessMove::CauseStalemate()
{
   // mark move as causing a stalemate //
   Check = 2;
}

// mark move as a kingside castle //
void ChessMove::CastleKingside()
{
   // mark move as a kingside castle //
   Castle = 0;
}

// mark move as a queenside castle //
void ChessMove::CastleQueenside()
{
   // mark move as a queenside castle //
   Castle = 1;
}

// determine if move is a kingside castle //
bool ChessMove::IsKingsideCastle()
{
   // return whether move is a kingside castle //
   return Castle == 0;
}

// determine if move is a queenside castle //
bool ChessMove::IsQueensideCastle()
{
   // return whether move is a queenside castle //
   return Castle == 1;
}

// get position of piece after has moved //
int ChessMove::UpdatedPosition()
{
   // return position of piece after has moved //
   return UpdatedPiece.PiecePosition();
}

// mark that move has done an en passant pawn capture //
void ChessMove::CaptureEnpassant()
{
   // mark that move has done an en passant pawn capture //
   Enpassant = true;
}

// determine if move has done an en passant pawn capture //
bool ChessMove::IsEnpassant()
{
   // return if move has done an en passant pawn capture //
   return Enpassant;
}

// get the identification label for this move //
int ChessMove::MoveLabel()
{
   // return the identification label for this move //
   return Label;
}

// get the color of the piece moving //
bool ChessMove::PieceColor()
{
   // return the color of the piece moving //
   return Piece.PieceColor();
}

// determine whether the piece moving is a pawn //
bool ChessMove::IsPawn()
{
   // return whether the piece moving is a pawn //
   return Piece.IsPawn();
}

// determine whether the piece moving is a knight //
bool ChessMove::IsKnight()
{
   // return whether the piece moving is a knight //
   return Piece.IsKnight();
}

// determine whether the piece moving is a bishop //
bool ChessMove::IsBishop()
{
   // return whether the piece moving is a bishop //
   return Piece.IsBishop();
}

// determine whether the piece moving is a rook //
bool ChessMove::IsRook()
{
   // return whether the piece moving is a rook //
   return Piece.IsRook();
}

// determine whether the piece moving is a queen //
bool ChessMove::IsQueen()
{
   // return whether the piece moving is a queen //
   return Piece.IsQueen();
}

// determine whether the piece moving is a king //
bool ChessMove::IsKing()
{
   // return whether the piece moving is a king //
   return Piece.IsKing();
}

// mark piece as eligible to be captured en passant on the next move //
void ChessMove::MakeEnpassantEligible()
{
   // mark piece as eligible to be captured en passant on the next move //
   UpdatedPiece.MakeEnpassantEligible();
}

// promote pawn to a knight after it has moved //
void ChessMove::PromotePawnToKnight()
{
   // promote pawn to a knight after it has moved //
   UpdatedPiece.PromotePawnToKnight();
}

// promote pawn to a bishop after it has moved //
void ChessMove::PromotePawnToBishop()
{
   // promote pawn to a bishop after it has moved //
   UpdatedPiece.PromotePawnToBishop();
}

// promote pawn to a rook after it has moved //
void ChessMove::PromotePawnToRook()
{
   // promote pawn to a rook after it has moved //
   UpdatedPiece.PromotePawnToRook();
}

// promote pawn to a queen after it has moved //
void ChessMove::PromotePawnToQueen()
{
   // promote pawn to a queen after it has moved //
   UpdatedPiece.PromotePawnToQueen();
}

// promotion symbol if move results in a pawn promotion //
std::string ChessMove::PromotionSymbol()
{
   // move results in a pawn promotion //
   if (IsPawnPromotion())
      // return symbol of piece that pawn promotes to //
      return UpdatedPiece.LowerPieceSymbol();
   else
      // return an empty string if no pawn promotion //
      return "";
}

// determine whether a promotion has occurred //
bool ChessMove::IsPawnPromotion()
{
   // return whether a promotion has occurred //
   return UpdatedPiece.PieceType() != Piece.PieceType();
}

// get index of captured piece in piece list //
int ChessMove::CapturedPieceIndex()
{
   // return index of captured piece in piece list //
   return CapturedPiece.PieceIndex();
}

// determine if move is white //
bool ChessMove::IsWhite()
{
   // return if piece moving is white //
   return Piece.IsWhite();
}

// determine if move is black //
bool ChessMove::IsBlack()
{
   // return if piece moving is black //
   return Piece.IsBlack();
}

// get direction that piece moves in //
int ChessMove::MoveDirection()
{
   // return direction that piece moves in //
   return Direction;
}

// get index of moving piece in piece list //
int ChessMove::PieceIndex()
{
   // return index of moving piece in piece list //
   return Piece.PieceIndex();
}

// increase the detail of the pgn description //
std::string ChessMove::PGNDetail()
{
   // if lowest level of pgn detail //
   if (PGNSelection == 0)
      // return an empty string //
      return "";
   // if next level of pgn detail //
   else if (PGNSelection == 1)
      // return file location of piece to resolve a label name conflict //
      return Piece.FileName();
   // if next level of pgn detail //
   else if (PGNSelection == 2)
      // return rank location of piece to resolve a label name conflict //
      return Piece.RankName();
   // if highest level of pgn detail //
   else
      // return file and rank location of piece to resolve a label name conflict //
      return Piece.FileName() + Piece.RankName();
}

// pgn description of a pawn promotion //
std::string ChessMove::PGNPromotion()
{
   // if have a pawn promotion //
   if (IsPawnPromotion())
      // return pgn description of promotion //
      return "=" + UpdatedPiece.PGNSymbol();
   // is not a pawn promotion //
   else
      // return an empty string //
      return "";
}

// pgn description of check //
std::string ChessMove::PGNCheck()
{
   // if move causes a check //
   if (IsCheck())
      // return pgn symbol for a check //
      return "+";
   // if move causes a checkmate //
   else if (IsCheckmate())
      // return pgn symbol for a checkmate //
      return "#";
   // move does not cause a check or checkmate //
   else
      // return an empty string //
      return "";
}

// pgn description of capture //
std::string ChessMove::PGNCapture()
{
   // if a pawn captures a piece //
   if (IsCapture() && Piece.IsPawn())
      // return pgn symbol for a piece capture by a pawn //
      return Piece.FileName() + "x";
   // if a piece other than a pawn captures a piece //
   else if (IsCapture())
      // return pgn symbol for a piece capture //
      return "x";
   // was not a capture //
   else
      // return an empty string //
      return "";
}

// determine if move does not capture a piece //
bool ChessMove::NotCapture()
{
   // return whether move does not capture a piece //
   return CapturedPiece.NotActive();
}

// determine whether move captures a piece //
bool ChessMove::IsCapture()
{
   // return whether move captures a piece //
   return CapturedPiece.IsActive();
}

// store the piece captured during the move //
void ChessMove::StoreCapturedPiece(ChessPiece AssignedCapturedPiece)
{
   // store the piece captured during the move //
   CapturedPiece = AssignedCapturedPiece;
}

// make identification label for move //
void ChessMove::MakeLabel()
{
   // if move is a kingside castle //
   if (IsKingsideCastle())
      // let move label be 0 //
      Label = 0;
   // if move is a queenside castle //
   else if (IsQueensideCastle())
      // let move label be 1 //
      Label = 1;
   // move is a noncastling move //
   else
      // generate move identification label //
      Label = (std::bitset<17>(Piece.PiecePosition()) | std::bitset<17>
              (UpdatedPosition()) << 6 | std::bitset<17>(UpdatedPiece.PieceType()) <<
              14).set(12, PieceColor()).set(13, IsCapture()).to_ulong();
}

// initialize pgn information in move //
void ChessMove::InitializePGN()
{
   // set to first pgn label selection and increase if have label conflicts //
   PGNSelection = 0;
}

// mark that move can cause a check on the next move //
void ChessMove::MarkCanCheck()
{
   // mark that move can cause a check on the next move //
   CanCheck = true;
}

// determine whehter can cause a check on the next move //
bool ChessMove::CanCauseCheck()
{
   // return whether can cause a check on the next move //
   return CanCheck;
}

// initialize move attributes //
void ChessMove::InitializeAttributes(ChessPiece AssignedPiece, int AssignedPosition,
                                     int AssignedDirection, int AssignedLength)
{
   // set that initially can't cause a check on the next move //
   CanCheck = false;

   // store piece that is moving //
   Piece = AssignedPiece;

   // store copy of piece moving into updated piece //
   UpdatedPiece = Piece;

   // set the new board position for the updated piece //
   UpdatedPiece.UpdatePosition(AssignedPosition);

   // mark that castle has not occurred //
   Castle = -1;

   // mark that a check or mate has not occurred //
   Check = -1;

   // mark that en passant pawn capture has not occurred //
   Enpassant = false;

   // deactivate the captured piece //
   CapturedPiece.DeactivatePiece();

   // store the length that the piece has moved along the board //
   Length = AssignedLength;

   // store the direction along which the piece is moving //
   Direction = AssignedDirection;
}

// mark that a pawn can cause a check if it moves to this square //
void ChessSquare::MarkPawnCanCauseCheck()
{
   // mark that a pawn can cause a check if it moves to this square //
   PiecesThatCauseCheck.set(0);
}

// mark that a knight can cause a check if it moves to this square //
void ChessSquare::MarkKnightCanCauseCheck()
{
   // mark that a knight can cause a check if it moves to this square //
   PiecesThatCauseCheck.set(1);
}

// mark that a bishop can cause a check if it moves to this square //
void ChessSquare::MarkBishopCanCauseCheck()
{
   // mark that a bishop can cause a check if it moves to this square //
   PiecesThatCauseCheck.set(2);
}

// mark that a rook can cause a check if it moves to this square //
void ChessSquare::MarkRookCanCauseCheck()
{
   // mark that a rook can cause a check if it moves to this square //
   PiecesThatCauseCheck.set(3);
}

// mark that queen can cause a check if it moves to this square //
void ChessSquare::MarkQueenCanCauseCheck()
{
   // mark that queen can cause a check if it moves to this square //
   PiecesThatCauseCheck.set(4);
}

// check whether a piece can cause a check if it moves to this square //
bool ChessSquare::PieceCanCheckHere(int Type)
{
   // return whether a piece can cause a check if it moves to this square //
   return PiecesThatCauseCheck.test(Type);
}

// get the number of attacks by the specified color //
int ChessSquare::NumberOfAttacksByColor(bool AttackColor)
{
   // if need number of attacks by white //
   if (AttackColor)
      // return the number of attacks by white //
      return AttacksByWhite;
   // need number of attacks by black //
   else
      // return the number of attacks by black //
      return AttacksByBlack;
}

// add an attack to the square by the specified color //
void ChessSquare::AddAttackToSquare(bool PieceColor)
{
   // if white is moving //
   if (PieceColor)
      // increment the number of attacks by white //
      AttacksByWhite++;
   // if black is moving //
   else
      // increment the number of attacks by black //
      AttacksByBlack++;
}

// if piece on square is black //
bool ChessSquare::IsBlack()
{
   // return if piece on square is black //
   return Piece.IsBlack();
}

// if piece on square is white //
bool ChessSquare::IsWhite()
{
   // return if piece on square is white //
   return Piece.IsWhite();
}

// if piece on square is a rook //
bool ChessSquare::IsRook()
{
   // return if piece on square is a rook //
   return Piece.IsRook();
}

// if piece on square is a pawn //
bool ChessSquare::IsPawn()
{
   // return if piece on square is a pawn //
   return Piece.IsPawn();
}

// if piece on square is a king //
bool ChessSquare::IsKing()
{
   // return if piece on square is a king //
   return Piece.IsKing();
}

// return the piece list index of the piece on this square //
int ChessSquare::PieceIndex()
{
   // return the piece list index of the piece on this square //
   return Piece.PieceIndex();
}

// the position of piece on square //
int ChessSquare::PiecePosition()
{
   // return the position of piece on square //
   return Piece.PiecePosition();
}

// get a zobrist key index for piece on square //
int ChessSquare::GetZobristKeyIndex()
{
   // return a zobrist key index for piece on square //
   return Piece.GetZobristKeyIndex();
}

// initialize square attributes //
void ChessSquare::InitializeAttributes()
{
   // deactivate piece on the square //
   Piece.DeactivatePiece();
   // deactivate pseudo en passant piece on square //
   EnpassantPiece.DeactivatePiece();

   // mark that square does not require piece to move here to block a check //
   Block = false;

   // initially set the number of attacks by white on this square to zero //
   AttacksByWhite = 0;

   // initially set the number of attacks by black on this square to zero //
   AttacksByBlack = 0;

   // initialze pieces that cause a check if move to specified square to zero //
   PiecesThatCauseCheck = 0;
}

// whether piece does not need to block a check here //
bool ChessSquare::NotBlock()
{
   // whether piece does not need to block a check here //
   return !Block;
}

// mark that square requires a piece to move here to block a check //
void ChessSquare::MarkBlock()
{
   // mark that square requires a piece to move here to block a check //
   Block = true;
}

// determine whether occupying piece has never moved //
bool ChessSquare::NeverMoved()
{
   // return whether occupying piece has never moved //
   return Piece.NeverMoved();
}

// return whether square is not attacked //
bool ChessSquare::NotAttacked(bool PieceColor)
{
   // return whether square is not attacked //
   return !IsAttacked(PieceColor);
}

// determine whether square is attacked //
bool ChessSquare::IsAttacked(bool PieceColor)
{
   // if piece is white //
   if (PieceColor)
      // return whether black attacks this square that a white piece is on //
      return AttacksByBlack;
   // piece is black //
   else
      // return whether white attacks this square that a black piece is on //
      return AttacksByWhite;
}

// determine whether square is unoccupied //
bool ChessSquare::IsUnoccupied()
{
   // return whether square is unoccupied //
   return !Piece.IsActive();
}

// determine if square is occupied by a piece //
bool ChessSquare::IsOccupied()
{
   // return whether square is occupied by a piece //
   return Piece.IsActive();
}

// determine if square is occupied by a pseudo en passant piece //
bool ChessSquare::IsEnpassantOccupied()
{
   // return whether square is occupied by a pseudo en passant piece //
   return EnpassantPiece.IsActive();
}

// get the file number of the en passant piece currently on this square //
int ChessSquare::EnpassantFile()
{
   // return the file number the en passant piece is currently on //
   return EnpassantPiece.PieceFile();
}

// deactivate the piece on the square //
void ChessSquare::DeactivatePiece()
{
   // deactivate the piece on the square //
   Piece.DeactivatePiece();
}

// get the color of the piece on the square //
bool ChessSquare::PieceColor()
{
   // get the color of the piece on the square //
   return Piece.PieceColor();
}

// get the color of the pseudo en passant piece on the square //
bool ChessSquare::EnpassantPieceColor()
{
   // get the color of the pseudo en passant piece on the square //
   return EnpassantPiece.PieceColor();
}

// retrieve the pseudo en passant piece on the square //
ChessPiece ChessSquare::RetrieveEnpassantPiece()
{
   // retrieve the pseudo en passant piece on the square //
   return EnpassantPiece;
}

// retrieve the piece on the square //
ChessPiece ChessSquare::RetrievePieceOnSquare()
{
   // retrieve the piece on the square //
   return Piece;
}

// store the piece that occupies the square //
void ChessSquare::StorePiece(ChessPiece OccupyingPiece)
{
   // store the piece that occupies the square //
   Piece = OccupyingPiece;
}

// store the pseudo en passant piece that occupies the square //
void ChessSquare::StoreEnpassantPiece(ChessPiece OccupyingPiece)
{
   // store the pseudo en passant piece that occupies the square //
   EnpassantPiece = OccupyingPiece;
}

// sort games in database by date played using a bubble sort //
void ChessDatabase::DateSort()
{
   // if there is not enough data to sort //
   if (DataList.size() < 2)
      // exit function since there is not enough data to sort //
      return;

   // clear the indices list //
   Indices.clear();

   // clear the temporary data list //
   TempDataList.clear();

   // clear the sorting dates list //
   SortingDates.clear();

   // store the game data list into a temporary data list //
   TempDataList = DataList;

   // loop over the entire data list //
   for (Index = 0; Index < DataList.size(); Index++)
   {
      // store the index location in the data list //
      Indices.push_back(Index);

      // store the date the corresponding game was played on //
      SortingDates.push_back(DataList.at(Index).SortingDate());
   }

   // initialize count to zero //
   Count = 0;

   // set that out of order values were swapped //
   WasSwapped = true;

   // while had at least one pair of out of order values that were swapped //
   while (WasSwapped)
   {
      // reset whether out of orders values were swapped //
      WasSwapped = false;

      // increment the number of times have looped through the list //
      Count++;

      // loop over the unsorted portion of the list //
      for (Index = 0; Index < DataList.size() - Count; Index++)
         // check to see if two adjacent values are out of order //
         if (SortingDates.at(Index) < SortingDates.at(Index + 1))
         {
            // set that two adjacent values are out of order //
            WasSwapped = true;

            // save the temporary value of the data list index //
            TempValue = Indices.at(Index);

            // swap the value at index with the next value in the list //
            Indices.at(Index) = Indices.at(Index + 1);

            // store the temporary value into the next value in the list //
            Indices.at(Index + 1) = TempValue;

            // save the temporary value of the data list sorting date //
            TempValue = SortingDates.at(Index);

            // swap the value at index with the next value in the list //
            SortingDates.at(Index) = SortingDates.at(Index + 1);

            // store the temporary value into the next value in the list //
            SortingDates.at(Index + 1) = TempValue;
         }
   }

   // loop over all the games in the data list //
   for (Index = 0; Index < DataList.size(); Index++)
      // copy over the sorted game data from the temporary data list //
      DataList.at(Index) = TempDataList.at(Indices.at(Index));
}

// convert an integer to a string //
std::string ChessDatabase::IntToString(int Number)
{
   // create a string stream //
   std::stringstream Stream;

   // send the number into the string stream //
   Stream << Number;

   // return the string stream as a string //
   return Stream.str();
}

// get the game descriptions from the game collection //
std::string ChessDatabase::GetGameDescriptions()
{
   // return the game descriptions list from the game collection //
   return DescriptionsList;
}

// make list of all game descriptions in database //
void ChessDatabase::GameDescriptions()
{
   // clear the game descriptions list string //
   DescriptionsList = "";

   // initialize game count //
   Count = 0;

   // loop over all games in database //
   for (DataIter = DataList.begin(); DataIter != DataList.end(); DataIter++)
   {
      // increment game count //
      Count++;

      // if games 1 to 9 //
      if (Count < 10)
         // set padding after game number //
         Pad = "      ";
      // if games 10 to 99 //
      else if (Count < 100)
         // set padding after game number //
         Pad = "     ";
      // if games 100 to 999 //
      else if (Count < 1000)
         // set padding after game number //
         Pad = "    ";
      // if games 1000 to 9999 //
      else if (Count < 10000)
         // set padding after game number //
         Pad = "   ";
      // if games 10000 or greater //
      else
         // set padding after game number //
         Pad = "  ";

      // add game description to descriptions list //
      DescriptionsList += IntToString(Count) + Pad + DataIter->GameDescription() +
                          "**********";
   }
}

// get the size of the data list from the game collection //
long ChessDatabase::GetDataListSize()
{
   // return the size of the data list from the game collection //
   return DataList.size();
}

// get the specified game data from the pgn collection //
ChessData ChessDatabase::GetGameData(int GameNumber)
{
   // if the specified game number if within the range of the game data list //
   if (GameNumber < DataList.size())
      // return the game from the data list corresponding to the given game number //
      return DataList.at(GameNumber);
   // the specified game number is outside the range of the game data list //
   else
   {
      // initialize game data to be empty //
      Data.InitializeAttributes();

      // return an empty game data //
      return Data;
   }
}

// remove the leading spaces from a string //
void ChessDatabase::RemoveLeadingSpaces()
{
   // initialize the count to zero //
   Count = 0;

   // loop over all the characters in the line string //
   for (Index = 0; Index < Line.length(); Index++)
      // if have encountered a space in this position in the string //
      if (isspace(Line.at(Index)))
         // count the number of spaces in the string //
         Count++;
      // haven't encountered a space in this position in the string //
      else
         // exit the loop since have encountered a nonspace character in string //
         break;

   // extract the substring from the line string with leading spaces removed //
   Line = Line.substr(Count, Line.length() - Count);
}

// read in a collection of pgn games from a pgn data file //
bool ChessDatabase::ReadGameCollection(std::string FileName)
{
   // set that haven't initially reached the beginning of the next game //
   BeginningOfNextGame = false;

   // initialize the game data list //
   DataList.clear();

   // clear the inut file stream //
   Read.clear();

   // open the pgn collection game data file //
   Read.open(FileName.c_str());

   // if failed to open file //
   if (Read.fail())
      // return that were not able to open file //
      return false;

   // clear the entire line of string data that will be read in //
   EntireLine.clear();

   // while have not gotten to the end of the pgn collection file //
   while (!Read.fail())
   {
      // initialize the game data attributes //
      Data.InitializeAttributes();

      // clear the game header string //
      Header.clear();

      // clear the game notation string //
      Notation.clear();

      // do while have not encountered end of file //
      while (!Read.fail())
      {
         // if have not reached the beginning of the next game yet //
         if (!BeginningOfNextGame)
         {
            // if the entire line read in is now empty //
            if (EntireLine.empty())
               // read in a new line of data from the game database file //
               getline(Read, EntireLine);

            // get the index in line of where a carriage return is //
            Index = EntireLine.find('\r', 0);

            // if there was a carriage return in the line read in //
            if (Index != std::string::npos)
            {
               // get the line up to the first carriage return //
               Line = EntireLine.substr(0, Index + 1);

               // save the part of the entire line after the first carriage return //
               EntireLine = EntireLine.substr(Index + 1, EntireLine.length() - Index
                             - 1);
            }
            // there wasn't a carriage return encountered in the line read in //
            else
            {
               // assign the entire line read in to the line of data //
               Line = EntireLine;

               // clear the entire line //
               EntireLine.clear();
            }

            // remove the leading spaces from the line string //
            RemoveLeadingSpaces();
         }
         // if have reached the beginning of the next game //
         else
            // reset whether have reached beginning of next game or not //
            BeginningOfNextGame = false;

         // if the line read in is not empty //
         if (!Line.empty())
         {
            // if the line is part of the game header //
            if (Line.at(0) == '[' && Notation.empty())
               // add the line to the header string //
               Header += Line + " ";
            // if may have started reading in the next game in the pgn game file //
            else if (Line.at(0) == '[' && !Notation.empty())
            {
               // if the length of the line is greater than five characters //
               if (Line.length() > 5)
                  // check to see if encountered an Event header tag //
                  if (Line.substr(0, 6) == "[Event" || Line.substr(0, 4) == "[ECO")
                  {
                     // set that have reached the beginning of the next game //
                     BeginningOfNextGame = true;

                     // exit the loop since finished reading in a game //
                     break;
                  }

                  // add the line and a space to the notation string //
                  Notation += Line + " ";
            }
            // the line is not part of the game header //
            else
               // add the line to the notation string //
               Notation += Line + " ";
         }
      }

      // store the corresponding game header for the game data file //
      Data.StoreGameHeader(Header);

      // store the corresponding move notation for the game data file //
      Data.StoreGameNotation(Notation);

      // process the pgn portion of the game data //
      Data.ProcessPGN();

      // push the game data onto the game data list //
      DataList.push_back(Data);
   }

   // close file that read //
   Read.close();

   // sort the games by date //
   DateSort();

   // generate the game descriptions for the game collection //
   GameDescriptions();

   // if there is at least one game in the data list //
   if (DataList.size() > 0)
      // return that the game data file exists and have data //
      return true;
   // there are no games in the data list //
   else
      // return that had data file but have no valid games in it //
      return false;
}

// remove the non ascii characters from a string //
std::string ChessData::RemoveNonAscii(std::string StringToClean)
{
   // loop over the length of the string to clean //
   for (Index = 0; Index < StringToClean.length(); Index++)
      // if have encountered a carriage return from a windows created file //
      if (StringToClean.at(Index) == '\r')
         // assign a carriage return a value of @ "
         StringToClean.at(Index) = '@';
      // if character is not an alphanumeric or a punctuation character //
      else if (!isalnum(StringToClean.at(Index)) && !ispunct(StringToClean.at(Index)))
         // set the non-ascii character to a space //
         StringToClean.at(Index) = ' ';

   // return the string cleaned of non ascii characters and carriage returns //
   return ReplaceSubstring(StringToClean, "@", "");
}

// store the game header data for the game //
void ChessData::StoreGameHeader(std::string GameHeader)
{
   // store the game header information for the game after removing non-ascii //
   Header = RemoveNonAscii(GameHeader);
}

// store the move notation data for the game //
void ChessData::StoreGameNotation(std::string GameNotation)
{
   // if the game notation was empty //
   if (GameNotation.empty())
      // set game notation to * to show that there were no moves //
      GameNotation = "*";

   // store the move notation data for the game after removing non-ascii //
   Notation = RemoveNonAscii(GameNotation);
}

// date as a single number //
int ChessData::SortingDate()
{
   // return date as a single number for sorting purposes //
   return DateNumber;
}

// total number of plies in the game //
int ChessData::TotalPlies()
{
   // return number of plies in the game //
   return MoveList.size();
}

// create game description //
std::string ChessData::GameDescription()
{
   // if white is the color to move first //
   if (FENColor == "w")
      // get the total number of moves in the game //
      TotalMoves = (TotalPlies() + 1) / 2;
   // black is to move first //
   else
      // get the total number of moves in the game //
      TotalMoves = TotalPlies() / 2 + 1;

   // return game description //
   return White + " - " + Black + " on " + LongDate + " (" + Result + ", " +
          IntToString(TotalMoves + atoi(FENStartMoveNumber.c_str()) - 1) + " moves)";
}

// initialize attributes //
void ChessData::InitializeAttributes()
{
   // clear the time control for the game //
   TimeControl.clear();

   // clear the opening for the game //
   Opening.clear();

   // clear the section for the game //
   Section.clear();

   // clear the board number for the game //
   BoardNumber.clear();

   // clear the white elo of the game //
   WhiteElo.clear();

   // clear the black elo of the game //
   BlackElo.clear();

   // clear the white uscf of the game //
   WhiteUSCF.clear();

   // clear the black uscf of the game //
   BlackUSCF.clear();

   // clear the annotator of the game //
   Annotator.clear();

   // clear the eco of the game //
   ECO.clear();

   // clear the year string //
   YearString.clear();

   // clear the month string //
   MonthString.clear();

   // clear the day string //
   DayString.clear();

   // clear the header string //
   Header.clear();

   // clear the notation string //
   Notation.clear();

   // clear the move annotation list //
   AnnotationList.clear();

   // clear the list of moves //
   MoveList.clear();

   // clear the event string //
   Event.clear();

   // clear the site string //
   Site.clear();

   // clear the date string //
   Date.clear();

   // clear the round string //
   Round.clear();

   // clear the white user name for the game //
   White.clear();

   // clear the black user name for the game //
   Black.clear();

   // clear the result string //
   Result.clear();

   // set the initial fen board layout //
   FENLayout = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR";

   // set the initial fen castling rights //
   FENCastlingRights = "KQkq";

   // set the initial fen starting color to move first //
   FENColor = "w";

   // set the initial fen enpassant square //
   FENEnpassantSquare = "-";

   // the starting fen move number for the game //
   FENStartMoveNumber = "1";

   // create the full fen string //
   FENString = FENLayout + " " + FENColor + " " + FENCastlingRights + " " +
               FENEnpassantSquare + " 0 " + FENStartMoveNumber;
}

// store the game header attributes into the game data //
void ChessData::StoreGameHeader(std::string WhiteName, std::string BlackName,
                                std::string SiteLocation, std::string EventTitle,
                                std::string RoundNumber, std::string EndResult,
                                std::string GameDate, std::string TimingControl,
                                std::string WhiteEloValue, std::string BlackEloValue,
                                std::string WhiteUSCFValue, std::string BlackUSCFValue,
                                std::string AnnotatorValue, std::string ECOValue,
                                std::string OpeningValue, std::string SectionValue,
                                std::string BoardNumberValue, std::string FENStringInfo)
{
   // set the fen string for the header //
   FENString = FENStringInfo;

   // process the fen string //
   ProcessFENString();

   // get the name of the white player //
   White = WhiteName;

   // get the name of the black player //
   Black = BlackName;

   // get the site location of the game or tournament //
   Site = SiteLocation;

   // get the title of the tournament event //
   Event = EventTitle;

   // get the round number of the game //
   Round = RoundNumber;

   // get the end result of the game //
   Result = EndResult;

   // get the date the game was played on //
   Date = GameDate;

   // get the time control for the game //
   TimeControl = TimingControl;

   // get the opening for the game //
   Opening = OpeningValue;

   // get the section for the game //
   Section = SectionValue;

   // get the board number for the game //
   BoardNumber = BoardNumberValue;

   // get the white elo of the game //
   WhiteElo = WhiteEloValue;

   // get the black elo of the game //
   BlackElo = BlackEloValue;

   // get the white uscf of the game //
   WhiteUSCF = WhiteUSCFValue;

   // get the black uscf of the game //
   BlackUSCF = BlackUSCFValue;

   // get the annotator of the game //
   Annotator = AnnotatorValue;

   // get the eco of the game //
   ECO = ECOValue;
}

// store move into move list and move annotation into the annotation list //
void ChessData::StoreMove(std::string MadeMove, std::string Annotation)
{
   // push made move onto move list //
   MoveList.push_back(MadeMove);

   // push the move annotation onto the move annotation list //
   AnnotationList.push_back(Annotation);
}

// append pgn game data file to preexisting pgn data file //
void ChessData::AppendPGN(std::string FileName, bool BlackMoves)
{
   // get whether black is moving first in the game or not //
   BlackMovesFirst = BlackMoves;

   // create an output file stream //
   std::ofstream Write;

   // open file to append game data to //
   Write.open(FileName.c_str(), std::ios::app);

   // construct pgn representation of game //
   ConstructPGN();

   // write pgn data to game file //
   Write << "\n" << PGN;

   // close the game data that appended current game to //
   Write.close();
}

// write pgn game data file //
void ChessData::WritePGN(std::string FileName, bool BlackMoves)
{
   // get whether black moves first in the game or not //
   BlackMovesFirst = BlackMoves;

   // create an output file stream //
   std::ofstream Write;

   // open file to write game data to //
   Write.open(FileName.c_str());

   // construct pgn representation of game //
   ConstructPGN();

   // write pgn data to game file //
   Write << PGN;

   // close game data file that wrote to //
   Write.close();
}

// construction the annotation string //
std::string ChessData::ConstructAnnotations()
{
   // initialize the annotations string //
   Annotations = "";

   // loop over the entire annotation list //
   for (Index = 0; Index < AnnotationList.size(); Index++)
   {
      // add the annotation to the annotations string //
      Annotations += AnnotationList.at(Index);

      // if not at the end of the annotation list //
      if (Index != AnnotationList.size() - 1)
         // add a separator to the individual annotations in the annotations string //
         Annotations += "*****";
   }

   // return the annotations string //
   return Annotations;
}

// construct the game in uci format //
std::string ChessData::ConstructUCI()
{
   // initialize the board //
   Board.InitializeAttributes(FENLayout, FENCastlingRights, FENColor,
                              FENEnpassantSquare);

   // initialize the uci string //
   UCI = "";

   // loop over the entire move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
   {
      // check if the current move exists in the move list //
      if (Board.FindMove(*MoveIter))
      {
         // make the current move in the move list //
         Board.MakeMove(0);

         // add the move the the uci string in uci notation //
         UCI += Board.UCILabel();
      }
      // check if the pgn move was ambiguous //
      else if (MoveIter->length() >= 4)
      {
         // replace the second character in the move for case such as Rcc4/Rcc4 //
         (*MoveIter).replace(1, 1, "");

         // if able to find the disambiguated move //
         if (Board.FindMove(*MoveIter))
         {
            // make the current move in the move list //
            Board.MakeMove(0);

            // add the move the the uci string in uci notation //
            UCI += Board.UCILabel();
         }
         // the current move does not exist in the move list //
         else
            // add a nonexistent uci move value to uci string //
            UCI += "&";
      }
      // the current move does not exist in the move list //
      else
         // add a nonexistent uci move value to uci string //
         UCI += "&";

      // if not the last move in the move list //
      if (MoveIter != MoveList.end() - 1)
         // add a space to separate moves in the uci string //
         UCI += " ";
   }

   // return the game in uci notation //
   return UCI;
}

// construct pgn string representing game //
void ChessData::ConstructPGN()
{
   // initialize number of move plies //
   Plies = BlackMovesFirst;

   // add header information to pgn string //
   PGN = "[Event \"" + Event + "\"]\n" + "[Site \"" + Site + "\"]\n" +
         "[Date \"" + Date + "\"]\n" + "[Round \"" + Round + "\"]\n" +
         "[White \"" + White + "\"]\n" + "[Black \"" + Black + "\"]\n" +
         "[Result \"" + Result + "\"]\n" + "[TimeControl \"" + TimeControl + "\"]\n" +
         "[WhiteElo \"" + WhiteElo + "\"]\n" + "[BlackElo \"" + BlackElo + "\"]\n" +
         "[WhiteUSCF \"" + WhiteUSCF + "\"]\n" + "[BlackUSCF \"" + BlackUSCF + "\"]\n"
         + "[Annotator \"" + Annotator + "\"]\n" + "[ECO \"" + ECO + "\"]\n" +
         "[Opening \"" + Opening + "\"]\n" + "[Section \"" + Section + "\"]\n" +
         "[Board \"" + BoardNumber + "\"]\n" + "[SetUp \"1\"]\n" +
         "[FEN \"" + FENString + "\"]\n\n";

   // loop over move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
   {
      // if is the first move and black happens to make first move from fen info //
      if (Plies == 1 && BlackMovesFirst)
         // add to pgn string that black is moving first such as from fen game //
         PGN += FENStartMoveNumber + "... ";

      // if starting a new move number //
      if (++Plies % 2 == 1)
         // add move number to pgn string //
         PGN += IntToString((Plies - 1) / 2 + atoi(FENStartMoveNumber.c_str())) + ". ";

      // add move to pgn string //
      PGN += *MoveIter;

      // if the move annotation is currently not empty //
      if (!AnnotationList.at(MoveIter - MoveList.begin()).empty())
         // add the move annotation to the pgn string //
         PGN += " " + AnnotationList.at(MoveIter - MoveList.begin());

      // if have added another five move sequences or at end of move list //
      if (Plies % 10 == 0 || MoveIter + 1 == MoveList.end())
         // add end of line to pgn string //
         PGN += "\n";
      // need to write a space after move //
      else
         // add space to pgn string //
         PGN += " ";
   }

   // if the move list was empty //
   if (MoveList.empty())
      // store an * to represent no moves in game //
      PGN += "*\n";

   // append result and newline to pgn string //
   PGN += Result + "\n";
}

// convert an integer to a string //
std::string ChessData::IntToString(int Number)
{
   // create a string stream //
   std::stringstream Stream;

   // send the number into the string stream //
   Stream << Number;

   // return the string stream as a string //
   return Stream.str();
}

// get the fen string from the game data //
std::string ChessData::GetFENString()
{
   // get the fen string from the game data //
   return FENString;
}

// get the time control for the game //
std::string ChessData::GetTimeControl()
{
   // return the time control for the game //
   return TimeControl;
}

// get the opening for the game //
std::string ChessData::GetOpening()
{
   // return the opening for the game //
   return Opening;
}

// get the section for the game //
std::string ChessData::GetSection()
{
   // return the section for the game //
   return Section;
}

// get the board number for the game //
std::string ChessData::GetBoardNumber()
{
   // return the board number for the game //
   return BoardNumber;
}

// get the white elo of the game //
std::string ChessData::GetWhiteElo()
{
   // return the white elo of the game //
   return WhiteElo;
}

// get the black elo of the game //
std::string ChessData::GetBlackElo()
{
   // return the black elo of the game //
   return BlackElo;
}

// get the white uscf of the game //
std::string ChessData::GetWhiteUSCF()
{
   // return the white uscf of the game //
   return WhiteUSCF;
}

// get the black uscf of the game //
std::string ChessData::GetBlackUSCF()
{
   // return the black uscf of the game //
   return BlackUSCF;
}
```

```
// get the annotator of the game //
std::string ChessData::GetAnnotator()
{
   // return the annotator of the game //
   return Annotator;
}

// get the eco of the game //
std::string ChessData::GetECO()
{
   // return the eco of the game //
   return ECO;
}

// get the site location for the game //
std::string ChessData::GetSite()
{
   // return the site location for the game //
   return Site;
}

// get the round number for the game //
std::string ChessData::GetRound()
{
   // return the round number for the game //
   return Round;
}

// get the event title for the game //
std::string ChessData::GetEvent()
{
   // return the event title for the game //
   return Event;
}

// get the end result of the game //
std::string ChessData::GetResult()
{
   // return the end result of the game //
   return Result;
}

// get the date the game was played //
std::string ChessData::GetDate()
{
   // return the date the game was played //
   return Date;
}

// get name of white player //
std::string ChessData::WhitePlayer()
{
   // return name of white player //
   return White;
}

// get name of black player //
std::string ChessData::BlackPlayer()
{
   // return name of black player //
   return Black;
}

// process date string into a user readable format //
void ChessData::ProcessDate()
{
   // create a string stream //
   std::stringstream Stream;

   // clear string stream //
   Stream.clear();

   // remove periods/question marks from date string and send into string stream //
   Stream << ReplaceSubstring(ReplaceSubstring(Date, "?", "0"), ".", " ");

   // send string stream into year number //
   Stream >> Year;

   // send string stream into month number to get rid of zero padding //
   Stream >> Month;

   // send string stream into day number to get rid of zero padding //
   Stream >> Day;

   // if is the first month //
   if (Month == 1)
      // assign the name of january //
      MonthString = "January";
   // if is the second month //
   else if (Month == 2)
      // assign the name of february //
      MonthString = "February";
   // if is the third month //
   else if (Month == 3)
      // assign the name of march //
      MonthString = "March";
   // if is the fourth month //
   else if (Month == 4)
      // assign the name of april //
      MonthString = "April";
   // if is the fifth month //
   else if (Month == 5)
      // assign the name of may //
      MonthString = "May";
   // if is the sixth month //
   else if (Month == 6)
      // assign the name of june //
      MonthString = "June";
   // if is the seventh month //
   else if (Month == 7)
      // assign the name of july //
      MonthString = "July";
   // if is the eighth month //
   else if (Month == 8)
      // assign the name of august //
      MonthString = "August";
   // if is the ninth month //
   else if (Month == 9)
      // assign the name of september //
      MonthString = "September";
   // if is the tenth month //
   else if (Month == 10)
      // assign the name of october //
      MonthString = "October";
   // if is the eleventh month //
   else if (Month == 11)
      // assign the name of november //
      MonthString = "November";
   // if is the twelfth month //
   else if (Month == 12)
      // assign the name of december //
      MonthString = "December";
   // have an unknown month outside of range 1 to 12 //
   else
      // assign a question mark to the month string //
      MonthString = "?";

   // convert year to string //
   YearString = IntToString(Year);

   // convert day to string //
   DayString = IntToString(Day);

   // if day string is an unknown day //
   if (DayString == "0")
      // let the day string be a question mark //
      DayString = "?";

   // descriptive version of date //
   LongDate = MonthString + " " + DayString + ", " + YearString;

   // date as a single number for sorting purposes //
   DateNumber = Year * int(1E4) + Month * 100 + Day;
}

// process the pgn portion of the game data //
bool ChessData::ProcessPGN()
{
   // read descriptive fields from file header //
   GetFields();

   // extract the portable game notation from the notation string //
   ExtractMoves();

   // process the date the game was played on //
   ProcessDate();

   // set move iterator to point to beginning of move list //
   MoveIter = MoveList.begin();

   // if move list is empty //
   if (MoveList.empty())
      // return that file was empty //
      return false;
   // move list is not empty //
   else
      // return that were able to successfully read file //
      return true;
}

// process the fen string //
void ChessData::ProcessFENString()
{
   // create a string stream //
   std::stringstream Stream;

   // clear the string stream //
   Stream.clear();

   // send the fen string to the stream //
   Stream << FENString;

   // extract the fen layout from the stream //
   Stream >> FENLayout;

   // extract the fen color from the stream //
   Stream >> FENColor;

   // extract the fen castling rights from the stream //
   Stream >> FENCastlingRights;

   // extract the fen en passant square from the stream //
   Stream >> FENEnpassantSquare;

   // extract the fen fifty move rule from the stream //
   Stream >> FENFiftyMoveRule;

   // extract the fen start move number from the stream //
   Stream  >> FENStartMoveNumber;
}

// replace all occurrences of the substring in the string to modify //
std::string ChessData::ReplaceSubstring(std::string StringToModify, std::string
                                        Substring, std::string ReplacementSubstring)
{
   // do while true //
   while (true)
   {
      // get the starting index of the substring to replace //
      Index = StringToModify.find(Substring, 0);

      // if didn't find the substring to replace //
      if (Index == std::string::npos)
         // exit the while loop since didn't find substring to replace //
         break;

      // replace the substring in the string //
      StringToModify.replace(Index, Substring.length(), ReplacementSubstring);
   }

   // return the modified string //
   return StringToModify;
}

// extract portable game notation from the notation string //
void ChessData::ExtractMoves()
{
   // replace incorrect occurrences of queenside castling with zeros in them //
   Notation = ReplaceSubstring(Notation, "0-0-0", "O-O-O");

   // replace incorrect occurrences of kingside castling with zeros in them //
   Notation = ReplaceSubstring(Notation, "0-0", "O-O");

   // replace possible en passant move notation with nothing //
   Notation = ReplaceSubstring(Notation, "e.p.", "");

   // replace possible en passant move notation with nothing //
   Notation = ReplaceSubstring(Notation, "E.P.", "");

   // initialize the string iterator //
   StringIter = Notation.begin();

   // do while have not reached end of notation string //
   do
   {
      // if have encountered a dollar sign //
      if (*StringIter == '$')
         // do while have not encounted a space and not end of notation string //
         while (*StringIter != ' ' && StringIter != Notation.end())
            // increment the string iterator //
            StringIter++;
      // if have encountered a space //
      else if (*StringIter == ' ')
         // while have encountered another space and not end of notation string //
         while (*StringIter == ' ' && StringIter != Notation.end())
            // increment the string iterator //
            StringIter++;
      // if have encountered a digit //
      else if (isdigit(*StringIter))
         // while have not encountered an alpha character and not end of string //
         while (!isalpha(*StringIter) && StringIter != Notation.end())
            // increment the string iterator //
            StringIter++;
      // if have encountered a left bracket or left parentheses //
      else if (*StringIter == '{' || *StringIter == '(')
      {
         // set number of left parentheses encountered to zero //
         LeftParentheses = 0;

         // set number of right parentheses to zero //
         RightParentheses = 0;

         // set number of left brackets to zero //
         LeftBrackets = 0;

         // set number of right brackets to zero //
         RightBrackets = 0;

         // get the starting position in the notation string //
         Start = StringIter - Notation.begin();

         // do while still processing a move annotation //
         do
         {
            // if have encountered a left bracket //
            if (*StringIter == '{')
               // increment the number of left brackets encountered so far //
               LeftBrackets++;
            // if have encountered a right bracket //
            else if (*StringIter == '}')
               // increment the number of right brackets encountered so far //
               RightBrackets++;
            // if have encountered a left parentheses //
            else if (*StringIter == '(')
               // increment the number of left parentheses encountered so far //
               LeftParentheses++;
            // if have encountered a right parentheses //
            else if (*StringIter == ')')
               // increment the number of right parentheses encountered so far //
               RightParentheses++;

            // increment the string iterator //
            StringIter++;
         // while still going thru the nested annotation //
         } while (!(LeftParentheses == RightParentheses && LeftBrackets ==
                 RightBrackets) && StringIter != Notation.end());

         // if the annotation list already exists //
         if (AnnotationList.size() > 0)
         {
            // get the current move annotation string to add to //
            AnnotationString = AnnotationList.at(AnnotationList.size() - 1);

            // if the move annotation string size is not zero //
            if (AnnotationString.size() != 0)
               // add a space to separate multiple annotations for same move //
               AnnotationString += " ";

            // get the corresponding move annotation from the notation string //
            AnnotationString += Notation.substr(Start, StringIter - Notation.begin()
                                - Start);

            // set the annotation string in the annotation list //
            AnnotationList.at(AnnotationList.size() - 1) = AnnotationString;
          }
      }
      // if have encountered an alpha character //
      else if (isalpha(*StringIter))
      {
         // get the starting position in the notation string //
         Start = StringIter - Notation.begin();

         // do while have not encountered space, ?, !, or end of notation string //
         while (*StringIter != ' ' && *StringIter != '!' && *StringIter != '?' &&
               StringIter != Notation.end())
            // increment the string iterator //
            StringIter++;

         // add the move to the move list //
         MoveList.push_back(Notation.substr(Start, StringIter - Notation.begin()
                            - Start));

         // add an empty annotation to the move annotation list //
         AnnotationList.push_back("");
      }
      // have encountered an extraneous character //
      else
         // increment the string iterator //
         StringIter++;
   // while have not reached the end of the notation string //
   } while(StringIter != Notation.end());
}

// get descriptive fields from game header //
void ChessData::GetFields()
{
   // get the time control field from the game header //
   TimeControl = HeaderField("[TimeControl");

   // get the opening field from the game header //
   Opening = HeaderField("[Opening");

   // get the section field from the game header //
   Section = HeaderField("[Section");

   // get the board number from the game header //
   BoardNumber = HeaderField("Board");

   // get the white elo field from the game header //
   WhiteElo = HeaderField("[WhiteElo");

   // get the black elo field from the game header //
   BlackElo = HeaderField("[BlackElo");

   // get the white uscf field from the game header //
   WhiteUSCF = HeaderField("[WhiteUSCF");

   // get the black uscf field from the game header //
   BlackUSCF = HeaderField("[BlackUSCF");

   // get the annotator field from the game header //
   Annotator = HeaderField("[Annotator");

   // get the eco field from the game header //
   ECO = HeaderField("[ECO");

   // get the event field from game header //
   Event = HeaderField("[Event");

   // get the site field from game header //
   Site = HeaderField("[Site");

   // get the date field from game header //
   Date = HeaderField("[Date");

   // get the round field from game header //
   Round = HeaderField("[Round");

   // get who is playing as white from game header //
   White = HeaderField("[White");

   // get who is playing as black from game header //
   Black = HeaderField("[Black");

   // get the outcome of the game from game header //
   Result = HeaderField("[Result");

   // if have a fen line in the pgn game header //
   if (HeaderField("[FEN").length() > 0)
   {
      // extract the fen string from the game header //
      FENString = HeaderField("[FEN");

      // process the fen string //
      ProcessFENString();
   }
}

// extract a field from the game header //
std::string ChessData::HeaderField(std::string Tag)
{
   // get position of tag to look for in game header //
   Start = Header.find(Tag, 0);

   // if were not able to find tag in game header //
   if (Start < 0 || Start >= int(Header.size()))
      // return empty string //
      return ("");

   // get position in header where left boundary of selection is //
   Start = Header.find("\"", Start);

   // get position in header where right boundary of selection is //
   End = Header.find("\"", Start + 1);

   // get the header substring from the game header //
   HeaderSubstring = Header.substr(Start + 1, End - Start - 1);

   // return the extracted substring from the game header //
   return HeaderSubstring;
}

// mark moves that can cause a check on the next move //
void ChessBoard::MarkMovesThatCauseCheck()
{
   // loop over all the moves in the list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if the move causes a check when it moves to this square //
      if (SquareList.at(MoveIter->UpdatedPosition()).PieceCanCheckHere(MoveIter->
         RetrieveUpdatedPiece().PieceType()))
         // mark that move can cause a check if it moves to this square //
         MoveIter->MarkCanCheck();
}

// sort the move list for optimal alpha-beta search //
void ChessBoard::SortMoves(int BestMoveForSorting, std::vector<int> KillerMoves)
{
   // save the move list to a temporary move list //
   TempMoveList = MoveList;

   // clear the move list //
   MoveList.clear();

   // loop over all the moves in the temporary move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if the move matches the best move for sorting //
      if (BestMoveForSorting == MoveIter->MoveLabel())
      {
         // add the best move for sorting to the move list //
         MoveList.push_back(*MoveIter);

         // erase the best move for sorting from the temporary move list //
         TempMoveList.erase(MoveIter);

         // exit loop since already found the best move for sorting //
         break;
      }

   // loop over all the killer moves for current depth in game tree //
   for (Index = 0; Index < KillerMoves.size(); Index++)
      // loop over all moves in the temporary move list //
      for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
         // if the move matches one of the killer moves //
         if (MoveIter->MoveLabel() == KillerMoves.at(Index))
         {
            // add the killer move from the move list //
            MoveList.push_back(*MoveIter);

            // erase the killer move from the temporary move list //
            TempMoveList.erase(MoveIter);

            // exit loop since already found the killer move //
            break;
         }

   // loop over all the moves in the temporary move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if the move is a capture //
      if (MoveIter->IsCapture())
         // if the piece captures a piece of greater value //
         if ((MoveIter->CapturedPieceValue() > MoveIter->PieceValue()) && MoveIter->
            IsPawn())
            // add the move to the move list //
            MoveList.push_back(*MoveIter);

   // loop over all the moves in the temporary move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if the move is a capture //
      if (MoveIter->IsCapture())
         // if the piece captures a piece of greater value //
         if ((MoveIter->CapturedPieceValue() > MoveIter->PieceValue()) && !MoveIter->
            IsPawn())
            // add the move to the move list //
            MoveList.push_back(*MoveIter);

   // loop over all the moves in the temporary move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if the move is a capture //
      if (MoveIter->IsCapture())
         // if the piece captures a piece of equal or lesser value //
         if (MoveIter->CapturedPieceValue() <= MoveIter->PieceValue())
            // add the move to the move list //
            MoveList.push_back(*MoveIter);

   // loop over all the moves in the temporary move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if the move is not a capture //
      if (MoveIter->NotCapture())
         // add the move to the move list //
         MoveList.push_back(*MoveIter);

   // set the move iterator back to the beginning of the move list //
   MoveIter = MoveList.begin();
}

// convert an integer to a string //
std::string ChessBoard::IntToString(int Number)
{
   // create a string stream //
   std::stringstream Stream;

   // send the number into the string stream //
   Stream << Number;

   // return the string stream as a string //
   return Stream.str();
}

// get the list of squares where pieces can be captured on //
std::string ChessBoard::GetListOfCaptureSquares()
{
   // clear the list of squares where pieces can be captured on //
   ListOfCaptureSquares = "";

   // loop over the list of moves //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if move is a capture //
      if (MoveIter->IsCapture())
         // add the piece position to the list of capture squares and add separator //
         ListOfCaptureSquares += IntToString(MoveIter->CapturedPieceSquare()) + "*";

   // return the list of squares where pieces can be captured on //
   return ListOfCaptureSquares;
}

// get the list of squares that pieces can be threatened on //
std::string ChessBoard::GetListOfThreatenedSquares()
{
   // switch the color moving so can look for threatened pieces on the board //
   Color = !Color;

   // generate the list of possible moves //
   GenerateMoves();

   // return the list of squares where pieces can be threatened on //
   return GetListOfCaptureSquares();
}

// initialize the zobrist keys //
void ChessBoard::InitializeZobristKeys()
{
   // seed the random number generator so it always makes same sequence each time //
   srand(1);

   // loop over all the zobrist keys //
   for (Index = 0; Index < ZobristKeys.size(); Index++)
      // create a 64 bit zobrist key from a pseudo-random number generator //
      ZobristKeys.at(Index) = (double) rand() / (double) RAND_MAX * pow(2, 64);
}

// get the zobrist hash value for the board position //
unsigned long long ChessBoard::GetZobristHash()
{
   // return the zobrist hash value for the board position //
   return ZobristHash;
}

// compute the zobrist hash for the current board position //
void ChessBoard::ComputeZobristHash()
{
   // initialize the zobrist hash value to zero //
   ZobristHash = 0;

   // loop over all the squares on the board //
   for (SquareIter = SquareList.begin(); SquareIter != SquareList.end(); SquareIter++)
   {
      // if the square is currently unoccupied //
      if (SquareIter->IsUnoccupied())
         // xor the zobrist hash with a zobrist key for an empty square //
         ZobristHash ^= ZobristKeys.at(13 * (SquareIter - SquareList.begin()));
      // the square is occupied //
      else
         // xor the zobrist hash with a zobrist key for the piece on square //
         ZobristHash ^= ZobristKeys.at(SquareIter->GetZobristKeyIndex() + 13 *
                        (SquareIter - SquareList.begin()));

      // if encountered an en passant eligible pawn on the board //
      if (SquareIter->IsEnpassantOccupied())
         // loop over all the moves //
         for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
         {
            // if there is a move that captures en passant //
            if (MoveIter->IsEnpassant())
            {
               // xor the zobrist hash with a zobrist key for the en passant file //
               ZobristHash ^= ZobristKeys.at(846 + SquareIter->EnpassantFile());

               // exit loop since already added the en passant zobrist hash //
               break;
            }
         }
   }

   // if white is to move //
   if (IsWhite())
      // xor the zobrist hash with the zobrist key representing white to move next //
      ZobristHash ^= ZobristKeys.at(832);
   // black is to move //
   else
      // xor the zobrist hash with the zobrist key representing black to move next //
      ZobristHash ^= ZobristKeys.at(833);

   // if a kingside castle is possible //
   if (KingsideCastlingPossible())
      // xor the zobrist hash with the zobrist key representing a kingside castle //
      ZobristHash ^= ZobristKeys.at(834);
   // a kingside castle is not possible //
   else
      // xor the zobrist hash with the zobrist key representing no kingside castle //
      ZobristHash ^= ZobristKeys.at(835);

   // if a queenside castle is possible //
   if (QueensideCastlingPossible())
      // xor the zobrist hash with the zobrist key representing a queenside castle //
      ZobristHash ^= ZobristKeys.at(836);
   // a queenside castle is not possible //
   else
      // xor the zobrist hash with the zobrist key representing no queenside castle //
      ZobristHash ^= ZobristKeys.at(837);

   // if white has the option to castle kingside //
   if (WhiteKingsideCastlingOption)
      // xor the zobrist hash with zobrist key for white kingside castling option //
      ZobristHash ^= ZobristKeys.at(838);
   // white does not have the option to castle kingside //
   else
      // xor the zobrist hash with the zobrist key for no white kingside castling //
      ZobristHash ^= ZobristKeys.at(839);

   // if black has the option to castle kingside //
   if (BlackKingsideCastlingOption)
      // xor the zobrist hash with zobrist key for a black kingside castling option //
      ZobristHash ^= ZobristKeys.at(840);
   // black does not have the option to castle kingside //
   else
      // xor the zobrist hash with the zobrist key for no black kingside castling //
      ZobristHash ^= ZobristKeys.at(841);

   // if white has the option to castle queenside //
   if (WhiteQueensideCastlingOption)
      // xor the zobrist hash with zobrist key for a white queenside castling option //
      ZobristHash ^= ZobristKeys.at(842);
   // white does not have the option to castle queenside //
   else
      // xor the zobrist hash with the zobrist key for no white queenside castling //
      ZobristHash ^= ZobristKeys.at(843);

   // if black has the option to castle queenside //
   if (BlackQueensideCastlingOption)
      // xor the zobrist hash with zobrist key for a black queenside castling option //
      ZobristHash ^= ZobristKeys.at(844);
   // black does not have the option to castle queenside //
   else
      // xor the zobrist hash with the zobrist key for no black queenside castling //
      ZobristHash ^= ZobristKeys.at(845);
}

// count the number of pawns on the file for the given rank range and color //
int ChessBoard::PawnCountForFile(int PawnFile, int MinRank, int MaxRank, bool
                                 PawnColor)
{
   // if the pawn count file range is off the board //
   if (PawnFile < 0 || PawnFile > 7 || MinRank < 0 || MinRank > 7 || MaxRank < 0 ||
      MaxRank > 7)
      // return zero since pawn file or rank range is not on the board //
      return 0;

   // initialize the pawn count to zero //
   PawnCount = 0;

   // loop over the rank range //
   for (Rank = MinRank; Rank <= MaxRank; Rank++)
   {
      // get the position on the board //
      Position = PawnFile + Rank * 8;

      // if a pawn of the specified color is on the board //
      if (SquareList.at(Position).IsOccupied() && (SquareList.at(Position).PieceColor()
         == PawnColor) && SquareList.at(Position).IsPawn())
         // increment the pawn count on this file //
         PawnCount++;
   }

   // return the pawn count for this file //
   return PawnCount;
}

// return the number of pawns on the specified file for the specified color //
int ChessBoard::PawnCountForFile(int PawnFile, bool PawnColor)
{
   // if file is off the board //
   if (PawnFile < 0 || PawnFile > 7)
      // return 0 since there are not pawns here //
      return 0;

   // if counting white pawns on this file //
   if (PawnColor)
      // return number of white pawns on this file //
      return WhitePawnFileCounts[PawnFile];
   // counting black pawns on this file //
   else
      // return number of black pawns on this file //
      return BlackPawnFileCounts[PawnFile];
}

// analyze the pawn structure on the board //
void ChessBoard::AnalyzePawnStructure()
{
   // loop over the pawn file count arrays //
   for (File = 0; File < 8; File++)
   {
      // set the number of white pawns on this file //
      WhitePawnFileCounts[File] = PawnCountForFile(File, 0, 7, true);

      // set the number of black pawns on this file //
      BlackPawnFileCounts[File] = PawnCountForFile(File, 0, 7, false);
   }

   // loop over all pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
   {
      // if the piece is currently active //
      if (PieceIter->IsActive() && PieceIter->IsPawn())
      {
         // if is a white pawn //
         if (PieceIter->IsWhite())
         {
            // set the minimum rank for checking for pawns on file //
            MinRank = PieceIter->PieceRank() + 1;

            // set the maximum rank for checking for pawns on file //
            MaxRank = 7;
         }
         // if is a black pawn //
         else
         {
            // set the minimum rank for checking for pawns on file //
            MinRank = 0;

            // set the maximum rank for checking for pawns on file //
            MaxRank = PieceIter->PieceRank() - 1;
         }

         // set whether pawn is a passed pawn or not //
         PieceIter->SetPassedPawnStatus(!PawnCountForFile(PieceIter->PieceFile() - 1,
                                        MinRank, MaxRank, !PieceIter->PieceColor()) &&
                                        !PawnCountForFile(PieceIter->PieceFile(),
                                        MinRank, MaxRank, !PieceIter->PieceColor()) &&
                                        !PawnCountForFile(PieceIter->PieceFile() + 1,
                                        MinRank, MaxRank, !PieceIter->PieceColor()));

         // set whether pawn is a doubled pawn or not //
         PieceIter->SetDoubledPawnStatus(PawnCountForFile(PieceIter->PieceFile(),
                                         PieceIter->PieceColor()) == 2);

         // set whether pawn is a tripled pawn or not //
         PieceIter->SetTripledPawnStatus(PawnCountForFile(PieceIter->PieceFile(),
                                         PieceIter->PieceColor()) > 2);

         // set whether pawn is an isolated pawn or not //
         PieceIter->SetIsolatedPawnStatus(!PawnCountForFile(PieceIter->PieceFile() - 1,
                                          PieceIter->PieceColor()) && !PawnCountForFile
                                          (PieceIter->PieceFile() + 1, PieceIter->
                                          PieceColor()));
      }
   }
}

// get whether a king was potentially captured which is illegal //
bool ChessBoard::KingIsCaptured()
{
   // return whether a king was potentially captured which is illegal //
   return KingCaptured;
}

// condense move list down to only the unstable moves //
void ChessBoard::CondenseMoves()
{
   // if currently in a state of check //
   if (IsInCheck())
      // don't condense moves in quiescence portion of tree since currently in check //
      return;

   // copy the move list to the temp move list //
   TempMoveList = MoveList;

   // clear the move list //
   MoveList.clear();

   // loop over all moves //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if move captures a piece, promotes, or can cause a check //
      if (MoveIter->IsCapture() || MoveIter->CanCauseCheck() || MoveIter->
         IsPawnPromotion())
         // add the move to the move list //
         MoveList.push_back(*MoveIter);
}

// pgn label in portable game notation //
std::string ChessBoard::PGNLabel()
{
   // return pgn label in portable game notation //
   return MadeMove.PGNLabel();
}

// determine if an en passant move was just made //
bool ChessBoard::IsEnpassant()
{
   // return whether an en passant move was just made //
   return MadeMove.IsEnpassant();
}

// get uci label for move that was made //
std::string ChessBoard::UCILabel()
{
   // return uci label for move that was made //
   return MadeMove.UCILabel();
}

// convert move label to pgn representation //
std::string ChessBoard::LabelToPGN(long Label)
{
   // loop over all moves //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if label passed into function matches a move label in move list //
      if (Label == MoveIter->MoveLabel())
         // return pgn representation of move label //
         return MoveIter->PGNLabel();

   // return an empty string if the move label could not be matched to a move //
   return "";
}

// find and fix pgn duplicates //
void ChessBoard::PGNDuplicates(int PGNDetail)
{
   // set number of occurrences of each pgn label to zero //
   Occurrences = 0;

   // get the current version of the pgn label depending on the detail type //
   PGNLabelVersion = MoveIter->PGNLabelVersion(PGNDetail);

   // loop over all pgn labels in the list //
   for (PGNIter = PGNLabels.begin(); PGNIter != PGNLabels.end(); PGNIter++)
      // if the pgn label matches another label in the pgn vector //
      if (!std::strcmp(PGNLabelVersion.c_str(), PGNIter->c_str()))
      {
         // increment the number of occurrences of this particular pgn label //
         Occurrences++;

         // if there are two or more occurrences of this pgn label //
         if (Occurrences > 1)
         {
            // increase the pgn detail for the move analyzing pgn uniqueness for //
            MoveIter->SetPGNDetail(PGNDetail + 1);

            // exit loop since don't need to search for more pgn label occurrences //
            break;
         }
      }
}

// look for pgn ambiguities //
void ChessBoard::PGNAmbiguities()
{
   // look for any pgn duplicates at detail zero //
   PGNDuplicates(0);

   // if there are two or more pgn duplicates at detail zero //
   if (Occurrences > 1)
      // look for pgn label duplicates at detail one //
      PGNDuplicates(1);

   // if there are two or more pgn duplicates at detail one //
   if (Occurrences > 1)
      // look for pgn label duplicates at detail two //
      PGNDuplicates(2);
}

// resolve pgn label conflicts due to pgn label ambiguities //
void ChessBoard::PGNResolve()
{
   // clear the pgn labels //
   PGNLabels.clear();

   // loop over all the moves in the move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
   {
      // initialize the pgn detail //
      MoveIter->InitializePGN();

      // if move is not for a pawn or a king //
      if (!(MoveIter->IsPawn() || MoveIter->IsKing()))
         // add the pgn version zero label to the pgn labels list //
         PGNLabels.push_back(MoveIter->PGNLabelVersion(0));
   }

   // sort the list of pgn labels //
   std::sort(PGNLabels.begin(), PGNLabels.end());

   // set that there are initially no occurrences of duplicated pgn labels //
   Occurrences = 0;

   // loop over the entire list of sorted pgn labels //
   for (PGNIter = PGNLabels.begin(); PGNIter != PGNLabels.end(); PGNIter++)
      // if next label to compare is not at the end of the sorted pgn label list //
      if ((PGNIter + 1) != PGNLabels.end())
         // if there is a duplicate pgn label in the list of sorted pgn labels //
         if (!std::strcmp(PGNIter->c_str(), (PGNIter + 1)->c_str()))
         {
            // increment number of occurrences of duplicate pgn labels //
            Occurrences++;

            // exit loop since already found an occurrence of a duplicated pgn label //
            break;
         }

   // if there are no occurrences of a duplicate pgn label //
   if (Occurrences == 0)
      // exit function since there are no occurrences of a duplicate pgn label //
      return;

   // loop over all the moves in the move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if piece moving is not a pawn or a king //
      if (!(MoveIter->IsPawn() || MoveIter->IsKing()))
      {
         // push the version one pgn label onto the pgn labels list //
         PGNLabels.push_back(MoveIter->PGNLabelVersion(1));

         // push the version two pgn label onto the pgn labels list //
         PGNLabels.push_back(MoveIter->PGNLabelVersion(2));
      }

   // loop over all the moves in the move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if piece moving is not a pawn or a king //
      if (!(MoveIter->IsPawn() || MoveIter->IsKing()))
         // look for any pgn ambiguities and resolve them //
         PGNAmbiguities();

   // clear the pgn labels //
   PGNLabels.clear();
}

// determine if not in a state of check //
bool ChessBoard::NoCheck()
{
   // return if not in a state of check //
   return Check == -1;
}

// determine if in a state of single check //
bool ChessBoard::IsSingleCheck()
{
   // return if in a state of single check //
   return Check == 0;
}

// determine if in a state of double check //
bool ChessBoard::IsDoubleCheck()
{
   // return if in a state of double check //
   return Check == 1;
}

// whether the board is not currently in check //
bool ChessBoard::NotInCheck()
{
   // return whether the board is not currently in check //
   return !IsInCheck();
}

// whether the board is in check //
bool ChessBoard::IsInCheck()
{
   // return whether in single or double check //
   return IsSingleCheck() || IsDoubleCheck();
}

// determine if in a state of checkmate //
bool ChessBoard::IsCheckmate()
{
   // return if in a state of checkmate //
   return Check == 2;
}

// determine if in a state of stalemate //
bool ChessBoard::IsStalemate()
{
   // return if in a state of stalemate //
   return Check == 3;
}

// mark if rook on this square has already moved //
void ChessBoard::MarkRookMoved(int Position, bool Color)
{
   // loop over all pieces on the board //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end();
        PieceIter++)
      // if piece is a rook of the specified color and specified position //
      if (PieceIter->IsRook() && PieceIter->PieceColor() == Color && Position ==
         PieceIter->PiecePosition())
         // mark that piece has already moved //
         PieceIter->MarkMoved();
}

// create the pieces for the board from the FEN notation //
void ChessBoard::CreatePieces(std::string FENLayout, std::string
                              FENCastlingRights, std::string FENEnpassantSquare)
{
   // clear the piece list //
   PieceList.clear();

   // clear the FEN string //
   FEN.clear();

   // loop over the string containing the pieces in FEN notation //
   for (StringIter = FENLayout.begin(); StringIter != FENLayout.end(); StringIter++)
   {
      // get one element of the FEN string //
      FENElement = *StringIter;

      // check if the element of the FEN string is a number //
      if (isdigit(*StringIter))
         // loop over the number of blank characters to add to the FEN string //
         for (Index = 0; Index < atoi(FENElement.c_str()); Index++)
            // add blank characters onto the FEN string //
            FEN += " ";
      // if the FEN character is not a FEN notation row divider //
      else if (FENElement != "/")
         // add the current character onto the FEN string //
         FEN += FENElement;
   }

   // if the fen string was corrupt and was not 64 squares long //
   if (FEN.size() != 64)
      // reset the fen string to the default fen string //
      FEN = "rnbqkbnrpppppppp                                PPPPPPPPRNBQKBNR";

   // loop over half of FEN board layout string //
   for (StringIter = FEN.begin(); StringIter < FEN.begin() + 32; StringIter++)
      // swap row elements in FEN board layout string //
      std::swap(*StringIter, *(StringIter + (56 - 16 * ((StringIter - FEN.begin())
             / 8))));

   // loop over board representation of FEN string //
   for (StringIter = FEN.begin(); StringIter != FEN.end(); StringIter++)
   {
      // if character in FEN string is a white king //
      if (*StringIter == 'K')
         // create a white king //
         Piece.CreateWhiteKing(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black king //
      else if (*StringIter == 'k')
         // create a black king //
         Piece.CreateBlackKing(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a white rook //
      else if (*StringIter == 'R')
         // create a white rook //
         Piece.CreateWhiteRook(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black rook //
      else if (*StringIter == 'r')
         // create a black rook //
         Piece.CreateBlackRook(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a white knight //
      else if (*StringIter == 'N')
         // create a white knight //
         Piece.CreateWhiteKnight(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black knight //
      else if (*StringIter == 'n')
         // create a black knight //
         Piece.CreateBlackKnight(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a white bishop //
      else if (*StringIter == 'B')
         // create a white bishop //
         Piece.CreateWhiteBishop(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black bishop //
      else if (*StringIter == 'b')
         // create a black bishop //
         Piece.CreateBlackBishop(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a white queen //
      else if (*StringIter == 'Q')
         // create a white queen //
         Piece.CreateWhiteQueen(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black queen //
      else if (*StringIter == 'q')
         // create a black queen //
         Piece.CreateBlackQueen(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a white pawn //
      else if (*StringIter == 'P')
         // create a white pawn //
         Piece.CreateWhitePawn(StringIter - FEN.begin(), PieceList.size());
      // if character in FEN string is a black pawn //
      else if (*StringIter == 'p')
         // create a black pawn //
         Piece.CreateBlackPawn(StringIter - FEN.begin(), PieceList.size());

      // if character in FEN string is not empty //
      if (*StringIter != ' ')
      {
         // if the enpassant location of the piece is same as the enpassant square //
         if (Piece.EnpassantBoardLocation() == FENEnpassantSquare && Piece.IsPawn())
            // mark the piece as being eligible to be captured enpassant //
            Piece.MakeEnpassantEligible();

         // add the new piece onto the piece list //
         PieceList.push_back(Piece);
      }
   }

   // update the pieces just created in terms of initial piece movement //
   UpdateCreatedPieces(FENCastlingRights);
}

// update the pieces just created in terms of initial piece movement //
void ChessBoard::UpdateCreatedPieces(std::string FENCastlingRights)
{
   // initialize that white does not have kingside castling option //
   WhiteKingsideCastlingOption = false;

   // initialize that white does not have queenside castling option //
   WhiteQueensideCastlingOption = false;

   // initialize that black does not have kingside castling option //
   BlackKingsideCastlingOption = false;

   // initialize that black does not have queenside castling option //
   BlackQueensideCastlingOption = false;

   // loop over FEN castling rights string //
   for (StringIter = FENCastlingRights.begin(); StringIter != FENCastlingRights.end();
        StringIter++)
   {
      // if white has a kingside castling option //
      if (*StringIter == 'K')
         // set that white has a kingside castling option //
         WhiteKingsideCastlingOption = true;
      // if white has a queenside castling option //
      else if (*StringIter == 'Q')
         // set that white has a queenside castling option //
         WhiteQueensideCastlingOption = true;
      // if black has a kingside castling option //
      else if (*StringIter == 'k')
         // set that black has a kingside castling option //
         BlackKingsideCastlingOption = true;
      // if black has a queenside castling option //
      else if (*StringIter == 'q')
         //  set that black has a queenside castling option //
         BlackQueensideCastlingOption = true;
   }

   // if white does not have the option to castle kingside //
   if (!WhiteKingsideCastlingOption)
      // mark that white king's rook has already moved if it is active //
      MarkRookMoved(7, true);

   // if white does not have the option to castle queenside //
   if (!WhiteQueensideCastlingOption)
      // mark that white queen's rook has already moved if it is active //
      MarkRookMoved(0, true);

   // if black does not have the option to castle kingside //
   if (!BlackKingsideCastlingOption)
      // mark that black king's rook has already moved if it is active //
      MarkRookMoved(63, false);

   // if black does not have the option to castle queenside //
   if (!BlackQueensideCastlingOption)
      // mark that black queen's rook has already moved if it is active //
      MarkRookMoved(56, false);

   // loop over all pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
   {
      // if piece is a pawn not on starting rank, set that pawn has moved //
      if (PieceIter->IsPawn() && (PieceIter->IsWhite() && PieceIter->PieceRank() != 1
         || PieceIter->IsBlack() && PieceIter->PieceRank() != 6))
         // mark that pawn piece has already moved since not on starting rank //
         PieceIter->MarkMoved();
      // if piece is a king //
      else if (PieceIter->IsKing())
      {
         // if piece is white //
         if (PieceIter->IsWhite())
         {
            // get the index of the white king in the piece list //
            WhiteKingIndex = PieceIter->PieceIndex();

            // if white king is not in the initial starting position //
            if (PieceIter->PiecePosition() != 4)
               // mark that white king has already moved at least once //
               PieceIter->MarkMoved();
         }
         // piece is black //
         else
         {
            // get the index of the black king in the piece list //
            BlackKingIndex = PieceIter->PieceIndex();

            // if black king is not in the initial starting position //
            if (PieceIter->PiecePosition() != 60)
               // mark that black king has already moved at least once //
               PieceIter->MarkMoved();
         }
      }
   }

   // create a white pseudo king //
   Piece.CreateWhitePseudoKing(PieceList.at(WhiteKingIndex).PiecePosition(),
                               PieceList.size());

   // add the white pseudo king to the piece list //
   PieceList.push_back(Piece);

   // create a black pseudo king //
   Piece.CreateBlackPseudoKing(PieceList.at(BlackKingIndex).PiecePosition(),
                               PieceList.size());

   // add the black pseudo king to the piece list //
   PieceList.push_back(Piece);
}

// update the pseudo king positions on the board //
void ChessBoard::UpdatePseudoKingPositions()
{
   // loop over all the pieces on the baord //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // if the piece is the white pseudo king //
      if (PieceIter->IsPseudoKing() && PieceIter->IsWhite())
         // update the white pseudo king position to the white king position //
         PieceIter->UpdatePosition(PieceList.at(WhiteKingIndex).PiecePosition());
      // the piece is the black pseudo king //
      else if (PieceIter->IsPseudoKing() && PieceIter->IsBlack())
         // update the black pseudo king position to the black king position //
         PieceIter->UpdatePosition(PieceList.at(BlackKingIndex).PiecePosition());
}

// initialize the board attributes //
void ChessBoard::InitializeAttributes(std::string FENLayout, std::string
                                      FENCastlingRights, std::string
                                      FENColor, std::string FENEnpassantSquare)
{
   // set the initial score vector size to one //
   Score.resize(1);

   // set the initial zobrist hashes vector size to zero //
   ZobristHashes.resize(0);

   // set the initial zobrist hashes count vector size to zero //
   ZobristHashCounts.resize(0);

   // resize the zobrist keys to the correct value //
   ZobristKeys.resize(900);

   // initialize the zobrist keys //
   InitializeZobristKeys();

   // create sixty-four squares for the board //
   SquareList.resize(64);

   // get color of the side currently moving (white is true, black is false) //
   Color = (FENColor == "w");

   // white has not initially castled kingside //
   WhiteCastledKingside = false;

   // black has not initially castled kingside //
   BlackCastledKingside = false;

   // white has not initially castled queenside //
   WhiteCastledQueenside = false;

   // black has not initially castled queenside //
   BlackCastledQueenside = false;

   // create all the pieces pieces for the board //
   CreatePieces(FENLayout, FENCastlingRights, FENEnpassantSquare);

   // generate the moves for the current color moving //
   GenerateMoves();

   // resolve pgn labels //
   PGNResolve();

   // update the zobrist hash counts so can determine threefold repetition //
   UpdateZobristHashCounts();

   // compute numbers of each piece type on the board //
   ComputeNumbersOfEachPieceType();

   // compute number of pieces left on the board //
   ComputeNumberOfPiecesLeft();
}

// reinitialize the piece attributes //
void ChessBoard::ReinitializePieces()
{
   // loop over all pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // reinitialize piece attributes //
      PieceIter->ReinitializeAttributes(Color);
}

// build the paths the pieces move along on board //
void ChessBoard::BuildPaths()
{
   // loop over all pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end();
        PieceIter++)
      // if piece is currently active on the board or is a pseudo king //
      if (PieceIter->IsActive() || (PieceIter->IsPseudoKing() && PieceIter->
         NotSideMoving()))
         // loop over all the directions a piece can possibly move in //
         for (Direction = 0; Direction < 16; Direction++)
            // determine if piece can actually move in this direction //
            if (PieceIter->AllowedDirection(Direction))
               // generate all possible moves along this path //
               BuildPath();
}

// reinitialize en passant eligibilities //
void ChessBoard::ReinitializeEnpassant()
{
   // loop over all pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // reset piece to be ineligible to be captured en passant //
      PieceIter->MakeEnpassantIneligible();
}

// find all possible moves on the current board //
void ChessBoard::GenerateMoves()
{
   // set that not in a state of check, checkmate, or stalemate //
   Check = -1;

   // set that a special pin does not initially exist //
   SpecialPin = false;

   // set that initially, the king is not potentially captured //
   KingCaptured = false;

   // reinitialize the piece attributes //
   ReinitializePieces();

   // refresh the current board layout from the locations of the pieces //
   RefreshLayout();

   // empty the move list //
   MoveList.clear();

   // update the pseudo king position to the current king positions //
   UpdatePseudoKingPositions();

   // build the paths on the board which the pieces move along on //
   BuildPaths();

   // mark all the moves that can cause a check on the next move //
   MarkMovesThatCauseCheck();

   // reinitialize en passant eligibilities //
   ReinitializeEnpassant();

   // test the legality of each move and remove illegal moves from move list //
   TestLegality();

   // update the castling rights //
   UpdateCastlingRights();

   // generate any possible castling moves //
   GenerateCastlingMoves();

   // mark if in a state of stalemate or checkmate //
   StalemateCheckmateTest();

   // create labels for move list //
   MakeLabels();
}

// create labels for moves //
void ChessBoard::MakeLabels()
{
   // loop over all moves //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // generate unique move identification label //
      MoveIter->MakeLabel();
}

// determine if king can legally make this move //
bool ChessBoard::KingMoveLegality()
{
   // return whether king can legally make this move //
   return !(SquareList.at(MoveIter->UpdatedPosition()).IsAttacked(MoveIter->
          PieceColor()) || PieceList.at(MoveIter->PieceIndex()).DirectionallyAttacked
          (MoveIter->MoveDirection()));
}

// set whether in a state of checkmate or stalemate //
void ChessBoard::StalemateCheckmateTest()
{
   // if no legal moves exist in move list //
   if (NoMoves())
   {
      // if was previously in a state of check or double check //
      if (IsInCheck())
         // set that now in a state of checkmate //
         Check = 2;
      // if was not previously in a state of check //
      else
         // set that now in a state of stalemate //
         Check = 3;
   }
}

// determine if a move is illegal //
bool ChessBoard::LegalMove()
{
   // if move is an en passant capture and have a special pin on board //
   if (MoveIter->IsEnpassant() && SpecialPin)
      // return that move is illegal //
      return false;
   // if is a king moving //
   else if (MoveIter->IsKing())
      // return the legality of this move by a king //
      return KingMoveLegality();
   // if in a state of double check or violates a pin //
   else if (IsDoubleCheck() || ViolatesPin())
      // return that move is illegal //
      return false;

   // set that move does not initially capture the piece causing a check //
   CapturesChecker = false;

   // if move captures a piece //
   if (MoveIter->IsCapture())
      // determine if move captures a piece causing a check //
      CapturesChecker = PieceList.at(MoveIter->CapturedPieceIndex()).IsChecker();

   // if in a state of check and move does not block the check or capture checker //
   if (IsInCheck() && SquareList.at(MoveIter->UpdatedPosition()).NotBlock() &&
      !CapturesChecker)
      // return that move is illegal //
      return false;

   // return that move is legal //
   return true;
}

// determine whether move violates pin //
bool ChessBoard::ViolatesPin()
{
   // return whether move violates pin //
   return PieceList.at(MoveIter->PieceIndex()).ViolatesPin(MoveIter->MoveDirection());
}

// test the legality of the moves in the move list //
void ChessBoard::TestLegality()
{
   // copy the move list to a temporary move list //
   TempMoveList = MoveList;

   // clear the move list //
   MoveList.clear();

   // loop over the move list //
   for (MoveIter = TempMoveList.begin(); MoveIter != TempMoveList.end(); MoveIter++)
      // if move is legal //
      if (LegalMove())
         // remove the move from list of moves //
         MoveList.push_back(*MoveIter);
}

// retrieve the next move from move list to simulate //
bool ChessBoard::RetrieveNextMove()
{
   // if have not reached end of move list //
   if (MoveIter != MoveList.end())
   {
      // get the next move in the move list //
      Move = *MoveIter;

      // save the move that is made //
      MadeMove = Move;

      // move to the next element in the move list //
      MoveIter++;

      // return that were successfully able to retrieve the next move //
      return true;
   }
   // have reached end of move list //
   else
      // return that were not able to successfully retrieve the next move //
      return false;
}

// determine whether piece can move to this square along path //
bool ChessBoard::CanMovePieceToThisSquare()
{
   // increment the length the piece has moved along path //
   Length++;

   // initialize done moving to true //
   DoneMoving = true;

   // get the next position the piece can possibly move to along this path //
   Position = PieceIter->NextPosition(Direction, Length);

   // if have an invalid position value //
   if (Position == -1)
      // return that piece cannot move to this square along path //
      return false;

   // get iterator to square that piece moves to //
   SquareIter = SquareList.begin() + Position;

   // push this square onto the path the piece has moved along //
   Path.push_back(Position);

   // analyze the move //
   AnalyzeMove();

   // return whether not done moving along the path or not //
   return !DoneMoving;
}

// if black forfeits a castle //
bool ChessBoard::BlackForfeitsCastle()
{
   // if black has already castled //
   if (BlackCastledKingside || BlackCastledQueenside)
      // since has castled, return that does not forfeit ability to castle //
      return false;

   // return if black has no castling options left //
   return !(BlackKingsideCastlingOption || BlackQueensideCastlingOption);
}

// if white forfeits a castle //
bool ChessBoard::WhiteForfeitsCastle()
{
   // if white has already castled //
   if (WhiteCastledKingside || WhiteCastledQueenside)
      // since has castled, return that does not forfeit ability to castle //
      return false;

   // return if white has no castling options left //
   return !(WhiteKingsideCastlingOption || WhiteQueensideCastlingOption);
}

// determine if have encountered a friendly piece on the board //
bool ChessBoard::EncounteredFriend()
{
   // return whether have encountered a friendly piece on the board //
   return SquareIter->IsOccupied() && (SquareIter->PieceColor() == Move.PieceColor());
}

// determine if have encountered an enemy piece on the board //
bool ChessBoard::EncounteredEnemy()
{
   // return whether have encountered an enemy piece on the board //
   return SquareIter->IsOccupied() && (SquareIter->PieceColor() != Move.PieceColor());
}

// determine if have encountered an en passant eligible enemy piece on board //
bool ChessBoard::EncounteredEnpassantEnemy()
{
   // return whether have encountered an en passant eligible enemy piece //
   return SquareIter->IsEnpassantOccupied() && Move.IsPawn() && (SquareIter->
          EnpassantPieceColor() != Move.PieceColor());
}

// analyze move to square //
void ChessBoard::AnalyzeMove()
{
   // initialize move attributes //
   Move.InitializeAttributes(*PieceIter, Position, Direction, Length);

   // if the piece moving is a pseudo king //
   if (PieceIter->IsPseudoKing())
   {
      // if the pseudo king is moving like a knight //
      if (Direction % 2 == 1)
         // mark that a knight can move here and cause a check //
         SquareIter->MarkKnightCanCauseCheck();
      // if the pseudo king is moving like a rook //
      else if (Direction % 4 == 0)
      {
         // mark that a queen can move here and cause a check //
         SquareIter->MarkQueenCanCauseCheck();

         // mark that a rook can move here and cause a check //
         SquareIter->MarkRookCanCauseCheck();
      }
      // the pseudo king is moving like a bishop //
      else
      {
         // mark that a bishop can move here and cause a check //
         SquareIter->MarkBishopCanCauseCheck();

         // mark that a queen can move here and cause a check //
         SquareIter->MarkQueenCanCauseCheck();
      }

      // if pseudo king is moving like a pawn in the allowed direction //
      if (Length == 1 && ((PieceIter->IsWhite() && (Direction == 2 || Direction == 14))
         || (PieceIter->IsBlack() && (Direction == 6 || Direction == 10))))
         // mark that a pawn can move here and cause a check //
         SquareIter->MarkPawnCanCauseCheck();

      // if haven't encounted a piece along this path //
      if (!(EncounteredFriend() || EncounteredEnemy()))
         // set that not done moving since haven't encountered a piece along path //
         DoneMoving = false;
   }
   // if have encountered a friendly piece on this square //
   else if (EncounteredFriend())
      // analyze friendly piece encounter //
      AnalyzeFriend();
   // if have encountered an enemy piece on this square //
   else if (EncounteredEnemy())
      // analyze enemy piece encounter //
      AnalyzeEnemy();
   // if have encountered an en passant eligible enemy piece //
   else if (EncounteredEnpassantEnemy())
      // analyze encounter with en passant enemy piece //
      AnalyzeEnpassantEnemy();
   // if piece moving is a pawn //
   else if (PieceIter->IsPawn())
      // analyze pawn move to this square //
      AnalyzePawn();
   // if have encountered an empty square //
   else
      // analyze move to empty square //
      AnalyzeEmptySquare();
}

// build path that piece moves along on board //
void ChessBoard::BuildPath()
{
   // initialize path length to zero //
   Length = 0;

   // clear list of pieces encountered along path //
   PathPieces.clear();

   // clear path that piece moves along //
   Path.clear();

   // move piece to new square while can //
   while (CanMovePieceToThisSquare());
}

// determine if conditions are met for being able to do queenside castle //
bool ChessBoard::QueensideCastlingPossible()
{
   // if white is currently moving //
   if (IsWhite())
   {
      // get the queenside castling option //
      QueensideCastlingOption = WhiteQueensideCastlingOption;

      // set square iterator to position of white queen's rook //
      SquareIter = SquareList.begin();
   }
   // if black is moving //
   else
   {
      // get the queenside castling option //
      QueensideCastlingOption = BlackQueensideCastlingOption;

      // set square iterator to position of black queen's rook //
      SquareIter = SquareList.begin() + 56;
   }

   // return if conditions are met for being able to do queenside castle //
   return QueensideCastlingOption && (SquareIter + 1)->IsUnoccupied() && (SquareIter
          + 2)->IsUnoccupied() && (SquareIter + 2)->NotAttacked(Color) && (SquareIter
          + 3)->IsUnoccupied() && (SquareIter + 3)->NotAttacked(Color) && (SquareIter
          + 4)->NotAttacked(Color);
}

// compute a castling option for the specified color, king, and rook location //
bool ChessBoard::ComputeCastlingRight(bool PieceColor, int KingLocation, int
                                      RookLocation)
{
   // get the square the king is on //
   KingSquare = SquareList.begin() + KingLocation;

   // get the square the rook is on //
   RookSquare = SquareList.begin() + RookLocation;

   // return whether color has to option to castle //
   return KingSquare->NeverMoved() && (KingSquare->PieceColor() == PieceColor) &&
          KingSquare->IsOccupied() && KingSquare->IsKing() &&
          RookSquare->NeverMoved() && (RookSquare->PieceColor() == PieceColor) &&
          RookSquare->IsOccupied() && RookSquare->IsRook();
}

// update the castling rights //
void ChessBoard::UpdateCastlingRights()
{
   // if white already had the option to castle kingside //
   if (WhiteKingsideCastlingOption)
      // update the white kingside castling option //
      WhiteKingsideCastlingOption = ComputeCastlingRight(true, 4, 7);

   // if black already had the option to castle kingside //
   if (BlackKingsideCastlingOption)
      // update the black kingside castling option //
      BlackKingsideCastlingOption = ComputeCastlingRight(false, 60, 63);

   // if white already had the option to castle queenside //
   if (WhiteQueensideCastlingOption)
      // update the white queenside castling option //
      WhiteQueensideCastlingOption = ComputeCastlingRight(true, 4, 0);

   // if black already had the option to castle queenside //
   if (BlackQueensideCastlingOption)
      // update the black queenside castling option //
      BlackQueensideCastlingOption = ComputeCastlingRight(false, 60, 56);
}

// determine if conditions are met for being able to do kingside castle //
bool ChessBoard::KingsideCastlingPossible()
{
   // if white is currently moving //
   if (IsWhite())
   {
      // get the kingside castling option //
      KingsideCastlingOption = WhiteKingsideCastlingOption;

      // set square iterator to position of white king //
      SquareIter = SquareList.begin() + 4;
   }
   // if black is currently moving //
   else
   {
      // get the kingside castling option //
      KingsideCastlingOption = BlackKingsideCastlingOption;

      // set square iterator to position of black king //
      SquareIter = SquareList.begin() + 60;
   }

   // return if conditions are met for being able to do kingside castle //
   return KingsideCastlingOption && SquareIter->NotAttacked(Color) && (SquareIter + 1)
          ->IsUnoccupied() && (SquareIter + 1)->NotAttacked(Color) && (SquareIter + 2)
          ->IsUnoccupied() && (SquareIter + 2)->NotAttacked(Color);
}

// generate castling moves //
void ChessBoard::GenerateCastlingMoves()
{
   // if a queenside castle is possible //
   if (QueensideCastlingPossible())
   {
      // initialize move attributes //
      Move.InitializeAttributes(SquareIter->RetrievePieceOnSquare(), SquareIter->
                                PiecePosition() + 3, 4, 3);

      // mark move as a queenside castle //
      Move.CastleQueenside();

      // push move onto the move list //
      MoveList.push_back(Move);
   }

   // if a kingside castle is possible //
   if (KingsideCastlingPossible())
   {
      // initialize move attributes //
      Move.InitializeAttributes((SquareIter + 3)->RetrievePieceOnSquare(),
                                (SquareIter + 3)->PiecePosition() - 2, 12, 2);

      // mark move as a kingside castle //
      Move.CastleKingside();

      // push move onto the move list //
      MoveList.push_back(Move);
   }
}

// generate promotion moves //
void ChessBoard::GeneratePromotionMoves()
{
   // promote pawn to knight //
   Move.PromotePawnToKnight();

   // push move onto move list //
   MoveList.push_back(Move);

   // promote pawn to bishop //
   Move.PromotePawnToBishop();

   // push move onto move list //
   MoveList.push_back(Move);

   // promote pawn to rook //
   Move.PromotePawnToRook();

   // push move onto move list //
   MoveList.push_back(Move);

   // promote pawn to queen //
   Move.PromotePawnToQueen();

   // push move onto move list //
   MoveList.push_back(Move);
}

// determine if able to analyze move of piece to empty square //
void ChessBoard::AnalyzeEmptySquare()
{
   // if no pieces have been encountered along this path //
   if (PathPieces.empty())
   {
      // mark that move attacks square that moves to //
      SquareList.at(Position).AddAttackToSquare(Move.PieceColor());

      // add move to the move list if it is legal //
      AddToMoveListIfLegal();
   }

   // set that not done moving along this path yet //
   DoneMoving = false;
}

// determine if able to analyze pawn move //
void ChessBoard::AnalyzePawn()
{
   // if pawn is moving directly forward //
   if (Direction == 0 || Direction == 8)
   {
      // if moved two spaces forward //
      if (Length == 2)
         // mark that move makes pawn eligible to be captured en passant //
         Move.MakeEnpassantEligible();

      // add move to the move list if legal //
      AddToMoveListIfLegal();
   }
   // pawn is moving diagonally //
   else
      // mark that pawn diagonally attacks square it moves to //
      SquareList.at(Position).AddAttackToSquare(Move.PieceColor());

   // set that not done moving along this path yet //
   DoneMoving = false;
}

// determine if able to analyze encounter of en passant piece //
void ChessBoard::AnalyzeEnpassantEnemy()
{
   // mark that move captures an en passant elibible piece //
   Move.CaptureEnpassant();

   // store the en passant eligible piece captured //
   Move.StoreCapturedPiece(SquareIter->RetrieveEnpassantPiece());

   // mark that move attacks square //
   SquareList.at(Position).AddAttackToSquare(Move.PieceColor());

   // add to the move list is legal //
   AddToMoveListIfLegal();

   // push the en passant eligible piece onto the list of encountered pieces //
   PathPieces.push_back(SquareIter->RetrieveEnpassantPiece());
   // mark that piece is attacked by a pawn //
   PieceList.at(PathPieces.at(0).PieceIndex()).MarkAttackedByPawn();

   // set that not done moving along this path yet //
   DoneMoving = false;
}

// add move to the move list if it is legal //
void ChessBoard::AddToMoveListIfLegal()
{
   // increase the mobility of this piece //
   PieceList.at(Move.PieceIndex()).IncreaseMobility();

   // if move is for the side moving //
   if (Move.SideMoving())
   {
      // if move can result in a promotion //
      if (Move.PawnCanPromote())
         // generate all promotion moves //
         GeneratePromotionMoves();
      // move does not result in a promotion //
      else
         // push move onto move list //
         MoveList.push_back(Move);
   }
}

// analyze encounter with an enemy piece //
void ChessBoard::AnalyzeEnemy()
{
   // if piece moving is a pawn and captures a piece directly forward //
   if (Move.IsPawn() && (Direction == 0 || Direction == 8))
      // return that this move cannot be properly analyzed since it is illegal //
      return;

   // store the piece captured on this square //
   Move.StoreCapturedPiece(SquareIter->RetrievePieceOnSquare());

   // push the captured piece onto the list of encountered pieces //
   PathPieces.push_back(SquareIter->RetrievePieceOnSquare());

   // if have only only encountered one piece along this path //
   if (PathPieces.size() == 1)
   {
      // mark that move attacks on the square it moves to //
      SquareList.at(Position).AddAttackToSquare(Move.PieceColor());

      // if piece moving is a pawn //
      if (PieceIter->IsPawn())
         // mark that piece is attacked by a pawn //
         PieceList.at(PathPieces.at(0).PieceIndex()).MarkAttackedByPawn();

      // add move to the move list if it is legal //
      AddToMoveListIfLegal();

      // test whether an encounter with a king has occurred //
      KingEncounterTest();

      // directionally attack piece if possible //
      DirectionallyAttack();

      // if the piece is the moving side and a king was potentially captured //
      if (Move.SideMoving() && Move.KingIsCaptured())
         // set that a king was potentially captured which is illegal to have //
         KingCaptured = true;
   }
   // have encountered only two pieces on this path //
   else if (PathPieces.size() == 2)
      // mark a pin if occurs //
      PinPiece();
   // have encountered only three pieces on this path //
   else if (PathPieces.size() == 3)
      // mark a special pin if occurs //
      SpecialPinTest();

   // set that not done moving along this path yet //
   DoneMoving = false;
}

// test if have encountered a king //
void ChessBoard::KingEncounterTest()
{
   // if the piece encountered was an enemy king //
   if (PathPieces.front().IsKing() && (PathPieces.front().PieceColor() == Color))
   {
      // mark the piece moving as causing a check //
      PieceList.at(Move.PieceIndex()).CauseCheck();

      // set whether in a state of check or double check //
      SetCheck();

      // mark the path between the enemy king and the attacking piece //
      BlockPath();
   }
}

// directionally attack piece if possible to //
void ChessBoard::DirectionallyAttack()
{
   // test if a bishop, rook, or queen is moving //
   if (Move.IsBishop() || Move.IsRook() || Move.IsQueen())
      // directionally attack piece //
      PieceList.at(PathPieces.front().PieceIndex()).DirectionallyAttack(Direction);
}

// pin piece if possible to //
void ChessBoard::PinPiece()
{
   // test if can possibly pin piece //
   if (Move.PieceColor() != PathPieces.front().PieceColor() && PathPieces.
      at(1).IsKing())
      // mark the directions along which pinned piece can move //
      PieceList.at(PathPieces.front().PieceIndex()).DirectionallyPin(Direction);
}

// test for a special type of pin involving an en passant pawn //
void ChessBoard::SpecialPinTest()
{
   // if have a special type of pin involving an en passant pawn //
   if (PathPieces.front().IsPawn() && PathPieces.at(1).IsPawn() && PathPieces.at(2).
      IsKing() && PathPieces.at(2).SideMoving() && (PathPieces.front().IsEnpassant()
      || PathPieces.at(1).IsEnpassant()) && (Direction == 4 || Direction == 12))
      // mark that special pin has occurred //
      SpecialPin = true;
}

// set if in check or double check //
void ChessBoard::SetCheck()
{
   // if not previously in a state of check //
   if (NoCheck())
      // set that in check now //
      Check = 0;
   // if were previously in a state of check //
   else
      // set that in double check now //
      Check = 1;
}

// mark path on board between king and attacker //
void ChessBoard::BlockPath()
{
   // loop over all elements of block path //
   for (std::vector<int>::iterator PathIter = Path.begin(); PathIter != Path.end();
       PathIter++)
      // mark path on board between king and attacker //
      SquareList.at(*PathIter).MarkBlock();
}

// determine if can analyze the encounter of a friendly piece //
void ChessBoard::AnalyzeFriend()
{
   // if is a pawn moving directly forward //
   if (PieceIter->IsPawn() && (Direction == 0 || Direction == 8))
      // return that move can not be analyzed since it is illegal //
      return;

   // push piece onto list of encountered pieces //
   PathPieces.push_back(SquareIter->RetrievePieceOnSquare());

   // if this is the only encountered piece //
   if (PathPieces.size() == 1)
   {
      // mark that this move attacks square //
      SquareList.at(Position).AddAttackToSquare(Move.PieceColor());

      // if piece moving is a pawn //
      if (PieceIter->IsPawn())
         // mark that piece is guarded by a pawn //
         PieceList.at(PathPieces.at(0).PieceIndex()).MarkGuardedByPawn();

      // if a special pin is not possibly occurring //
      if (!(PathPieces.at(0).IsPawn() && ((PieceIter->IsQueen() || PieceIter->IsRook())
         && (Direction == 4 || Direction == 12))))
         // set that done analyzing along this path //
         return;
   }

   // set that not done moving along this path yet //
   DoneMoving = false;
}

// refresh the positions of the pieces on the board //
void ChessBoard::RefreshLayout()
{
   // loop over all the squares //
   for (SquareIter = SquareList.begin(); SquareIter != SquareList.end(); SquareIter++)
      // initialize square attributes //
      SquareIter->InitializeAttributes();

   // loop over all the pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // if piece is active //
      if (PieceIter->IsActive())
      {
         // store the occupying piece into the square //
         SquareList.at(PieceIter->PiecePosition()).StorePiece(*PieceIter);

         // if piece is eligible to be captured enpassant //
         if (PieceIter->IsEnpassant())
            // store the en passant occupying piece into the square //
            SquareList.at(PieceIter->EnpassantPiecePosition()).StoreEnpassantPiece
                                                                (*PieceIter);
      }
}

// get the identification label for this move //
int ChessBoard::MoveLabel()
{
   // return the identification label for this move //
   return MadeMove.MoveLabel();
}

// determine which color is currently moving on the board //
bool ChessBoard::ColorMoving()
{
   // return the color currently moving on the board //
   return Color;
}

// test if white is currently moving //
bool ChessBoard::IsWhite()
{
   // return whether white is currently moving //
   return Color;
}

// test if black is currently moving //
bool ChessBoard::IsBlack()
{
   // return whether black is currently moving //
   return !Color;
}

// get the score vector with the tree path in it //
std::vector<int> ChessBoard::GetScoreWithTreePath()
{
   // return the score vector with the tree path in it //
   return Score;
}

// calculate the score at this tree leaf //
std::vector<int> ChessBoard::EvaluateLeaf(int Depth)
{
   // if have a checkmate //
   if (IsCheckmate())
      // return the score for being mated modified by the tree depth //
      Score.at(0) = int(-1E5) + Depth - 1;
   // if have a stalemate or threefold repetition //
   else if (IsStalemate() || IsThreefoldRepetition())
      // assign a neutral score to stalemate or threefold repetition //
      Score.at(0) = 0;
   // if not a checkmate, stalemate, or threefold repetition //
   else
      // sum up components of leaf evaluation function //
      ComputeScore();

   // return the score vector with the tree path in it //
   return Score;
}

// find the rank and file of the king currently searching best move for //
void ChessBoard::FindKingFileAndRank(bool IsWhite)
{
   // if the original color engine is calculating for is white //
   if (IsWhite)
      // set index to be for the white king //
      Index = WhiteKingIndex;
   // original color engine is calculating for is black //
   else
      // set index to be for the black king //
      Index = BlackKingIndex;

   // get the file the king is on //
   KingFile = PieceList.at(Index).PieceFile();

   // get the rank the king is on //
   KingRank = PieceList.at(Index).PieceRank();

   // if white is color engine is searching for //
   if (IsWhite)
   {
      // set the first rank in front of the king //
      KingRankPlusOne = KingRank + 1;

      // set the second rank in front of the king //
      KingRankPlusTwo = KingRank + 2;
   }
   // black is the color engine is searching for //
   else
   {
      // set the first rank in front of the king //
      KingRankPlusOne = KingRank - 1;

      // set the second rank in front of the king //
      KingRankPlusTwo = KingRank - 2;
   }
}

// compute the safety of the king //
void ChessBoard::ComputeKingSafety()
{
   // find the king rank and file and adjacent ranks for the white king //
   FindKingFileAndRank(true);

   // compute attacks and guards in the king zone for white //
   ComputeAttacksAndGuardsInKingZone(true);

   // compute the king pawn shield for the castled white king //
   ComputeKingPawnShield(true);

   // find the king rank and file and adjacent ranks for the black king //
   FindKingFileAndRank(false);

   // compute attacks and guards in the king zone for black //
   ComputeAttacksAndGuardsInKingZone(false);

   // compute the king pawn shield for the castled black king //
   ComputeKingPawnShield(false);
}

// compute the attacks in the king zone //
void ChessBoard::ComputeAttacksAndGuardsInKingZone(bool IsWhite)
{
   // if there are less than ten pieces on the board //
   if (PiecesLeft < 10)
      // exit function since there are less than ten pieces on the board //
      return;

   // initialize attacks in the king zone to zero //
   AttacksInKingZonePoints = 0;

   // initialize guards in the king zone to zero //
   GuardsInKingZonePoints = 0;

   // initialize friendly pieces in the king zone to zero //
   FriendlyPiecesInKingZonePoints = 0;

   // initialize enemy pieces in the king zone to zero //
   EnemyPiecesInKingZonePoints = 0;

   // loop over files near the king //
   for (File = KingFile - 2; File <= KingFile + 2; File++)
      // loop over ranks near the king //
      for (Rank = KingRank - 3; Rank <= KingRank + 3; Rank++)
         // if the file or rank is not off the board //
         if (File >= 0 && File <= 7 && Rank >= 0 && Rank <= 7)
         {
            // set the current board position //
            Position = Rank * 8 + File;

            // increment the number of enemy attacks on the king zone //
            AttacksInKingZonePoints -= SquareList.at(Position).NumberOfAttacksByColor
                                       (!IsWhite);

            // increment the number of guards on the king zone by friendly pieces //
            GuardsInKingZonePoints += SquareList.at(Position).NumberOfAttacksByColor
                                      (IsWhite);

            // if there is a piece on this square //
            if (SquareList.at(Position).IsOccupied())
            {
               // if the piece on square is of color engine is searching for //
               if (SquareList.at(Position).PieceColor() == IsWhite)
                  // increment number of friendly pieces in the king zone //
                  FriendlyPiecesInKingZonePoints++;
               // piece on square is not of color engine is searching for //
               else
                  // increment number of enemy pieces in the king zone //
                  EnemyPiecesInKingZonePoints--;
            }
         }

   // if the color being searched for is black //
   if (!IsWhite)
   {
      // negate the attacks in king zone points //
      AttacksInKingZonePoints *= -1;

      // negate the guards in king zone points //
      GuardsInKingZonePoints *= -1;

      // negate the friendly pieces in king zone points //
      FriendlyPiecesInKingZonePoints *= -1;

      // negate the enemy pieces in king zone points //
      EnemyPiecesInKingZonePoints *= -1;
   }

   // sum up the total points for attacks in the king zone //
   TotalAttacksInKingZonePoints += AttacksInKingZonePoints;

   // sum up the total points for guards in the king zone //
   TotalGuardsInKingZonePoints += GuardsInKingZonePoints;

   // sum up the total points for friendly pieces in the king zone //
   TotalFriendlyPiecesInKingZonePoints += FriendlyPiecesInKingZonePoints;

   // sum up the total points for enemy pieces in the king zone //
   TotalEnemyPiecesInKingZonePoints += EnemyPiecesInKingZonePoints;
}

// compute the points for the pawn shield in front of king engine is searching for //
void ChessBoard::ComputeKingPawnShield(bool IsWhite)
{
   // if there are twenty or more pieces left on the board and have castled //
   if (PiecesLeft >= 20 && ((IsWhite && (WhiteCastledKingside ||
      WhiteCastledQueenside)) || (!IsWhite && (BlackCastledKingside ||
      BlackCastledQueenside))))
   {
      // initialize the king pawn shield points to zero //
      KingPawnShieldPoints = 0;

      // give points for the king having shielding pawns one rank in front of it //
      KingPawnShieldPoints += 4 * (PawnCountForFile(KingFile - 1, KingRankPlusOne,
                              KingRankPlusOne, IsWhite) + PawnCountForFile
                              (KingFile, KingRankPlusOne, KingRankPlusOne,
                              IsWhite) + PawnCountForFile(KingFile + 1,
                              KingRankPlusOne, KingRankPlusOne, IsWhite));

      // give points for the king having shielding pawns two ranks in front of it //
      KingPawnShieldPoints += PawnCountForFile(KingFile - 1, KingRankPlusTwo,
                              KingRankPlusTwo, IsWhite) + PawnCountForFile
                              (KingFile, KingRankPlusTwo, KingRankPlusTwo,
                              IsWhite) + PawnCountForFile(KingFile + 1,
                              KingRankPlusTwo, KingRankPlusTwo, IsWhite);

      // if original color engine is search for is black //
      if (!IsWhite)
         // negate the king pawn shield points //
         KingPawnShieldPoints *= -1;
   }

   // sum up the total king pawn shield points //
   TotalKingPawnShieldPoints += KingPawnShieldPoints;
}

// compute the points for having castled or forfeiting castling //
void ChessBoard::ComputeCastlingPoints()
{
   // compute the points for having castled or forfeiting castling //
   CastlingPoints = 50 * (WhiteCastledKingside || WhiteCastledQueenside) - 50 *
                    (BlackCastledKingside || BlackCastledQueenside) - 50 *
                    WhiteForfeitsCastle() + 50 * BlackForfeitsCastle();
}

// the number of pieces left on the board //
int ChessBoard::NumberOfPiecesLeft()
{
   // return the number of pieces left on the board //
   return PiecesLeft;
}

// count the number of pieces left on the board //
void ChessBoard::ComputeNumberOfPiecesLeft()
{
   // initializes pieces left on the board to zero //
   PiecesLeft = 0;

   // loop over all pieces on the board //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // if the piece is currently active //
      if (PieceIter->IsActive())
         // increment the number of pieces left on the board //
         PiecesLeft++;
}

// compute the individual components for the evaluation function //
void ChessBoard::ComputeEvaluationComponents()
{
   // initialize material points to zero //
   MaterialPoints = 0;

   // initialize points for pieces on outposts to zero //
   OutpostPoints = 0;

   // initialize points for total attacks in the king zone //
   TotalAttacksInKingZonePoints = 0;

   // initialize points for total guards in the king zone //
   TotalGuardsInKingZonePoints = 0;

   // initialize number of total enemy pieces in the king zone //
   TotalEnemyPiecesInKingZonePoints = 0;

   // initialize the number of total friendly pieces in the king zone //
   TotalFriendlyPiecesInKingZonePoints = 0;

   // initialize king mobility to zero //
   KingMobilityPoints = 0;

   // initialize king development to zero //
   KingDevelopmentPoints = 0;

   // initialize queen mobility to zero //
   QueenMobilityPoints = 0;

   // initialize queen development to zero //
   QueenDevelopmentPoints = 0;

   // initilaize rook mobility to zero //
   RookMobilityPoints = 0;

   // initialize rook development to zero //
   RookDevelopmentPoints = 0;

   // initialize knight mobility to zero //
   KnightMobilityPoints = 0;

   // initialize knight development to zero //
   KnightDevelopmentPoints = 0;

   // initialize bishop mobility to zero //
   BishopMobilityPoints = 0;

   // initialie bishop development to zero //
   BishopDevelopmentPoints = 0;

   // initialize pawn development to zero //
   PawnDevelopmentPoints = 0;

   // initialize isolated pawn points to zero //
   IsolatedPawnPoints = 0;

   // initialize passed pawn points to zero //
   PassedPawnPoints = 0;

   // initialize doubled pawn points to zero //
   DoubledPawnPoints = 0;

   // initialize tripled pawn points to zero //
   TripledPawnPoints = 0;

   // initialize the total pawn shield points in front of the king //
   TotalKingPawnShieldPoints = 0;

   // analyze the pawn structure on the board //
   AnalyzePawnStructure();

   // compute the points for the bishop and rook pairs //
   ComputeBishopRookPairPoints();

   // compute points for castling //
   ComputeCastlingPoints();

   // compute the safety of the king //
   ComputeKingSafety();

   // loop over all the pieces //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // if the piece is active //
      if (PieceIter->IsActive())
      {
         // increment material points //
         MaterialPoints += PieceIter->GetMaterialPoints();

         // increment outpost pieces points //
         OutpostPoints += PieceIter->GetOutpostPoints();

         // if piece is a king //
         if (PieceIter->IsKing())
         {
            // increment king mobility points //
            KingMobilityPoints += PieceIter->GetMobilityPoints(PiecesLeft);

            // increment king development points //
            KingDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);
         }
         // if piece is a queen //
         else if (PieceIter->IsQueen())
         {
            // increment queen mobility points //
            QueenMobilityPoints += PieceIter->GetMobilityPoints(PiecesLeft);

            // increment queen development points //
            QueenDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);
         }
         // if piece is a rook //
         else if (PieceIter->IsRook())
         {
            // increment rook mobility points //
            RookMobilityPoints += PieceIter->GetMobilityPoints(PiecesLeft);

            // increment rook development points //
            RookDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);
         }
         // if piece is a bishop //
         else if (PieceIter->IsBishop())
         {
            // increment bishop mobility points //
            BishopMobilityPoints += PieceIter->GetMobilityPoints(PiecesLeft);

            // increment bishop development points //
            BishopDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);
         }
         // if piece is a knight //
         else if (PieceIter->IsKnight())
         {
            // increment knight mobility points //
            KnightMobilityPoints += PieceIter->GetMobilityPoints(PiecesLeft);

            // increment knight development points //
            KnightDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);
         }
         // if piece is a pawn //
         else if (PieceIter->IsPawn())
         {
            // increment pawn development points //
            PawnDevelopmentPoints += PieceIter->GetDevelopedPoints(PiecesLeft);

            // increment isolated pawn points //
            IsolatedPawnPoints += PieceIter->GetIsolatedPawnPoints();

            // increment passed pawn points //
            PassedPawnPoints += PieceIter->GetPassedPawnPoints();

            // increment doubled pawn points //
            DoubledPawnPoints += PieceIter->GetDoubledPawnPoints();

            // increment tripled pan points //
            TripledPawnPoints += PieceIter->GetTripledPawnPoints();
         }
      }
}

// whether have insufficient material for a mate or not //
bool ChessBoard::InsufficientMaterialToMate()
{
   // return whether have insufficient material for a checkmate or not //
   return (!NumberOfWhiteRooks && !NumberOfWhiteQueens && !NumberOfWhitePawns
          && !NumberOfBlackRooks && !NumberOfBlackQueens && !NumberOfBlackPawns)
          && ((NumberOfWhiteKnights == 1 && !NumberOfBlackKnights &&
          !NumberOfWhiteBishops && !NumberOfBlackBishops) || (!NumberOfWhiteKnights &&
          NumberOfBlackKnights == 1 && !NumberOfWhiteBishops &&
          !NumberOfBlackBishops) || (!NumberOfWhiteKnights && !NumberOfBlackKnights
          && NumberOfWhiteBishops == 1 && !NumberOfBlackBishops) ||
          (!NumberOfWhiteKnights && !NumberOfBlackKnights && !NumberOfWhiteBishops &&
          NumberOfBlackBishops == 1) || (!NumberOfWhiteKnights &&
          !NumberOfBlackKnights && !NumberOfWhiteBishops && !NumberOfBlackBishops)
          || (!NumberOfWhiteKnights && !NumberOfBlackKnights &&
          !NumberOfWhiteBishopsOnWhite && !NumberOfBlackBishopsOnWhite)
          || (!NumberOfWhiteKnights && !NumberOfBlackKnights &&
          !NumberOfWhiteBishopsOnBlack && !NumberOfBlackBishopsOnBlack));
}

// compute the number of each piece type //
void ChessBoard::ComputeNumbersOfEachPieceType()
{
   // set the number of white bishops to zero //
   NumberOfWhiteBishops = 0;

   // set the number of black bishops to zero //
   NumberOfBlackBishops = 0;

   // set the number of white bishops on white squares to zero //
   NumberOfWhiteBishopsOnWhite = 0;

   // set the number of black bishops on white squares to zero //
   NumberOfBlackBishopsOnWhite = 0;

   // set the number of white bishops on black squares to zero //
   NumberOfWhiteBishopsOnBlack = 0;

   // set the number of black bishops on black squares to zero //
   NumberOfBlackBishopsOnBlack = 0;

   // set the number of white rooks to zero //
   NumberOfWhiteRooks = 0;

   // set the number of black rooks to zero //
   NumberOfBlackRooks = 0;

   // set the number of white knights to zero //
   NumberOfWhiteKnights = 0;

   // set the number of black knights to zero //
   NumberOfBlackKnights = 0;

   // set the number of white queens to zero //
   NumberOfWhiteQueens = 0;

   // set the number of black queens to zero //
   NumberOfBlackQueens = 0;

   // set the number of white pawns to zero //
   NumberOfWhitePawns = 0;

   // set the number of black pawns to zero //
   NumberOfBlackPawns = 0;

   // loop over all the pieces on the board //
   for (PieceIter = PieceList.begin(); PieceIter != PieceList.end(); PieceIter++)
      // if the piece is active //
      if (PieceIter->IsActive())
      {
         // if the piece is white //
         if (PieceIter->IsWhite())
         {
            // if the piece is a pawn //
            if (PieceIter->IsPawn())
               // increment the number of white pawns //
               NumberOfWhitePawns++;
            // if the piece is a knight //
            else if (PieceIter->IsKnight())
               // increment the number of white knights //
               NumberOfWhiteKnights++;
            // if the piece is a bishop //
            else if (PieceIter->IsBishop())
            {
               // increment the number of white bishops //
               NumberOfWhiteBishops++;

               // if bishop is on a white square //
               if (PieceIter->OnWhiteSquare())
                  // increment the number of white bishops on white squares //
                  NumberOfWhiteBishopsOnWhite++;
               // bishop is on a black square //
               else
                   // increment the number of white bishops on black squares //
                  NumberOfWhiteBishopsOnBlack++;
            }
            // if the piece is a rook //
            else if (PieceIter->IsRook())
               // increment the number of white rooks //
               NumberOfWhiteRooks++;
            // if the piece is a queen //
            else if (PieceIter->IsQueen())
               // increment the number of white queens //
               NumberOfWhiteQueens++;
         }
         // if the piece is black //
         else if (PieceIter->IsBlack())
         {
            // if the piece is a pawn //
            if (PieceIter->IsPawn())
               // increment the number of black pawns //
               NumberOfBlackPawns++;
            // if the piece is a knight //
            else if (PieceIter->IsKnight())
               // increment the number of black knights //
               NumberOfBlackKnights++;
            // if the piece is a bishop //
            else if (PieceIter->IsBishop())
            {
               // increment the number of black bishops //
               NumberOfBlackBishops++;

               // if bishop is on a white square //
               if (PieceIter->OnWhiteSquare())
                  // increment number of black bishops on white squares //
                  NumberOfBlackBishopsOnWhite++;
               // bishop is on a black square //
               else
                  // increment number of black bishops on black squares //
                  NumberOfBlackBishopsOnBlack++;
            }
            // if the piece is a rook //
            else if (PieceIter->IsRook())
               // increment the number of black rooks //
               NumberOfBlackRooks++;
            // if the piece is a queen //
            else if (PieceIter->IsQueen())
               // increment the number of black queens //
               NumberOfBlackQueens++;
         }
      }
}

// the points for the bishop and the rook pairs //
void ChessBoard::ComputeBishopRookPairPoints()
{
   // return points for having the bishop pair and the rook pair //
   BishopRookPairPoints = 100 * (NumberOfWhiteBishops >= 2) - 100 *
                          (NumberOfBlackBishops >= 2) + 50 * (NumberOfWhiteRooks >= 2)
                          - 50 * (NumberOfBlackRooks >= 2) + 100 * (NumberOfBlackRooks
                          == 0 && NumberOfWhiteRooks > 0) - 100 * (NumberOfWhiteRooks
                          == 0 && NumberOfBlackRooks > 0);
}

// compute the score for the given board position //
void ChessBoard::ComputeScore()
{
   // compute the individual components of the evaluation function //
   ComputeEvaluationComponents();

   // add together the scaled individual evaluation function components //
   Score.at(0) = 1.0 * MaterialPoints +
                 0.2 * BishopRookPairPoints +
                 0.5 * KingMobilityPoints +
                 0.5 * QueenMobilityPoints +
                 1.5 * RookMobilityPoints +
                 2.0 * BishopMobilityPoints +
                 2.0 * KnightMobilityPoints +
                 0.5 * KingDevelopmentPoints +
                 1.0 * QueenDevelopmentPoints +
                 1.0 * RookDevelopmentPoints +
                 5.0 * BishopDevelopmentPoints +
                 5.0 * KnightDevelopmentPoints +
                 3.0 * PawnDevelopmentPoints +
                 0.3 * CastlingPoints +
                 2.0 * TotalAttacksInKingZonePoints +
                 2.0 * TotalGuardsInKingZonePoints +
                 5.0 * TotalEnemyPiecesInKingZonePoints +
                 5.0 * TotalFriendlyPiecesInKingZonePoints +
                 5.0 * TotalKingPawnShieldPoints +
                 10.0 * IsolatedPawnPoints +
                 10.0 * PassedPawnPoints +
                 10.0 * DoubledPawnPoints +
                 10.0 * TripledPawnPoints +
                 15.0 * OutpostPoints;

   // if black is moving //
   if (IsBlack())
      // negate the score so it can be seen from black's perspective //
      Score.at(0) *= -1;
}

// get the index of the currently moving king in the piece list //
int ChessBoard::MovingKingIndex()
{
   // if white is currently moving //
   if (IsWhite())
      // return index position of white king in piece list //
      return WhiteKingIndex;
   // if black is currently moving //
   else
      // return index position of black king in piece list //
      return BlackKingIndex;
}

// record a move on the board //
void ChessBoard::RecordMove()
{
   // update the piece that moved //
   PieceList.at(MadeMove.PieceIndex()) = MadeMove.RetrieveUpdatedPiece();

   // if was a capturing move //
   if (MadeMove.IsCapture())
      // deactivate the piece that was captured //
      PieceList.at(MadeMove.CapturedPieceIndex()).DeactivatePiece();

   // if move was a kingside castle //
   if (MadeMove.IsKingsideCastle())
   {
      // if white moved //
      if (MadeMove.IsWhite())
         // mark as a kingside castle for white //
         WhiteCastledKingside = true;
      // black moved //
      else
         // mark as a kingside castle for black //
         BlackCastledKingside = true;

      // increment the king's file position by two //
      PieceList.at(MovingKingIndex()).ShiftFile(2);
   }
   // if move was a queenside castle //
   else if (MadeMove.IsQueensideCastle())
   {
      // if white moved //
      if (MadeMove.IsWhite())
         // mark as a queenside castle for white //
         WhiteCastledQueenside = true;
      // black moved //
      else
         // mark as a queenside castle for black //
         BlackCastledQueenside = true;

      // decrement the king's file position by two //
      PieceList.at(MovingKingIndex()).ShiftFile(-2);
   }
}

// move a piece on the board //
void ChessBoard::MakeMove(int Depth)
{
   // store the move being made //
   MadeMove = Move;

   // record a move //
   RecordMove();

   // switch the color moving //
   Color = !Color;

   // generate all the moves that the current color can make //
   GenerateMoves();

   // determine if need to mark move as causing check, checkmate, or stalemate //
   MadeMoveCheckTest();

   // if at top level of board list //
   if (Depth == 0)
      // resolve pgn labels //
      PGNResolve();
   // not at root of game tree //
   else
      // add the move to the tree path portion of the score vector //
      Score.push_back(MadeMove.MoveLabel());

   // update the zobrist hash counts used in determining threefold repetition //
   UpdateZobristHashCounts();

   // compute the numbers of each piece type on the board //
   ComputeNumbersOfEachPieceType();

   // compute number of pieces left on the board //
   ComputeNumberOfPiecesLeft();
}

// update the zobrist hash counts used in determining threefold repetition //
void ChessBoard::UpdateZobristHashCounts()
{
   // set that have not previously found a zobrist hash //
   FoundPreviousZobristHash = false;

   // compute the current zobrist hash //
   ComputeZobristHash();

   // loop over the zobrist hash list of previous hash values //
   for (Index = 0; Index < ZobristHashes.size(); Index++)
      // if found a zobrist hash matching that of the current value //
      if (ZobristHashes.at(Index) == ZobristHash)
      {
         // increment the zobrist hash count for this hash value by one //
         ZobristHashCounts.at(Index)++;

         // set that found a previous zobrist hash value //
         FoundPreviousZobristHash = true;
      }

   // if didn't find a previous zobrist hash value //
   if (!FoundPreviousZobristHash)
   {
      // add the zobrist hash to the zobrist hashes vector //
      ZobristHashes.push_back(ZobristHash);

      // set the zobrist hash count for this hash value to one //
      ZobristHashCounts.push_back(1);
   }
}

// determine whether have threefold repetition or not //
bool ChessBoard::IsThreefoldRepetition()
{
   // loop over the entire zobrist hashes vector //
   for (Index = 0; Index < ZobristHashes.size(); Index++)
      // if the current zobrist hash has occurred three times //
      if (ZobristHashCounts.at(Index) == 3 && (ZobristHashes.at(Index) == ZobristHash))
         // return that have a case of threefold repetition which is a possible draw //
         return true;

   // return that did not have threefold repetition //
   return false;
}

// determine if need to mark move as causing check, checkmate, or stalemate //
void ChessBoard::MadeMoveCheckTest()
{
   // if have a state of check or double check //
```

```cpp
   if (IsInCheck())
      // mark that move causes a check //
      MadeMove.CauseCheck();
   // if have a state of checkmate //
   else if (IsCheckmate())
      // mark that move causes a checkmate //
      MadeMove.CauseCheckmate();
   // if have a state of stalemate //
   else if (IsStalemate())
      // mark that move causes a stalemate //
      MadeMove.CauseStalemate();
}

// find the move corresponding move label on the board //
bool ChessBoard::FindMove(int LabelToFind)
{
   // loop over move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if label matches a move label //
      if (LabelToFind == MoveIter->MoveLabel())
      {
         // assign the correct move //
         Move = *MoveIter;

         // return that move exists //
         return true;
      }

   // return that didn't find the move on the board //
   return false;
}

// find the corresponding move for the given move label //
bool ChessBoard::FindMove(std::string Label)
{
   // if pgn label is empty //
   if (Label.empty())
      // return that cannot find a move for empty pgn label //
      return false;

   // if pgn label has extraneous notation on the end of it //
   if (Label.at(Label.length() - 1) == '+' || Label.at(Label.length() - 1) == '#')
      // remove extraneous notation from pgn label //
      Label.resize(Label.length() - 1);

   // loop over move list //
   for (MoveIter = MoveList.begin(); MoveIter != MoveList.end(); MoveIter++)
      // if label matches a move label //
      if (Label == MoveIter->PGNLabel() || Label == MoveIter->UCILabel())
      {
         // assign the correct move //
         Move = *MoveIter;

         // return that move exists //
         return true;
      }

   // return that move does not exist //
   return false;
}

// determine if checkmate, stalemate, threefold repetition, or insufficient material //
bool ChessBoard::GameOver()
{
   // return if checkmate, stalemate, threefold repetition, or insufficient material //
   return IsCheckmate() || IsStalemate() || IsThreefoldRepetition() ||
          InsufficientMaterialToMate();;
}

// determine if there are no moves left in the move list //
bool ChessBoard::NoMoves()
{
   // return whether there are no moves left in the move list //
   return MoveList.empty();
}

// determine if there is only one possible move on the board //
bool ChessBoard::OnlyOnePossibleMove()
{
   // return whether there is only one possible move on the board //
   return MoveList.size() == 1;
}

// send information about the engine search to standard output //
void ChessEngine::InfoToStandardOutput(int Plies, long Count, int Duration,
                                       std::vector<int> Score, ChessBoard Board,
                                       int TransCount, int TransUsed)
{
   // if duration in seconds is zero //
   if (Duration == 0)
      // set the search rate to the total number of nodes searched //
      SearchRate = Count;
   // duration of search in seconds is greater than zero //
   else
      // set the search rate to number of nodes searched per second //
      SearchRate = Count / Duration;

   // send the search information to standard output //
   std::cout << "Depth=" << Plies << " Nodes=" << Count / 1000 << "K NPS="
             << SearchRate / 1000 << "K Time=" << Duration << "S Trans="
             << TransCount / 1000 << "K Used=" << TransUsed / 1000 << "K Score="
             << Score.at(0) << " Line=";

   // loop over the moves in the move list portion of the score vector //
   for (Index = 1; Index < Score.size(); Index++)
   {
      // if found the move on the board //
      if (Board.FindMove(Score.at(Index)))
      {
         // make the move on the board //
         Board.MakeMove(0);

         // print the move to standard output //
         std::cout << Board.PGNLabel() << " ";
      }
   }

   // print a newline //
   std::cout << "\n";
}

// make the chess engine stop thinking if sent stop command from the gui //
void ChessEngine::MakeEngineStop()
{
   // set that the gui wants the engine to stop thinking //
   StopEngine = true;
}

// whether can use a score from the transposition table or not //
bool ChessEngine::CanUseTranspositionTableScore(int Depth, int Beta, std::vector
                                                <ChessBoard>::iterator BoardIter)
{
   // set the best move for sorting to a move label of - 1 //
   BestMoveForSorting = -1;

   // if the zobrist hash matches a stored board position //
   if (TransTable.BoardPositionWasPreviouslyStored(BoardIter->GetZobristHash()))
   {
      // get the best move for sorting that was stored in the transposition table //
      BestMoveForSorting = TransTable.GetBestMove();

      // if data in transposition table was searched to the needed depth // //
      if (TransTable.DataSearchedDeepEnough(Plies, Depth))
      {
         // increment the number of transposition table elements used //
         TransUsed++;

         // return if an exact score value stored or if stored value makes a cutoff //
         return TransTable.NotCutoff() || (TransTable.WasCutoff() &&
                TransTable.GetScore() >= Beta);
      }
   }

   // return that can't use a score from the transposition table //
   return false;
}

// whether need to stop searching the game tree or not //
bool ChessEngine::NeedToStopSearching()
{
   // return whether need to stop searching the game tree or not //
   return ((difftime(time(NULL), Start) > MaxTime && TimeLimited) || StopEngine) &&
          (Plies > 1);
}

// negate the score vector from alpha-beta search of the game tree //
std::vector<int> ChessEngine::NegateScore(std::vector<int> ScoreToNegate)
{
   // negate the score vector from alpha-beta search of the game tree //
   ScoreToNegate.at(0) *= -1;

   // return the negated score vector from alpha-beta search of the game tree //
   return ScoreToNegate;
}

// recursive alpha-beta search algorithm //
std::vector<int> ChessEngine::AlphaBetaSearch(std::vector<int> Alpha, std::vector<int>
                                              Beta, std::vector<ChessBoard>::iterator
                                              BoardIter, int Depth)
{
   // the best move calculated so far to save //
   int BestMoveToSave = -1;

   // set that alpha has not been updated yet //
   bool UpdatedAlpha = false;

   // stop alpha-beta search if reached search time limit or need to stop //
   if (NeedToStopSearching())
      // return alpha value //
      return Alpha;

   // if have threefold repetition //
   if (BoardIter->IsThreefoldRepetition())
      // return the score for this leaf node //
      return BoardIter->EvaluateLeaf(Depth);

   // if can use a score from the transposition table //
   if (CanUseTranspositionTableScore(Depth, Beta.at(0), BoardIter))
   {
      // get the score vector with the game tree path in it //
      Score = BoardIter->GetScoreWithTreePath();

      // return a score from the transposition table //
      Score.at(0) = TransTable.GetScore();

      // return the score vector //
      return Score;
   }

   // if in the quiescence portion of the game tree //
   if (Depth >= Plies)
      // condense move list to only unstable moves for quiescence calculations //
      BoardIter->CondenseMoves();

   // if no moves on board or have reached maximum quiescence depth to search to //
   if (BoardIter->NoMoves() || (Depth >= (Plies / 2) * 2 + 8) || (Plies >= 10 &&
      Depth >= Plies) || (BoardIter->NumberOfPiecesLeft() <= 6 && Depth >=
      (Plies / 2) * 2 + 2))
      // return the score for this leaf node //
      return BoardIter->EvaluateLeaf(Depth);

   // if the current depth is greater than main search depth and not in check //
   if (Depth >= Plies && BoardIter->NotInCheck())
   {
      // get the score from the current board position //
      Score = BoardIter->EvaluateLeaf(Depth);

      // if score is better than alpha //
      if (Score.at(0) > Alpha.at(0))
         // update alpha to the better score //
         Alpha = Score;

      // if the score is greater than beta //
      if (Score.at(0) >= Beta.at(0))
         // return the score //
         return Score;
   }

   // sort all the movess in the move list to optimize alpha-beta search of tree //
   BoardIter->SortMoves(BestMoveForSorting, KillerMoves.at(Depth));

   // while there is another move left to simulate //
   while (BoardIter->RetrieveNextMove())
   {
      // copy current board into next level down in search tree //
      *(BoardIter + 1) = *BoardIter;

      // make a move at next level down in search tree //
      (BoardIter + 1)->MakeMove(Depth + 1);

      // increment number of moves simulated //
      Count++;

      // get score from alpha-beta search and negate and swap alpha and beta //
      Score = NegateScore(AlphaBetaSearch(NegateScore(Beta), NegateScore(Alpha),
                          BoardIter + 1, Depth + 1));

      // update alpha value //
      if (Score.at(0) > Alpha.at(0))
      {
         // set the best move to save to the current move label in the board //
         BestMoveToSave = BoardIter->MoveLabel();

         // update alpha with better score //
         Alpha = Score;

         // set that have updated alpha //
         UpdatedAlpha = true;
      }

      // if have an alpha-beta cutoff //
      if (Alpha.at(0) >= Beta.at(0))
      {
         // update the killer moves with the best move //
         UpdateKillerMoves(Depth, BoardIter->MoveLabel());

         // exit loop since already found an alpha-beta cutoff and done searching //
         break;
      }
   }

   // if updated alpha and score is not possibly from a threefold repetition //
   if (UpdatedAlpha && (Alpha.at(0) != 0) && (abs(Alpha.at(0)) < 9E4) &&
      !NeedToStopSearching() && (Plies > Depth))
      // proccess the score to see if will save it in transposition table or not //
      TransTable.ProcessTransData(BoardIter->GetZobristHash(), Plies - Depth,
                                  Alpha.at(0), Alpha.at(0) >= Beta.at(0),
                                  BestMoveToSave);

   // return the alpha value //
   return Alpha;
}

// compact the transposition table //
void ChessEngine::CompactTransTable(bool AgeForward)
{
   // compact the transposition table //
   TransTable.CompactTransTable(AgeForward);
}

// initialize attributes for chess engine //
void ChessEngine::InitializeAttributes(ChessBoard Board, int TimeLimit, int PlyLimit)
{
   // compact the transposition table //
   CompactTransTable(true);

   // make room for killer move lists done to a depth of two hundred //
   KillerMoves.resize(200);

   // loop over all the killer moves //
   for (Index = 0; Index < KillerMoves.size(); Index++)
      // make room for five killer moves per depth in game tree //
      KillerMoves.at(Index).resize(5);

   // loop over the killer move list //
   for (Index = 0; Index < KillerMoves.size(); Index++)
      // loop over the killer moves per depth in the killer moves list //
      for (Index2 = 0; Index2 < KillerMoves.at(Index).size(); Index2++)
         // initialize the killer move label to minus one //
         KillerMoves.at(Index).at(Index2) = -1;

   // set that initially gui has not told the chess engine to stop thinking //
   StopEngine = false;

   // assign maximum time limit for computational engine search //
   MaxTime = TimeLimit;

   // determine whether engine search is time limited or not //
   TimeLimited = TimeLimit;

   // assign maximum ply limit for computational engine search //
   MaxPlies = PlyLimit;

   // determine whether engine search is ply limited or not //
   PlyLimited = PlyLimit;

   // make room for two hundered boards in board list //
   BoardList.resize(200);

   // assign root board to root of game search tree //
   BoardList.at(0) = Board;

   // set start time of search //
   Start = time(NULL);

   // initialize number of moves simulated to zero //
   Count = 0;

   // initialize the number of transposition table elements used to zero //
   TransUsed = 0;

   // resize the alpha vector for alpha-beta search to one //
   Alpha.resize(1);

   // resize the beta vector for alpha-beta search to one //
   Beta.resize(1);

   // initialize the score portion of the alpha vector for alpha-beta search //
   Alpha.at(0) = -1E6;

   // initialize the score portion of the alpha vector for alpha-beta search //
   Beta.at(0) = 1E6;

   // set the initial ply depth to one //
   Plies = 1;
}

// convert best move to uci notation //
void ChessEngine::ConvertBestMove(ChessBoard Board)
{
   // find the first move along best line of play //
   Board.FindMove(Score.at(1));

   // make the move along best line of play //
   Board.MakeMove(0);

   // get uci label for the best move //
   BestMove = Board.UCILabel();
}

// update the killer moves //
void ChessEngine::UpdateKillerMoves(int Depth, int MoveLabel)
{
   // loop over the killer moves for the given depth //
   for (Index = KillerMoves.at(Depth).size() - 1; Index > 0; Index--)
      // move the killer moves back one spot to make room for the newest killer move //
      KillerMoves.at(Depth).at(Index) = KillerMoves.at(Depth).at(Index - 1);

   // save the newest killer move in the first spot //
   KillerMoves.at(Depth).at(0) = MoveLabel;
}

// initialize the transposition table //
void ChessEngine::InitializeTransTable()
{
   // initialize the transposition table //
   TransTable.InitializeAttributes();
}

// calculate theoretical best move in the game tree //
std::string ChessEngine::FindBestMove(ChessBoard Board, int TimeLimit, int PlyLimit)
{
   // initialize attributes for chess engine //
   InitializeAttributes(Board, TimeLimit, PlyLimit);

   // do until reached the closest checkmate in the game tree //
   do
   {
      // extract the score and best move from the game tree //
      Score = AlphaBetaSearch(Alpha, Beta, BoardList.begin(), 0);

      // if there is only one possible move that could be made //
      if (Board.OnlyOnePossibleMove())
      {
         // convert the only move to a uci label and exit early //
         ConvertBestMove(Board);

         // exit the loop since don't need to search anymore since only one move //
         break;
      }

      // if have reached the time limit passed into the chess engine or need to stop //
      if (NeedToStopSearching())
         // break out of loop //
         break;

      // update the killer moves with the newest killer move //
      UpdateKillerMoves(0, Score.at(1));

      // convert best line of play in game tree to uci notation //
      ConvertBestMove(Board);

      // send search information to standard output //
      InfoToStandardOutput(Plies, Count, int(difftime(time(NULL), Start)), Score,
                           Board, TransTable.TableSize(), TransUsed);

      // if search is ply limited and have searched to maximum ply depth //
      if (PlyLimited && MaxPlies == Plies)
         // break out of loop if reached ply search limit //
         break;

      // increment the search depth by one //
      Plies++;
   // do while have not found checkmate or not ending up being mated //
   } while (abs(Score.at(0)) < 9E4);

   // return pgn label of best theoretical move from the chess engine //
   return BestMove;
}

// store move from pgn database into the opening book //
void ChessOpeningBookElement::StoreOpeningBookData(unsigned long long ZobristHashValue,
                                                   std::string NextMoveValue)
{
   // save the zobrist hash value for the given board position //
   ZobristHash = ZobristHashValue;

   // save the move for the next move to be made from the given board position //
   NextMove = NextMoveValue;
}

// get the next move for the given board position in the opening book //
std::string ChessOpeningBookElement::GetNextMove()
{
   // return the next move for the given board position in the opening book //
   return NextMove;
}

// get the zobrist hash for this specific chess opening book element //
unsigned long long ChessOpeningBookElement::GetZobristHash()
{
   // return the zobrist hash for this specific chess opening book element //
   return ZobristHash;
}

// get the next move for the specific board position from the opening book //
std::string ChessOpeningBook::GetNextMove()
{
   // seed the random number generator with the current time //
   srand(time(NULL));

   // call the random number generator once to randomize it //
   rand();

   // retun one of the next moves for the given board position //
   return OpeningBookElements.at(rand() % OpeningBookElements.size()).GetNextMove();
}

// initialize the opening book attributes //
void ChessOpeningBook::InitializeAttributes()
{
   // resize the chess opening book to be one hundred thousand elements //
   OpeningBook.resize(1E5);

   // loop over the entire opening book //
   for (BookIter = OpeningBook.begin(); BookIter != OpeningBook.end(); BookIter++)
      // clear each list of opening book elements at this place in the opening book //
      BookIter->clear();
}

// determine whether a move exists in the opening book //
bool ChessOpeningBook::MoveExists(unsigned long long ZobristHash)
{
   // get the index in the opening book from the zobrist hash value //
   Index = ZobristHash % OpeningBook.size();

   // clear a list of opening book elements //
   OpeningBookElements.clear();

   // if there are possible moves at this location in the opening book //
   if (!OpeningBook.at(Index).empty())
      // loop over the moves stored at this location in the opening book //
      for (ElementIter = OpeningBook.at(Index).begin(); ElementIter != OpeningBook.
          at(Index).end(); ElementIter++)
         // if the zobrist hash value matches the stored value //
         if (ElementIter->GetZobristHash() == ZobristHash)
            // add an element to the opening book element list //
            OpeningBookElements.push_back(*ElementIter);

   // return whether found possible moves at this location in the opening book //
   return !OpeningBookElements.empty();
}

// store data into the opening book //
void ChessOpeningBook::StoreOpeningBookData(unsigned long long ZobristHash, std::string
                                            NextMove)
{
   // set that don't already have next move value stored in the opening book //
   DataAlreadyExists = false;

   // get the corresponding index in the chess opening book //
   Index = ZobristHash % OpeningBook.size();

   // store the zobrist hash and the next move into an opening book element //
   OpeningBookElement.StoreOpeningBookData(ZobristHash, NextMove);

   // loop over all the elements at this location in the opening book //
   for (ElementIter = OpeningBook.at(Index).begin(); ElementIter != OpeningBook.at
       (Index).end(); ElementIter++)
      // if already have the same exact value stored in the opening book //
      if (ElementIter->GetZobristHash() == ZobristHash && ElementIter->GetNextMove()
         == NextMove)
      {
         // set that the next move data already exists in the opening book //
         DataAlreadyExists = true;

         // exit the loop since already have the next move data in the opening book //
         break;
      }

   // if the data doesn't already exist in the opening book //
   if (!DataAlreadyExists)
      // add the opening book element into the opening book //
      OpeningBook.at(Index).push_back(OpeningBookElement);
}

// compact the transposition table //
void ChessTranspositionTable::CompactTransTable(bool AgeForward)
{
   // if aging forward //
   if (AgeForward)
      // age the transposition table forward by one //
      AgeTable(1);
   // if aging backwards //
   else
      // age the transposition table backwards by one //
      AgeTable(-1);

   // remove old elements in the transposition table //
   RemoveOldElements();
}

// initialize the transposition table attributes //
void ChessTranspositionTable::InitializeAttributes()
{
   // if the transposition table size is already one million elements //
   if (Table.size() == 1E6)
      // loop over the entire transposition table //
      for (TableIter = Table.begin(); TableIter != Table.end(); TableIter++)
         // clear the list of elements at this location in the transposition table //
         TableIter->clear();
   // transposition table has not been sized yet //
   else
      // set the transposition table to hold one million elements //
      Table.resize(1E6);
}

// return whether data in the transposition table was searched to the required depth //
bool ChessTranspositionTable::DataSearchedDeepEnough(int Plies, int Depth)
{
   // if the search depth is less than two or in the quiescence portion //
   if (Depth < 2 || (Depth >= Plies))
      // exit since don't want to use score at root or in quiescence portion of tree //
      return false;

   // return whether data in the transposition table was search to required depth //
   return ElementIter->GetDepth() >= Plies - Depth;
}

// get whether there is a stored score for the corresponding zobrist hash or not //
bool ChessTranspositionTable::BoardPositionWasPreviouslyStored(unsigned long long
                                                               ZobristHash)
{
   // get the index in the transposition table to store the score at //
   Index = ZobristHash % Table.size();

   // initially set that haven't found any stored data yet //
   FoundStoredData = false;

   // if there is a list of data stored at this hash table index //
   if (Table.at(Index).size() > 0)
      // loop over all the transposition table elements at this index //
      for (ElementIter = Table.at(Index).begin(); ElementIter != Table.at(Index).end();
          ElementIter++)
         // if there is a match to the zobrist hash and search depth is deep enough //
         if (ElementIter->GetZobristHash() == ZobristHash)
         {
            // set that found stored data at this hash table index data list //
            FoundStoredData = true;

            // exit loop since already found stored data //
            break;
         }

   // return that there is stored data matching the given zobrist hash value //
   return FoundStoredData;
}

// get the best move that was searched from the transposition table element //
int ChessTranspositionTable::GetBestMove()
{
   // return the best move that was searched from the tranposition table element //
   return ElementIter->GetBestMove();
}

// get the score for the given transposition table element //
int ChessTranspositionTable::GetScore()
{
   // return the score for the given transposition table element //
   return ElementIter->GetScore();
}

// whether the score for the given transposition table element was from a cutoff //
bool ChessTranspositionTable::WasCutoff()
{
   // return whether the score for the given transposition table element was cutoff //
   return ElementIter->WasCutoff();
}

// whether the score for the given transposition table element not from a cutoff //
bool ChessTranspositionTable::NotCutoff()
{
   // return whether the score for the given transposition table element not cutoff //
   return !ElementIter->WasCutoff();
}

// remove old elements in the transposition table //
void ChessTranspositionTable::RemoveOldElements()
{
   // loop over the entire transposition table //
   for (TableIter = Table.begin(); TableIter != Table.end(); TableIter++)
   {
      // if the vector of elements at this part of the trans table is not empty //
      if (!TableIter->empty())
      {
         // get the list of elements at this location in the transposition table //
         ElementList = *TableIter;

         // clear the list of elements at this location in the transposition table //
         TableIter->clear();

         // loop over all the elements in the element list //
         for (ElementIter = ElementList.begin(); ElementIter != ElementList.end();
             ElementIter++)
            // if the transposition table element age is from 1 to 25 //
            if (ElementIter->GetAge() >= 1 && ElementIter->GetAge() <= 25)
               // add the transposition element back to the transposition table //
               TableIter->push_back(*ElementIter);
      }
   }
}

// the size of the transposition table //
int ChessTranspositionTable::TableSize()
{
   // initialize transposition table size to zero //
   Count = 0;

   // loop over the entire tranposition table //
   for (TableIter = Table.begin(); TableIter != Table.end(); TableIter++)
      // if the vector of elements at this location in the trans table is not empty //
      if (!TableIter->empty())
         // add the vector count to the overall size of the tranposition table //
         Count += TableIter->size();

   // return the size of the transposition table //
   return Count;
}

// age the tranposition table //
void ChessTranspositionTable::AgeTable(int AgeChange)
{
   // loop over the entire transposition table //
   for (TableIter = Table.begin(); TableIter != Table.end(); TableIter++)
      // loop over the vector of elements at this location in the trans table //
      for (ElementIter = TableIter->begin(); ElementIter != TableIter->end();
          ElementIter++)
         // age the transposition table element //
         ElementIter->AgeElement(AgeChange);
}

// process the transposition table data to see if will store score or not //
void ChessTranspositionTable::ProcessTransData(unsigned long long ZobristHash, int
                                               Depth, int Score, bool CutoffOccurred,
                                               int BestMove)
{
   // store the transposition data to a table element //
   TableElement.StoreTransData(ZobristHash, Depth, Score, CutoffOccurred, BestMove);

   // convert the zobrist hash value to an index within range of table size //
   Index = ZobristHash % Table.size();

   // if the element list at the table index is empty //
   if (Table.at(Index).empty())
      // push the table element onto the transposition table //
      Table.at(Index).push_back(TableElement);
   // the element list at the table index is not empty //
   else
   {
      // set that initially have not found any stored data //
      FoundStoredData = false;

      // loop over the list of data at for this index in the hash table //
      for (ElementIter = Table.at(Index).begin(); ElementIter != Table.at(Index).end();
          ElementIter++)
         // if element in the transposition tables matches the given zobrist hash //
         if (ElementIter->GetZobristHash() == ZobristHash)
         {
            // set that have found previous data stored in the transposition table //
            FoundStoredData = true;

            // if score computed is of greater or equal depth than previous score //
            if (Depth >= ElementIter->GetDepth())
               // copy over the transposition table element into trans table //
               *ElementIter = TableElement;

            // exit loop since already stored data into the transposition table //
            break;
         }

      // if didn't find any previously stored data in the transposition table //
      if (!FoundStoredData)
         // add transposition table element into the transposition table //
         Table.at(Index).push_back(TableElement);
   }
}

// age the transposition table element //
void ChessTranspositionTableElement::AgeElement(int AgeChange)
{
   // age the transposition table element //
   Age += AgeChange;
}

// get the age of the transposition table element //
int ChessTranspositionTableElement::GetAge()
{
   // return the age of the transposition table element //
   return Age;
}

// store the transposition data into a transposition table element //
void ChessTranspositionTableElement::StoreTransData(unsigned long long
                                                    ZobristHashValue, int DepthValue,
                                                    int ScoreValue, bool
                                                    CutoffOccurredValue, int
                                                    BestMoveToSave)
{
   // set the age of the transposition table element initially to 1 //
   Age = 1;

   // save the best move to make for this particular board position //
   BestMove = BestMoveToSave;

   // store whether score is from an alpha-beta cutoff or not //
   CutoffOccurred = CutoffOccurredValue;

   // store the score in the transposition table element //
   Score = ScoreValue;

   // store the zobrist hash value representing the board position //
   ZobristHash = ZobristHashValue;

   // store the depth to which this board position was searched //
   Depth = DepthValue;
}

// get the zobrist hash value for the given board position //
unsigned long long ChessTranspositionTableElement::GetZobristHash()
{
   // return the zobrist hash value for the given board position //
   return ZobristHash;
}

// get the depth to which the board position was searched //
int ChessTranspositionTableElement::GetDepth()
{
   // return the depth to which the board position was searched //
   return Depth;
}

// get the best move searched from the transposition table element //
int ChessTranspositionTableElement::GetBestMove()
{
   // return the best move searched from the transposition table element //
   return BestMove;
}

// get the score for the given board position //
int ChessTranspositionTableElement::GetScore()
{
   // return the score for the given board position //
   return Score;
}

// get whether the score was an exact value or a cutoff //
bool ChessTranspositionTableElement::WasCutoff()
{
   // return whether a cutoff occurred for the given board position or not //
   return CutoffOccurred;
}
```