```
////////////////////////////////////////////////////////////////////////////////////////
// C++ source code of Wrappers.cpp Version 0.1.1 at gyosothree.com //
////////////////////////////////////////////////////////////////////////////////////////

// include c++ wrappers and the gyosothree engine //
#include "Wrappers.h"
#include "GyosoThreeEngine.h"

// wrapper to create new chess game data //
void *CreateNew()
{
  // return the new chess game data //
  return new ChessData;
}

// wrapper to copy game data from source to destination //
void CopyData(void *Destination, void *Source)
{
  // copy game data from source to destination //
  *((ChessData *) Destination) = *((ChessData *) Source);
}

// wrapper to write the chess game pgn to a data file //
void DataWritePGN(void *Data, const char *FileName, bool BlackMoves)
{
  // write the chess game pgn to a data file //
  ((ChessData *) Data)->WritePGN(FileName, BlackMoves);
}

// wrapper to append the chess game pgn to a data file //
void DataAppendPGN(void *Data, const char *FileName, bool BlackMoves)
{
  // append the chess game pgn to a data file //
  ((ChessData *) Data)->AppendPGN(FileName, BlackMoves);
}

// wrapper to initialize the game data //
void DataInitializeAttributes(void *Data)
{
  // initialize the game data //
  ((ChessData *) Data)->InitializeAttributes();
}

// wrapper to get the fen string //
char *DataGetFENString(void *Data)
{
  // create string to hold the game data fen string //
  std::string FENString = ((ChessData *) Data)->GetFENString();

  // return the game data fen string //
  return strcpy((char *) malloc(FENString.length() + 1), FENString.c_str());
}

// wrapper to get the white player name //
char *DataGetWhitePlayer(void *Data)
{
  // create string to hold the white player name //
  std::string WhitePlayer = ((ChessData *) Data)->WhitePlayer();

  // return the white player name //
  return strcpy((char *) malloc(WhitePlayer.length() + 1), WhitePlayer.c_str());
}

// wrapper to get the black player name //
char *DataGetBlackPlayer(void *Data)
{
  // create string to hold the black player name //
  std::string BlackPlayer = ((ChessData *) Data)->BlackPlayer();

  // return the black player name //
  return strcpy((char *) malloc(BlackPlayer.length() + 1), BlackPlayer.c_str());
}

// wrapper to get the time control //
char *DataGetTimeControl(void *Data)
{
  // create string to hold the time control //
  std::string TimeControl = ((ChessData *) Data)->GetTimeControl();

  // return the time control //
  return strcpy((char *) malloc(TimeControl.length() + 1), TimeControl.c_str());
}

// wrapper to get the opening //
char *DataGetOpening(void *Data)
{
  // create string to hold the opening //
  std::string Opening = ((ChessData *) Data)->GetOpening();

  // return the opening //
  return strcpy((char *) malloc(Opening.length() + 1), Opening.c_str());
}

// wrapper to get the section //
char *DataGetSection(void *Data)
{
  // create string to hold the section //
  std::string Section = ((ChessData *) Data)->GetSection();

  // return the section //
  return strcpy((char *) malloc(Section.length() + 1), Section.c_str());
}

// wrapper to get the board number //
char *DataGetBoardNumber(void *Data)
{
  // create string to hold the board number //
  std::string BoardNumber = ((ChessData *) Data)->GetBoardNumber();

  // return the board number //
  return strcpy((char *) malloc(BoardNumber.length() + 1), BoardNumber.c_str());
}

// wrapper to get the white elo //
char *DataGetWhiteElo(void *Data)
{
  // create string to hold the white elo //
  std::string WhiteElo = ((ChessData *) Data)->GetWhiteElo();

  // return the white elo //
  return strcpy((char *) malloc(WhiteElo.length() + 1), WhiteElo.c_str());
}

// wrapper to get the black elo //
char *DataGetBlackElo(void *Data)
{
  // create string to hold the black elo //
  std::string BlackElo = ((ChessData *) Data)->GetBlackElo();

  // return the black elo //
  return strcpy((char *) malloc(BlackElo.length() + 1), BlackElo.c_str());
}

// wrapper to get the white usef //
char *DataGetWhiteUSCF(void *Data)
{
  // create string to hold the white usef //
  std::string WhiteUSCF = ((ChessData *) Data)->GetWhiteUSCF();

  // return the white usef //
  return strcpy((char *) malloc(WhiteUSCF.length() + 1), WhiteUSCF.c_str());
}

// wrapper to get the black usef //
char *DataGetBlackUSCF(void *Data)
{
  // create string to hold the black usef //
  std::string BlackUSCF = ((ChessData *) Data)->GetBlackUSCF();

  // return the black usef //
  return strcpy((char *) malloc(BlackUSCF.length() + 1), BlackUSCF.c_str());
}

// wrapper to get the annotator //
char *DataGetAnnotator(void *Data)
{
  // create string to hold the annotator //
  std::string Annotator = ((ChessData *) Data)->GetAnnotator();

  // return the annotator //
  return strcpy((char *) malloc(Annotator.length() + 1), Annotator.c_str());
}

// wrapper to get the eco //
char *DataGetECO(void *Data)
{
  // create string to hold the eco //
  std::string ECO = ((ChessData *) Data)->GetECO();

  // return the eco //
  return strcpy((char *) malloc(ECO.length() + 1), ECO.c_str());
}

// wrapper to get the site //
char *DataGetSite(void *Data)
{
  // create string to hold the site //
  std::string Site = ((ChessData *) Data)->GetSite();

  // return the site //
  return strcpy((char *) malloc(Site.length() + 1), Site.c_str());
}

// wrapper to get the event //
char *DataGetEvent(void *Data)
{
  // create string to hold the event //
  std::string Event = ((ChessData *) Data)->GetEvent();

  // return the event //
  return strcpy((char *) malloc(Event.length() + 1), Event.c_str());
}

// wrapper to get the round //
char *DataGetRound(void *Data)
{
  // create string to hold the round //
  std::string Round = ((ChessData *) Data)->GetRound();

  // return the round //
  return strcpy((char *) malloc(Round.length() + 1), Round.c_str());
}

// wrapper to get the result //
char *DataGetResult(void *Data)
{
  // create string to hold the result //
  std::string Result = ((ChessData *) Data)->GetResult();

  // return the result //
  return strcpy((char *) malloc(Result.length() + 1), Result.c_str());
}

// wrapper to get the date //
char *DataGetDate(void *Data)
{
  // create string to hold the date //
  std::string Date = ((ChessData *) Data)->GetDate();

  // return the date //
  return strcpy((char *) malloc(Date.length() + 1), Date.c_str());
}

// wrapper to get the game data url //
char *DataGetGameDataUrl(void *Data)
{
  // create string to hold the game data url //
  std::string UCI = ((ChessData *) Data)->CreateJsonUCI();

  // return the game data string url //
  return strcpy((char *) malloc(UCI.length() + 1), UCI.c_str());
}

// wrapper to get the game data annotations //
char *DataGetGameDataAnnotations(void *Data)
{
  // create string to hold the game data annotations //
  std::string Annotations = ((ChessData *) Data)->ConstructAnnotations();

  // return the game data string annotations //
  return strcpy((char *) malloc(Annotations.length() + 1), Annotations.c_str());
}

// wrapper to store the game data header //
void DataStoreGameHeader(void *Data, const char *WhiteName, const char *BlackName,
                         const char *EloCalculus, const char *EventTitle, const char
                         *RoundNumber, const char *EndResult, const char
                         *TournamentSite, const char *EventDate, const char
                         *WhiteElo, const char *BlackElo, const char *WhiteUSCF,
                         const char *BlackUSCF, const char *Annotator, const char *ECO,
                         const char *Opening, const char *Section, const char
                         *BoardNumber, const char *FENString)
{
  // store the game data header //
  ((ChessData *) Data)->StoreGameHeader(WhiteName, BlackName, EloCalculus,
                                        EventTitle, RoundNumber, EndResult,
                                        TournamentSite, EventDate, WhiteElo,
                                        BlackElo, WhiteUSCF, BlackUSCF, Annotator, ECO,
                                        Opening, Section, BoardNumber, FENString);
}

// wrapper to store a move and annotation to the game data //
void DataStoreMoveAnnotation(void *Data, const char *NewMove, const char *Annotation)
{
  // store a move and annotation to the game data //
  ((ChessData *) Data)->StoreMoveAndAnnotation(NewMove, Annotation);
}

// wrapper to create new chess database //
void *CreateNewDB()
{
  // return the new chess database //
  return new ChessDatabase;
}

// wrapper to get game data from the game database //
void *DataBaseGetGameData(void *Data, void *Database, int GameNumber)
{
  // get the game data from the game database //
  *((ChessData *) Data) = ((ChessDatabase *) Database)->GetGameData(GameNumber);

  // return the void pointer to the game data //
  return Data;
}

// wrapper to get the database data list size //
int DataBaseGetDataListSize(void *Database)
{
  // return the database data list size //
  return ((ChessDatabase *) Database)->GetDataListSize();
}

// wrapper to read in a game collection //
void DataBaseReadGameCollection(void *Database, const char *GameFile)
{
  // read in the game collection //
  ((ChessDatabase *) Database)->ReadGameCollection(GameFile);
}

// wrapper to get the game data url //
char *DataBaseGetGameDataUrl(void *Database, int OpeningBookIndex)
{
  // create string to hold the game data url //
  std::string UCI = ((ChessDatabase *) Database)->GetGameDataOpeningBookIndex(
                    OpeningBookIndex, ConstructUCI());

  // return the game data string url //
  return strcpy((char *) malloc(UCI.length() + 1), UCI.c_str());
}

// wrapper to get the game descriptions //
char *DataBaseGetGameDescriptions(void *Database)
{
  // create string to hold the game descriptions //
  std::string Descriptions = ((ChessDatabase *) Database)->GetGameDescriptions();

  // return the game descriptions //
  return strcpy((char *) malloc(Descriptions.length() + 1), Descriptions.c_str());
}

// wrapper to create a new opening book //
void *OpeningBookNew()
{
  // create a new opening book //
  return new ChessOpeningBook;
}

// wrapper to initialize the opening book attributes //
void OpeningBookInitializeAttributes(void *OpeningBook)
{
  // initialize the opening book attributes //
  ((ChessOpeningBook *) OpeningBook)->InitializeAttributes();
}

// wrapper to determine if a move exists in the opening book //
bool OpeningBookMoveExists(void *OpeningBook, unsigned long long EndsetHash)
{
  // determine if a move exists in the opening book //
  return ((ChessOpeningBook *) OpeningBook)->MoveExists(EndsetHash);
}

// wrapper to store data in the opening book //
void OpeningBookStoreOpeningBookData(void *OpeningBook, unsigned long long
                                     EndsetHash, const char *NextMove)
{
  // store data in the opening book //
  ((ChessOpeningBook *) OpeningBook)->StoreOpeningBookData(EndsetHash, NextMove);
}

// wrapper to get the next move in the opening book //
char *OpeningBookGetNextMove(void *OpeningBook)
{
  // create string to hold the next move in the opening book //
  std::string NextMove = ((ChessOpeningBook *) OpeningBook)->GetNextMove();

  // return the next move in the opening book //
  return strcpy((char *) malloc(NextMove.length() + 1), NextMove.c_str());
}

// wrapper to get the list of capture squares //
char *EnumMoveListOfCaptureSquares(void *Board)
{
  // create string to hold the list of capture squares //
  std::string Captures = ((ChessBoard *) Board)->EnumListOfCapturedSquares();

  // return the list of capture squares //
  return strcpy((char *) malloc(Captures.length() + 1), Captures.c_str());
}

// wrapper to get the list of threatened squares //
char *EnumMoveListOfThreatenedSquares(void *Board)
{
  // create string to hold the list of threatened squares //
  std::string Threatened = ((ChessBoard *) Board)->EnumListOfThreatenedSquares();

  // return the list of threatened squares //
  return strcpy((char *) malloc(Threatened.length() + 1), Threatened.c_str());
}

// wrapper to determine if king was captured //
bool BoardKingIsCaptured(void *Board)
{
  // return whether king was captured //
  return ((ChessBoard *) Board)->KingIsCaptured();
}

// wrapper to determine if board is white to move next //
bool BoardIsWhiteToMove(void *Board)
{
  // return whether board is white to move next //
  return ((ChessBoard *) Board)->IsWhite();
}

// wrapper to determine if board is black to move next //
bool BoardIsBlack(void *Board)
{
  // return whether board is black to move next //
  return ((ChessBoard *) Board)->IsBlack();
}

// wrapper to determine if the game is over //
bool BoardGameOver(void *Board)
{
  // return whether game is over //
  return ((ChessBoard *) Board)->GameOver();
}

// wrapper to determine if there is a checkmate //
bool BoardIsCheckmate(void *Board)
{
  // return if there is a checkmate //
  return ((ChessBoard *) Board)->IsCheckmate();
}

// wrapper to determine if there is a stalemate //
bool BoardIsStalemate(void *Board)
{
  // return if there is a stalemate //
  return ((ChessBoard *) Board)->IsStalemate();
}

// wrapper to determine if threefold repetition has occurred //
bool BoardIsThreefoldRepetition(void *Board)
{
  // return if there is a threefold repetition //
  return ((ChessBoard *) Board)->IsThreefoldRepetition();
}

// wrapper to determine if there is insufficient material for mate //
bool BoardIsufficientMaterialToMate(void *Board)
{
  // return if there is insufficient material to mate //
  return ((ChessBoard *) Board)->InsufficientMaterialToMate();
}

// wrapper to determine if an en passant move was just made on the board //
bool BoardIsEnpassant(void *Board)
{
  // return if an en passant move was just made on the board //
  return ((ChessBoard *) Board)->IsEnpassant();
}

// wrapper to determine if the board is in check //
bool BoardIsInCheck(void *Board)
{
  // return if the board is in check //
  return ((ChessBoard *) Board)->InCheck();
}

// wrapper to initialize the board attributes //
void BoardInitializeAllAttributes(void *Board, const char *FENLayout, const char
                                  *FENActiveColor, const char *FENCastleInfo, const char *FENClock, const char
                                  *FENEnpassantSquare)
{
  // initialize the board attributes //
  ((ChessBoard *) Board)->InitializeAllAttributes(FENLayout, FENCastleInfoBits, FENClock,
                                                  FENEnpassantSquare);
}

// wrapper to create a new board //
void *BoardNew()
{
  // create a new board //
  return new ChessBoard;
}

// wrapper to copy board from from source to destination //
void BoardCopy(void *Destination, void *Source)
{
  // copy board from source to destination //
  *((ChessBoard *) Destination) = *((ChessBoard *) Source);
}

// wrapper to find the given move on the board //
bool BoardFindGivenMove(void *Board, const char *Move)
{
  // find the given move on the board //
  return ((ChessBoard *) Board)->FindGivenMove(Move);
}

// wrapper to make the given move on the board //
void BoardMakeMove(void *Board, int Depth)
{
  // make the given move on the board //
  ((ChessBoard *) Board)->MakeMove(Depth);
}

// wrapper to get the zobrist hash for the current board position //
unsigned long long BoardGetZobristHash(void *Board)
{
  // get the zobrist hash for the current board position //
  return ((ChessBoard *) Board)->GetZobristHash();
}

// wrapper to find the uci label for the move //
char *BoardGetLabel(void *Board)
{
  // create string to hold the uci label for the move //
  std::string UCI = ((ChessBoard *) Board)->UCILabel();

  // return the uci label for the move //
  return strcpy((char *) malloc(UCI.length() + 1), UCI.c_str());
}

// wrapper to find the pgn label for the move //
char *BoardPGNLabel(void *Board)
{
  // create string to hold the pgn label for the move //
  std::string PGN = ((ChessBoard *) Board)->PGNLabel();

  // return the pgn label for the move //
  return strcpy((char *) malloc(PGN.length() + 1), PGN.c_str());
}

// wrapper to create a new engine //
void *EngineNew()
{
  // create a new engine //
  return new ChessEngine;
}

// wrapper to initialize the transposition table //
void EngineInitializeTranspositionTable(void *Engine)
{
  // initialize the transposition table //
  ((ChessEngine *) Engine)->InitializeTranspositionTable();
}

// wrapper to make the engine stop //
void EngineMakeEngineStop(void *Engine)
{
  // make the engine stop //
  ((ChessEngine *) Engine)->MakeEngineStop();
}

// wrapper to compact the transposition table //
void EngineCompactTranspositionTable(void *Engine, bool AgeForward)
{
  // compact the transposition table //
  ((ChessEngine *) Engine)->CompactTranspositionTableAgeForward();
}

// wrapper to find the best move on the board //
char *EngineFindBestMove(void *Engine, void *Board, int TimeLimit, int PlyLimit)
{
  // create string to hold the best move //
  std::string BestMove = ((ChessEngine *) Engine)->FindBestMove(
                         (ChessBoard *) Board, TimeLimit, PlyLimit);

  // return the best move //
  return strcpy((char *) malloc(BestMove.length() + 1), BestMove.c_str());
}

// wrapper to see if there is only one possible move on the board //
bool OnlyOnePossibleMove(void *Board)
{
  // return if there is only one possible move on the board //
  return ((ChessBoard *) Board)->OnlyOnePossibleMove();
}
```