Dagor-in-Erain

v0.0.0

Generated by Doxygen 1.10.0

Chapter 1

Dagor-in-Erain

Dagor-in-Erain (Sindarin ‘battle of the kings’) is a chess engine by Jakob Teuber.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Dagor Namespace Reference

Namespaces

- namespace BitBoards
- namespace Board
- namespace MoveTables

5.2 Dagor::BitBoards Namespace Reference

Classes

- class BitBoard

  BitBoards represent some subset of the chess boards pieces, such as the set of all fields occupied by whit pawns or the set of all pieces to which a given piece can move on it's next turn etc.

Functions

- std::ostream & operator<< (std::ostream &out, const BitBoard &board)
- BitBoard operator& (BitBoard a, BitBoard b)
- BitBoard operator| (BitBoard a, BitBoard b)
- BitBoard operator~ (BitBoard a)
- bool operator== (BitBoard a, BitBoard b)
- bool operator!= (BitBoard a, BitBoard b)

Variables

- const BitBoard edgesOnly {0xff818181818181ff}

  A bitboard containing all squares adjacent to one of the edges of the board.

5.2.1 Function Documentation

5.2.1.1 operator"!=()

```
bool Dagor::BitBoards::operator!= (
            BitBoard a,
            BitBoard b )  [inline]
```

5.2.1.2 operator&()

```
BitBoard Dagor::BitBoards::operator& (
            BitBoard a,
            BitBoard b )  [inline]
```

5.2.1.3 operator<<()

```
std::ostream & Dagor::BitBoards::operator<< (
            std::ostream & out,
            const BitBoard & board )
```

5.2.1.4 operator==()

```
bool Dagor::BitBoards::operator== (
            BitBoard a,
            BitBoard b )  [inline]
```

5.2.1.5 operator"|()

```
BitBoard Dagor::BitBoards::operator| (
            BitBoard a,
            BitBoard b )  [inline]
```

5.2.1.6 operator~()

```
BitBoard Dagor::BitBoards::operator~ (
            BitBoard a )  [inline]
```

5.2.2 Variable Documentation

5.2.2.1 edgesOnly

```
const BitBoard Dagor::BitBoards::edgesOnly {0xff818181818181ff}  [inline]
```

A bitboard containing all squares adjacent to one of the edges of the board.

```
8 | @ @ @ @ @ @ @ @
7 | @ . . . . . . @
6 | @ . . . . . . @
5 | @ . . . . . . @
4 | @ . . . . . . @
3 | @ . . . . . . @
2 | @ . . . . . . @
1 | @ @ @ @ @ @ @ @
    ----------------     as decimal: 18411139144890810879
    a b c d e f g h      as hex:     0xff818181818181ff
```

5.3 Dagor::Board Namespace Reference

Enumerations

- enum CompassOffsets {
  north_west = +7 , north = +8 , north_east = +9 , west = -1 ,
  east = 1 , south_west = -9 , south = -8 , south_east = -7 }

  The offsets to add to a square index to go in the intended direction.
- enum Square {
  a1 , b1 , c1 , d1 ,
  e1 , f1 , g1 , h1 ,
  a2 , b2 , c2 , d2 ,
  e2 , f2 , g2 , h2 ,
  a3 , b3 , c3 , d3 ,
  e3 , f3 , g3 , h3 ,
  a4 , b4 , c4 , d4 ,
  e4 , f4 , g4 , h4 ,
  a5 , b5 , c5 , d5 ,
  e5 , f5 , g5 , h5 ,
  a6 , b6 , c6 , d6 ,
  e6 , f6 , g6 , h6 ,
  a7 , b7 , c7 , d7 ,
  e7 , f7 , g7 , h7 ,
  a8 , b8 , c8 , d8 ,
  e8 , f8 , g8 , h8 ,
  no_square }

  The names of the squares in algebraic notation. The indices of squares are counted sequentially with `a1` being equal to `0` and `h8` being equal to `63`. A special `no_square` constant denotes an absent value.

Functions

- constexpr int file (int square)

  Computes the rank (i. e. row) of a square from its index.
- constexpr int rank (int square)

  Computes the file (i. e. column) of a square from its index.
- constexpr int index (int file, int rank)

  Computes the index of a square from its file and rank.
- constexpr char file_name (int file)

Variables

- constexpr int width {8}

  The width of a chess board, that is `8`.
- constexpr int size {width * width}

  The number of squares of a chess board, that is `64`.

5.3.1 Enumeration Type Documentation

5.3.1.1 CompassOffsets

`enum Dagor::Board::CompassOffsets`

The offsets to add to a square index to go in the intended direction.

Enumerator

north_west	
north	
north_east	
west	
east	
south_west	
south	
south_east	

5.3.1.2 Square

`enum Dagor::Board::Square`

The names of the squares in algebraic notation. The indices of squares are counted sequentially with `a1` being equal to `0` and `h8` being equal to `63`. A special `no_square` constant denotes an absent value.

Enumerator

a1	
b1	
c1	
d1	
e1	
f1	
g1	
h1	
a2	
b2	
c2	
d2	
e2	
f2	
g2	
h2	
a3	
b3	
c3	
d3	
e3	
f3	
g3	
h3	
a4	
b4	
c4	
d4	
e4	
f4	
g4	

Enumerator

h4	
a5	
b5	
c5	
d5	
e5	
f5	
g5	
h5	
a6	
b6	
c6	
d6	
e6	
f6	
g6	
h6	
a7	
b7	
c7	
d7	
e7	
f7	
g7	
h7	
a8	
b8	
c8	
d8	
e8	
f8	
g8	
h8	
no_square	

5.3.2 Function Documentation

5.3.2.1 file()

```
constexpr int Dagor::Board::file (
            int square )  [constexpr]
```

Computes the rank (i. e. row) of a square from its index.

Parameters

square	the index of the square.

Returns

its rank.

5.3.2.2 file_name()

```
constexpr char Dagor::Board::file_name (
            int file )  [constexpr]
```

Parameters

file	the numeric value of a file (i. e. column) form {0,...,7}.

Returns

the name of that file in algebraic chess notation from {a,...,h}.

5.3.2.3 index()

```
constexpr int Dagor::Board::index (
            int file,
            int rank )  [constexpr]
```

Computes the index of a square from its file and rank.

Parameters

file	file (i. e. column)
rank	rank (i. e. row)

Returns

the index of the specified square.

5.3.2.4 rank()

```
constexpr int Dagor::Board::rank (
            int square )  [constexpr]
```

Computes the file (i. e. column) of a square from its index.

Parameters

square	the index of a square.

Returns

its file.

5.3.3 Variable Documentation

5.3.3.1 size

```
constexpr int Dagor::Board::size {width * width}  [inline], [constexpr]
```

The number of squares of a chess board, that is `64`.

5.3.3.2 width

```
constexpr int Dagor::Board::width {8}  [inline], [constexpr]
```

The width of a chess board, that is `8`.

5.4 Dagor::MoveTables Namespace Reference

Classes

- class BlockerHash

  A hash function that maps a configuration of blocking pieces to an index into the `slidingMoves` table, where the possible moves of a rook or bishop are stored. Both rooks and bishops have one separate hash function for each square.

Variables

- const BitBoard pawnAttacks [2][Board::size]

  The attacks a pawn can make on a given square. Access: `pawnAttacks[color][square]`, where white is `0` and black is `1`.
- const BitBoard knightMoves [Board::size]

  The moves a knight can make on a given square.
- const BitBoard kingMoves [Board::size]

  The moves a king can make on a given square. For his home square this does not include castling moves.
- const BlockerHash bishopHashes [Board::size]

  the hash functions to look up bishop moves, by square.
- const BlockerHash rookHashes [Board::size]

  the hash function to look up rook moves, by square.
- const BitBoards::BitBoard slidingMoves [ ]

  The move that a sliding piece (bishop, rook or queen) can make on a given square. Access through the hash functions in `bishopHashes` and `rookHashes`.

5.4.1 Variable Documentation

5.4.1.1 bishopHashes

```
const BlockerHash Dagor::MoveTables::bishopHashes
```

the hash functions to look up bishop moves, by square.

5.4.1.2 kingMoves

```
const BitBoards::BitBoard Dagor::MoveTables::kingMoves
```

The moves a king can make on a given square. For his home square this does not include castling moves.

5.4.1.3 knightMoves

```
const BitBoards::BitBoard Dagor::MoveTables::knightMoves
```

The moves a knight can make on a given square.

5.4.1.4 pawnAttacks

```
const BitBoards::BitBoard Dagor::MoveTables::pawnAttacks
```

The attacks a pawn can make on a given square. Access: `pawnAttacks[color][square]`, where white is `0` and black is `1`.

5.4.1.5 rookHashes

```
const BlockerHash Dagor::MoveTables::rookHashes
```

the hash function to look up rook moves, by square.

5.4.1.6 slidingMoves

```
const BitBoards::BitBoard Dagor::MoveTables::slidingMoves
```

The move that a sliding piece (bishop, rook or queen) can make on a given square. Access through the hash functions in `bishopHashes` and `rookHashes`.

Chapter 6

Class Documentation

6.1 Dagor::BitBoards::BitBoard Class Reference

BitBoards represent some subset of the chess boards pieces, such as the set of all fields occupied by whit pawns or the set of all pieces to which a given piece can move on it's next turn etc.

`#include <bitboard.h>`

Public Member Functions

- BitBoard ()

  constructs an empty BitBoard.
- BitBoard (std::uint64_t bitboard)
- std::uint64_t as_uint () const
- BitBoard & operator&= (BitBoard other)

  removes all the squares that are not also present in `other`.
- BitBoard & operator|= (BitBoard other)

  adds all the squares of the `other` bitboard to this one.
- constexpr bool is_empty () const

  checks whether the bitboard is empty, that is whether no squares are set.
- constexpr bool is_set (int square) const

  Checks whether a particular square is set.
- void set_bit (int square)

  Adds the given square to the bitboard.
- void set_bit_if_index_valid (int file, int rank)
- void unset_bit (int square)

  Removes a given square from the bitboard.
- constexpr int popcount () const

  Counts the number of set squares in the bitboard.
- constexpr int findFirstSet () const

  Finds the index of the first set square in the bitboard. Do not call this function for the empty bitboard.

Static Public Member Functions

- static BitBoard single_square_set (int square)

  Constructs a bitboard with only a single square set.

6.1.1 Detailed Description

BitBoards represent some subset of the chess boards pieces, such as the set of all fields occupied by whit pawns or the set of all pieces to which a given piece can move on it's next turn etc.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BitBoard() [1/2]

```
Dagor::BitBoards::BitBoard::BitBoard ( )
```

constructs an empty BitBoard.

6.1.2.2 BitBoard() [2/2]

```
Dagor::BitBoards::BitBoard::BitBoard (
            std::uint64_t bitboard )
```

Parameters

bitboard	a uint64 as returned by the `as_uint` function.

6.1.3 Member Function Documentation

6.1.3.1 as_uint()

```
std::uint64_t Dagor::BitBoards::BitBoard::as_uint ( ) const  [inline]
```

Returns

a uint64 where all the 1 bits indicate the set squares

6.1.3.2 findFirstSet()

```
constexpr int Dagor::BitBoards::BitBoard::findFirstSet ( ) const  [inline], [constexpr]
```

Finds the index of the first set square in the bitboard. Do not call this function for the empty bitboard.

Returns

the index of the first set square.

6.1.3.3 is_empty()

```
constexpr bool Dagor::BitBoards::BitBoard::is_empty ( ) const  [inline], [constexpr]
```

checks whether the bitboard is empty, that is whether no squares are set.

Returns

`true`, iff no squares are set.

6.1.3.4 is_set()

```
constexpr bool Dagor::BitBoards::BitBoard::is_set (
            int square ) const  [inline], [constexpr]
```

Checks whether a particular square is set.

Parameters

square	the square to check.

Returns

`true`, iff the square is set.

6.1.3.5 operator&=()

```
BitBoard & Dagor::BitBoards::BitBoard::operator&= (
            BitBoard other )  [inline]
```

removes all the squares that are not also present in `other`.

Parameters

other	

Returns

this bitboard after the modification

6.1.3.6 operator"|=()

```
BitBoard & Dagor::BitBoards::BitBoard::operator|= (
            BitBoard other )  [inline]
```

adds all the squares of the `other` bitboard to this one.

Parameters

other	

Returns

this bitboard after the modification

6.1.3.7 popcount()

```
constexpr int Dagor::BitBoards::BitBoard::popcount ( ) const  [inline], [constexpr]
```

Counts the number of set squares in the bitboard.

Returns

the number of set squares in the bitboard.

6.1.3.8 set_bit()

```
void Dagor::BitBoards::BitBoard::set_bit (
            int square )  [inline]
```

Adds the given square to the bitboard.

Parameters

square	the square to add.

6.1.3.9 set_bit_if_index_valid()

```
void Dagor::BitBoards::BitBoard::set_bit_if_index_valid (
            int file,
            int rank )  [inline]
```

Adds the given square to the bitboard, if the coordinates are valid on a chess board, that is, if `file, rank are from {0,...,7}`. If this is not the case, nothing happens. This function exists to protect against warping around the edges of the board when calculating moves etc.

Parameters

file	the file (i. e. column) of the square to add.
rank	the rank (i. e. row) of the square to add.

6.1.3.10 single_square_set()

```
static BitBoard Dagor::BitBoards::BitBoard::single_square_set (
            int square )  [inline], [static]
```

Constructs a bitboard with only a single square set.

Parameters

square	the square to be set

Returns

the bitboard

6.1.3.11 unset_bit()

```
void Dagor::BitBoards::BitBoard::unset_bit (
            int square )  [inline]
```

Removes a given square from the bitboard.

Parameters

square	the square to remove.

The documentation for this class was generated from the following files:

- bitboard.h
- bitboard.cpp

6.2 Dagor::MoveTables::BlockerHash Class Reference

A hash function that maps a configuration of blocking pieces to an index into the `slidingMoves` table, where the possible moves of a rook or bishop are stored. Both rooks and bishops have one separate hash function for each square.

```
#include <movetables.h>
```

Public Member Functions

- BlockerHash (BitBoards::BitBoard mask, BitBoards::BitBoard magic, unsigned downShift, unsigned tableOffset)
- unsigned hash (BitBoards::BitBoard blockers) const

  Computes the hash for a configuration of blocking pieces.
- BitBoards::BitBoard lookUp (BitBoards::BitBoard blockers) const

  Looks up the possible moves for a bishop/rook with the specified blocking pieces.

Public Attributes

- const BitBoards::BitBoard blockerMask

  The mask singling out the blocking pieces that actually matter to the figure under consideration.
- const BitBoards::BitBoard magic

  The magic number that yields the perfect hash function.
- const unsigned downShift

  The amount by which the hash should be shifted down.
- const unsigned tableOffset

  The offset that should be added to the hash. In `slidingMoves` all entries lie consecutively, this marks where the entries begin, that can be accessed through this hash function.

6.2.1 Detailed Description

A hash function that maps a configuration of blocking pieces to an index into the `slidingMoves` table, where the possible moves of a rook or bishop are stored. Both rooks and bishops have one separate hash function for each square.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 BlockerHash()

```
Dagor::MoveTables::BlockerHash::BlockerHash (
            BitBoards::BitBoard mask,
            BitBoards::BitBoard magic,
            unsigned downShift,
            unsigned tableOffset )  [inline]
```

6.2.3 Member Function Documentation

6.2.3.1 hash()

```
unsigned Dagor::MoveTables::BlockerHash::hash (
            BitBoards::BitBoard blockers ) const  [inline]
```

Computes the hash for a configuration of blocking pieces.

Parameters

blockers	pieces blocking the bishop's/rook's movement.

Returns

the hash.

6.2.3.2 lookUp()

```
BitBoards::BitBoard Dagor::MoveTables::BlockerHash::lookUp (
```

```
BitBoards::BitBoard blockers ) const  [inline]
```

Looks up the possible moves for a bishop/rook with the specified blocking pieces.

Parameters

blockers	pieces blocking the bishop's/rook's movement.

Returns

a bitboard where all squares, to which the bishop/rook can move, are set.

6.2.4 Member Data Documentation

6.2.4.1 blockerMask

```
const BitBoards::BitBoard Dagor::MoveTables::BlockerHash::blockerMask
```

The mask singling out the blocking pieces that actually matter to the figure under consideration.

6.2.4.2 downShift

```
const unsigned Dagor::MoveTables::BlockerHash::downShift
```

The amount by which the hash should be shifted down.

6.2.4.3 magic

```
const BitBoards::BitBoard Dagor::MoveTables::BlockerHash::magic
```

The magic number that yields the perfect hash function.

6.2.4.4 tableOffset

```
const unsigned Dagor::MoveTables::BlockerHash::tableOffset
```

The offset that should be added to the hash. In `slidingMoves` all entries lie consecutively, this marks where the entries begin, that can be accessed through this hash function.

The documentation for this class was generated from the following file:

- movetables.h

6.3 LeaperInfo Class Reference

Public Member Functions

- LeaperInfo (bool isBishop, int square)

Public Attributes

- bool isBishop
- int square
- BitBoard blockerMask
- vector< BitBoard > blockers
- vector< BitBoard > moves

6.3.1 Constructor & Destructor Documentation

6.3.1.1 LeaperInfo()

```
LeaperInfo::LeaperInfo (
            bool isBishop,
            int square )  [inline]
```

6.3.2 Member Data Documentation

6.3.2.1 blockerMask

```
BitBoard LeaperInfo::blockerMask
```

6.3.2.2 blockers

```
vector<BitBoard> LeaperInfo::blockers
```

6.3.2.3 isBishop

```
bool LeaperInfo::isBishop
```

6.3.2.4 moves

```
vector<BitBoard> LeaperInfo::moves
```

6.3.2.5 square

```
int LeaperInfo::square
```

The documentation for this class was generated from the following file:

- generate_movetables.cpp

Chapter 7

File Documentation

7.1 bitboard.cpp File Reference

```
#include "bitboard.h"
#include <ios>
#include "board.h"
```

Namespaces

- namespace Dagor
- namespace Dagor::BitBoards

Functions

- std::ostream & Dagor::BitBoards::operator<< (std::ostream &out, const BitBoard &board)

7.2 bitboard.h File Reference

```
#include <cstdint>
#include <ostream>
#include "board.h"
```

Classes

- class Dagor::BitBoards::BitBoard

  BitBoards represent some subset of the chess boards pieces, such as the set of all fields occupied by whit pawns or the set of all pieces to which a given piece can move on it's next turn etc.

Namespaces

- namespace Dagor
- namespace Dagor::BitBoards

Functions

- BitBoard Dagor::BitBoards::operator& (BitBoard a, BitBoard b)
- BitBoard Dagor::BitBoards::operator| (BitBoard a, BitBoard b)
- BitBoard Dagor::BitBoards::operator~ (BitBoard a)
- bool Dagor::BitBoards::operator== (BitBoard a, BitBoard b)
- bool Dagor::BitBoards::operator!= (BitBoard a, BitBoard b)
- std::ostream & Dagor::BitBoards::operator<< (std::ostream &out, const BitBoard &board)

Variables

- const BitBoard Dagor::BitBoards::edgesOnly {0xff818181818181ff}

  A bitboard containing all squares adjacent to one of the edges of the board.

7.3 bitboard.h

Go to the documentation of this file.

```
00001
00002 #ifndef BITBOARD_H
00003 #define BITBOARD_H
00004
00005 #include <cstdint>
00006 #include <ostream>
00007
00008 #include "board.h"
00009
00010 namespace Dagor::BitBoards {
00011
00012 static_assert(sizeof(std::uint64_t) == 8,
00013               "For its BitBoards, this program assumes 64 bit integers.");
00014
00019 class BitBoard {
00020  private:
00021   std::uint64_t board;
00022
00023  public:
00025   BitBoard();
00028   BitBoard(std::uint64_t bitboard);
00029
00032   std::uint64_t as_uint() const { return board; }
00033
00037   BitBoard &operator&=(BitBoard other) {
00038     board &= other.board;
00039     return *this;
00040   }
00041
00045   BitBoard &operator|=(BitBoard other) {
00046     board |= other.board;
00047     return *this;
00048   }
00049
00053   static inline BitBoard single_square_set(int square) {
00054     return BitBoard(static_cast<std::uint64_t>(1) << square);
00055   }
00056
00060   constexpr bool is_empty() const { return board == 0; }
00061
00065   constexpr bool is_set(int square) const {
00066     return (board & (static_cast<std::uint64_t>(1) << square)) != 0;
00067   }
00068
00071   void set_bit(int square) { *this |= single_square_set(square); }
00072
00079   void set_bit_if_index_valid(int file, int rank) {
00080     if (0 <= file && file < Board::width && 0 <= rank && rank < Board::width) {
00081       set_bit(Board::index(file, rank));
00082     }
00083   }
00084
00087   void unset_bit(int square) { board &= ~single_square_set(square).board; }
00088
00091   constexpr int popcount() const { return __builtin_popcountll(board); }
00092
```

```
00096   constexpr int findFirstSet() const { return __builtin_ctzll(board); }
00097 };
00098
00099 inline BitBoard operator&(BitBoard a, BitBoard b) { return a &= b; }
00100 inline BitBoard operator|(BitBoard a, BitBoard b) { return a |= b; }
00101 inline BitBoard operator~(BitBoard a) { return BitBoard(~a.as_uint()); }
00102
00103 inline bool operator==(BitBoard a, BitBoard b) {
00104   return a.as_uint() == b.as_uint();
00105 }
00106 inline bool operator!=(BitBoard a, BitBoard b) {
00107   return a.as_uint() != b.as_uint();
00108 }
00109
00110 std::ostream &operator«(std::ostream &out, const BitBoard &printer);
00111
00125 inline const BitBoard edgesOnly{0xff818181818181ff};
00126
00127 }  // namespace Dagor::BitBoards
00128
00129 #endif
```

7.4 board.h File Reference

`#include <string_view>`

Namespaces

- namespace Dagor
- namespace Dagor::Board

Enumerations

- enum Dagor::Board::CompassOffsets {
  Dagor::Board::north_west = +7 , Dagor::Board::north = +8 , Dagor::Board::north_east = +9 , Dagor::Board::west = -1 ,
  Dagor::Board::east = 1 , Dagor::Board::south_west = -9 , Dagor::Board::south = -8 , Dagor::Board::south_east = -7 }

  The offsets to add to a square index to go in the intended direction.
- enum Dagor::Board::Square {
  Dagor::Board::a1 , Dagor::Board::b1 , Dagor::Board::c1 , Dagor::Board::d1 ,
  Dagor::Board::e1 , Dagor::Board::f1 , Dagor::Board::g1 , Dagor::Board::h1 ,
  Dagor::Board::a2 , Dagor::Board::b2 , Dagor::Board::c2 , Dagor::Board::d2 ,
  Dagor::Board::e2 , Dagor::Board::f2 , Dagor::Board::g2 , Dagor::Board::h2 ,
  Dagor::Board::a3 , Dagor::Board::b3 , Dagor::Board::c3 , Dagor::Board::d3 ,
  Dagor::Board::e3 , Dagor::Board::f3 , Dagor::Board::g3 , Dagor::Board::h3 ,
  Dagor::Board::a4 , Dagor::Board::b4 , Dagor::Board::c4 , Dagor::Board::d4 ,
  Dagor::Board::e4 , Dagor::Board::f4 , Dagor::Board::g4 , Dagor::Board::h4 ,
  Dagor::Board::a5 , Dagor::Board::b5 , Dagor::Board::c5 , Dagor::Board::d5 ,
  Dagor::Board::e5 , Dagor::Board::f5 , Dagor::Board::g5 , Dagor::Board::h5 ,
  Dagor::Board::a6 , Dagor::Board::b6 , Dagor::Board::c6 , Dagor::Board::d6 ,
  Dagor::Board::e6 , Dagor::Board::f6 , Dagor::Board::g6 , Dagor::Board::h6 ,
  Dagor::Board::a7 , Dagor::Board::b7 , Dagor::Board::c7 , Dagor::Board::d7 ,
  Dagor::Board::e7 , Dagor::Board::f7 , Dagor::Board::g7 , Dagor::Board::h7 ,
  Dagor::Board::a8 , Dagor::Board::b8 , Dagor::Board::c8 , Dagor::Board::d8 ,
  Dagor::Board::e8 , Dagor::Board::f8 , Dagor::Board::g8 , Dagor::Board::h8 ,
  Dagor::Board::no_square }

  The names of the squares in algebraic notation. The indices of squares are counted sequentially with `a1` being equal to `0` and `h8` being equal to `63`. A special `no_square` constant denotes an absent value.

Functions

- constexpr int Dagor::Board::file (int square)

  Computes the rank (i. e. row) of a square from its index.
- constexpr int Dagor::Board::rank (int square)

  Computes the file (i. e. column) of a square from its index.
- constexpr int Dagor::Board::index (int file, int rank)

  Computes the index of a square from its file and rank.
- constexpr char Dagor::Board::file_name (int file)

Variables

- constexpr int Dagor::Board::width {8}

  The width of a chess board, that is `8`.
- constexpr int Dagor::Board::size {width * width}

  The number of squares of a chess board, that is `64`.

7.5 board.h

Go to the documentation of this file.

```
00001 #ifndef BOARD_H
00002 #define BOARD_H
00003
00004 #include <string_view>
00005
00006 namespace Dagor::Board {
00007
00009 inline constexpr int width{8};
00011 inline constexpr int size{width * width};
00012
00016 constexpr int file(int square) { return square % width; }
00017
00021 constexpr int rank(int square) { return square / width; }
00022
00027 constexpr int index(int file, int rank) { return file + width * rank; }
00028
00031 constexpr char file_name(int file) { return static_cast<char>('a' + file); }
00032
00034 enum CompassOffsets {
00035   north_west = +7,
00036   north = +8,
00037   north_east = +9,
00038   west = -1,
00039   east = 1,
00040   south_west = -9,
00041   south = -8,
00042   south_east = -7
00043 };
00044
00048 enum Square {
00049   a1,
00050   b1,
00051   c1,
00052   d1,
00053   e1,
00054   f1,
00055   g1,
00056   h1,
00057   a2,
00058   b2,
00059   c2,
00060   d2,
00061   e2,
00062   f2,
00063   g2,
00064   h2,
00065   a3,
00066   b3,
00067   c3,
00068   d3,
```

```
00069   e3,
00070   f3,
00071   g3,
00072   h3,
00073   a4,
00074   b4,
00075   c4,
00076   d4,
00077   e4,
00078   f4,
00079   g4,
00080   h4,
00081   a5,
00082   b5,
00083   c5,
00084   d5,
00085   e5,
00086   f5,
00087   g5,
00088   h5,
00089   a6,
00090   b6,
00091   c6,
00092   d6,
00093   e6,
00094   f6,
00095   g6,
00096   h6,
00097   a7,
00098   b7,
00099   c7,
00100   d7,
00101   e7,
00102   f7,
00103   g7,
00104   h7,
00105   a8,
00106   b8,
00107   c8,
00108   d8,
00109   e8,
00110   f8,
00111   g8,
00112   h8,
00113   no_square
00114 };
00115
00116 }  // namespace Dagor::Board
00117
00118 #endif
```

7.6 constants.h File Reference

Enumerations

- enum Color { white , black }

7.6.1 Enumeration Type Documentation

7.6.1.1 Color

```
enum Color
```

Enumerator

white	
black	

7.7 constants.h

Go to the documentation of this file.

```
00001 #ifndef CONSTANTS_H
00002 #define CONSTANTS_H
00003
00004 enum Color { white, black };
00005
00006 #endif
```

7.8 generate_movetables.cpp File Reference

```
#include <cassert>
#include <cstdlib>
#include <fstream>
#include <ios>
#include <iostream>
#include <random>
#include <vector>
#include "bitboard.h"
#include "board.h"
#include "constants.h"
#include "movetables.h"
```

Classes

- class LeaperInfo

Functions

- BitBoard pawnAttack (int square, int color)

  Compute the attacks that a pawn can make on a given square.
- void writePawnAttacks (std::ofstream &f)

  writes all possible pawn attacks into a file.
- BitBoard knightMove (int square)

  Computes all moves a knight can make on a given square.
- void writeKnightMoves (std::ofstream &f)

  writes the knight moves to a file.
- BitBoard kingMove (int square)

  Computes the possible moves of a king on a given square (assuming the board is empty otherwise). This includes only the 'standard' moves, not castling, which needs to be handled as a special case.
- void writeKingMoves (std::ofstream &f)

  writes the king moves to a file.
- BitBoard bishopBlockers (int square)

  Computes a mask, where all locations are set, where a blocking piece could impede the further movement of a bishop. Squares on the edge are not considered, because they can only be endpoints of a move anyway. For example, this is the result for a bishop on d5:
- BitBoard bishopMoveRay (int square, bool fileUp, bool rankUp, BitBoard blockers)

  Computes one ray of a bishops movement.
- BitBoard bishopMoves (int square, BitBoard blockers)

  Computes the moves for a bishop.

- BitBoard rookBlockers (int square)

  Computes a mask, where all locations are set, where a blocking piece could impede the further movement of a rook. Squares on the edge are not considered, because they can only be endpoints of a move anyway. For example, this is the result for a rook on d5:
- BitBoard rookMoveRay (int square, int addFile, int addRank, BitBoard blockers)
- BitBoard rookMoves (int square, BitBoard blockers)

  Computes the possible move for a rook.
- BitBoard spreadBitsInMask (unsigned bitsToSpread, BitBoard mask)
- std::uint64_t randomLong ()

  Generates an uniformly distributed uint64.
- std::uint64_t randomFewBitsSet ()

  Generates a random number with a bias towards numbers where only a few bits are set.
- vector< BitBoard > generatePossibleBlockers (BitBoard mask)

  Generates all possibilities for how blocking pieces can be distributed within the range of the given mask. Mathematically speaking, this produces the powerset of the mask.
- vector< BitBoard > generatePossibleMoves (int square, const vector< BitBoard > &blockers, bool isBishop)
- MoveTables::BlockerHash findPerfectHash (LeaperInfo &info)

  Finds the configuration for a perfect hash function for the powerset of the given mask.
- void writeHash (std::ostream &f, MoveTables::BlockerHash &hash, unsigned offset)

  Write the representation of a hash function to a file.
- void initHashFunctions (vector< LeaperInfo > &squareInfo, vector< MoveTables::BlockerHash > &hash↩ Functions, unsigned &numberOfMoves, bool isBishop)
- void hashMoves (vector< BitBoard > &moves, vector< LeaperInfo > &squareInfo, vector< MoveTables::BlockerHash > &hashFunctions, unsigned &currentOffset)
- void writeLeapers (std::ostream &f)
- int main ()

7.8.1 Detailed Description

This file generates `movetables.cpp`. It pre-calculates the possible moves for a given position to speed up move generation in the search.

7.8.2 Function Documentation

7.8.2.1 bishopBlockers()

```
BitBoard bishopBlockers (
            int square )
```

Computes a mask, where all locations are set, where a blocking piece could impede the further movement of a bishop. Squares on the edge are not considered, because they can only be endpoints of a move anyway. For example, this is the result for a bishop on d5:

```
8 | . . . . . . . .
7 | . @ . . . @ . .
6 | . . @ . @ . . .
5 | . . . . . . . .
4 | . . @ . @ . . .
3 | . @ . . . @ . .
2 | . . . . . . @ .
1 | . . . . . . . .
    ----------------     as decimal: 9592139778506752
    a b c d e f g h      as hex:     0x22140014224000
```

Parameters

square	the position of the bishop.

Returns

a bitboard with the relevant squares set.

7.8.2.2 bishopMoveRay()

```
BitBoard bishopMoveRay (
            int square,
            bool fileUp,
            bool rankUp,
            BitBoard blockers )
```

Computes one ray of a bishops movement.

Parameters

square	the position of the bishop
fileUp	if `true`, the ray will travel to the right of the board, otherwise to the left.
rankUp	if `true`, the ray will travel to the top of the board, otherwise to the bottom.
blockers	the pieces blocking the bishops movement.

Returns

a bitboard with all the squares set, to which the bishop can move in that rey.

7.8.2.3 bishopMoves()

```
BitBoard bishopMoves (
            int square,
            BitBoard blockers )
```

Computes the moves for a bishop.

Parameters

square	the position of the bishop.
blockers	the pieces blocking the bishops movement.

Returns

a bitboard with all the squares set, to which the bishop can move.

7.8.2.4 findPerfectHash()

```
MoveTables::BlockerHash findPerfectHash (
            LeaperInfo & info )
```

Finds the configuration for a perfect hash function for the powerset of the given mask.

Parameters

mask	

Returns

An object implementing the hash function. `MoveTables::BlockerHash({0}, {0}, 0, 0)` is returned if the generation has failed.

7.8.2.5 generatePossibleBlockers()

```
vector< BitBoard > generatePossibleBlockers (
            BitBoard mask )
```

Generates all possibilities for how blocking pieces can be distributed within the range of the given mask. Mathematically speaking, this produces the powerset of the mask.

Parameters

mask	the locations where other pieces could potentially block our movement

Returns

the powerset of that mask.

7.8.2.6 generatePossibleMoves()

```
vector< BitBoard > generatePossibleMoves (
            int square,
            const vector< BitBoard > & blockers,
            bool isBishop )
```

7.8.2.7 hashMoves()

```
void hashMoves (
            vector< BitBoard > & moves,
            vector< LeaperInfo > & squareInfo,
            vector< MoveTables::BlockerHash > & hashFunctions,
            unsigned & currentOffset )
```

7.8.2.8 initHashFunctions()

```
void initHashFunctions (
            vector< LeaperInfo > & squareInfo,
            vector< MoveTables::BlockerHash > & hashFunctions,
            unsigned & numberOfMoves,
            bool isBishop )
```

7.8.2.9 kingMove()

```
BitBoard kingMove (
            int square )
```

Computes the possible moves of a king on a given square (assuming the board is empty otherwise). This includes only the 'standard' moves, not castling, which needs to be handled as a special case.

Parameters

square	the position of the king.

Returns

a bitboard with all the squares set, to which a king can move.

7.8.2.10 knightMove()

```
BitBoard knightMove (
            int square )
```

Computes all moves a knight can make on a given square.

Parameters

square	position of the knight

Returns

a bitboard with all squares set to which the knight could move (assuming the board is otherwise empty).

7.8.2.11 main()

```
int main ( )
```

7.8.2.12 pawnAttack()

```
BitBoard pawnAttack (
            int square,
            int color )
```

Compute the attacks that a pawn can make on a given square.

Parameters

square	the position of the pawn
color	the color of the pawn (`enum Color`). White pawns move upwards, black pawns move downwards.

Returns

a bitboard with the attacked squares set.

7.8.2.13 randomFewBitsSet()

```
std::uint64_t randomFewBitsSet ( )
```

Generates a random number with a bias towards numbers where only a few bits are set.

Returns

the random number.

7.8.2.14 randomLong()

```
std::uint64_t randomLong ( )
```

Generates an uniformly distributed uint64.

Returns

the random number.

7.8.2.15 rookBlockers()

```
BitBoard rookBlockers (
            int square )
```

Computes a mask, where all locations are set, where a blocking piece could impede the further movement of a rook. Squares on the edge are not considered, because they can only be endpoints of a move anyway. For example, this is the result for a rook on d5:

```
8 | . . . . . . . .
7 | . . . @ . . . .
6 | . . . @ . . . .
5 | . @ @ . @ @ @ .
4 | . . . @ . . . .
3 | . . . @ . . . .
2 | . . . @ . . . .
1 | . . . . . . . .
    ----------------     as decimal: 2261102847592448
    a b c d e f g h      as hex:     0x8087608080800
```

Parameters

square	the position of the rook.

Returns

a bitboard with the relevant squares set.

7.8.2.16 rookMoveRay()

```
BitBoard rookMoveRay (
            int square,
            int addFile,
            int addRank,
            BitBoard blockers )
```

Computes one ray of a rooks movement. The pair `(addFile, addRank)` describes the direction of the ray:

- `(+1, 0)` = rook goes to the right
- `(-1, 0)` = rook goes to the right
- `(0, +1)` = rook goes to the top
- `(0, -1)` = rook goes to the bottom

Other values should not be uses.

Parameters

square	the position of the rook
addFile	If `1`, the rook will travel to the right, if `-1` the rook will travel to the left, if `0` the rook won't change its file.
addRank	If `1`, the rook will travel to the top, if `-1` the rook will travel to the bottom, if `0` the rook won't change its rank.
blockers	the pieces blocking the rook's movement.

Returns

a bitboard with the moves of this ray set.

7.8.2.17 rookMoves()

```
BitBoard rookMoves (
            int square,
            BitBoard blockers )
```

Computes the possible move for a rook.

Parameters

square	the position of the rook.
blockers	the pieces blocking the rook's movement.

Returns

a bitboard with all the squares set, to which the bishop can move.

7.8.2.18 spreadBitsInMask()

```
BitBoard spreadBitsInMask (
            unsigned bitsToSpread,
            BitBoard mask )
```

Spreads the given bits out to cover the ones of the mask. If the `n`th bit in `bitsToSpread` is set, then the `n`th set square in mask will be set in the result as well. This is used to generate subsets of the mask.

Example:

```
bitsToSpread: 0b101010101010      →     result:
mask:
8 | . . . . . . . .                     8 | . . . . . . . .
7 | @ . . . . . . .                     7 | . . . . . . . .
6 | @ . . . . . . .                     6 | @ . . . . . . .
5 | @ . . . . . . .                     5 | . . . . . . . .
4 | @ . . . . . . .                     4 | @ . . . . . . .
3 | @ . . . . . . .                     3 | . . . . . . . .
2 | @ . . . . . . .                     2 | @ . . . . . . .
1 | . @ @ @ @ @ @ .                     1 | . @ . @ . @ . .
   -----------------                       -----------------
    a b c d e f g h                         a b c d e f g h
```

Parameters

bitsToSpread	the binary data to fill the mask with.
mask	the places where the binary data should

Returns

A bitboard, in which a square is set iff it's the `n`th set square of the mask and the `n`th bit of `bitsToSpread` is set.

7.8.2.19 writeHash()

```
void writeHash (
            std::ostream & f,
            MoveTables::BlockerHash & hash,
            unsigned offset )
```

Write the representation of a hash function to a file.

Parameters

f	
hash	
offset	a new offset, potentially differing from the one specified in the hash function.

7.8.2.20 writeKingMoves()

```
void writeKingMoves (
            std::ofstream & f )
```

writes the king moves to a file.

Parameters

f	

7.8.2.21 writeKnightMoves()

```
void writeKnightMoves (
            std::ofstream & f )
```

writes the knight moves to a file.

Parameters

f	

7.8.2.22 writeLeapers()

```
void writeLeapers (
            std::ostream & f )
```

7.8.2.23 writePawnAttacks()

```
void writePawnAttacks (
            std::ofstream & f )
```

writes all possible pawn attacks into a file.

Parameters

f	

7.9 main.cpp File Reference

```
#include <iostream>
#include "bitboard.h"
#include "movetables.h"
```

Functions

- int main ()

7.9.1 Function Documentation

7.9.1.1 main()

```
int main ( )
```

7.10 movetables.cpp File Reference

```
#include "movetables.h"
```

Namespaces

- namespace Dagor
- namespace Dagor::MoveTables

Variables

- const BitBoard Dagor::MoveTables::pawnAttacks [2][Board::size]

  The attacks a pawn can make on a given square. Access: `pawnAttacks[color][square]`*, where white is* `0` *and black is* `1`*.*
- const BitBoard Dagor::MoveTables::knightMoves [Board::size]

  The moves a knight can make on a given square.
- const BitBoard Dagor::MoveTables::kingMoves [Board::size]

  The moves a king can make on a given square. For his home square this does not include castling moves.
- const BlockerHash Dagor::MoveTables::bishopHashes [Board::size]

  the hash functions to look up bishop moves, by square.
- const BlockerHash Dagor::MoveTables::rookHashes [Board::size]

  the hash function to look up rook moves, by square.
- const BitBoards::BitBoard Dagor::MoveTables::slidingMoves [ ]

  The move that a sliding piece (bishop, rook or queen) can make on a given square. Access through the hash functions in `bishopHashes` *and* `rookHashes`*.*

7.11 movetables.h File Reference

```
#include <array>
#include "bitboard.h"
#include "board.h"
```

Classes

- class Dagor::MoveTables::BlockerHash

  A hash function that maps a configuration of blocking pieces to an index into the `slidingMoves` table, where the possible moves of a rook or bishop are stored. Both rooks and bishops have one separate hash function for each square.

Namespaces

- namespace Dagor
- namespace Dagor::MoveTables

7.12 movetables.h

Go to the documentation of this file.

```
00001 #ifndef MOVETABLES_H
00002 #define MOVETABLES_H
00003
00004 #include <array>
00005
00006 #include "bitboard.h"
00007 #include "board.h"
00008
00009 namespace Dagor::MoveTables {
00010
00013 extern const BitBoards::BitBoard pawnAttacks[2][Board::size];
00014
00016 extern const BitBoards::BitBoard knightMoves[Board::size];
00017
00020 extern const BitBoards::BitBoard kingMoves[Board::size];
00021
00025 extern const BitBoards::BitBoard slidingMoves[];
00026
00031 class BlockerHash {
00032  public:
00035   const BitBoards::BitBoard blockerMask;
00037   const BitBoards::BitBoard magic;
00039   const unsigned downShift;
00043   const unsigned tableOffset;
00044
00045   BlockerHash(BitBoards::BitBoard mask, BitBoards::BitBoard magic,
00046               unsigned downShift, unsigned tableOffset)
00047       : blockerMask{mask},
00048         magic{magic},
00049         downShift{downShift},
00050         tableOffset{tableOffset} {}
00051
00055   unsigned hash(BitBoards::BitBoard blockers) const {
00056     blockers &= blockerMask;
00057     std::uint64_t h = blockers.as_uint() * magic.as_uint();
00058     return static_cast<unsigned>(h >> downShift) + tableOffset;
00059   }
00060
00066   BitBoards::BitBoard lookUp(BitBoards::BitBoard blockers) const {
00067     return slidingMoves[hash(blockers)];
00068   }
00069 };
00070
00072 extern const BlockerHash bishopHashes[Board::size];
00074 extern const BlockerHash rookHashes[Board::size];
00075 }  // namespace Dagor::MoveTables
00076
00077 #endif
```

7.13 Readme.md File Reference

Index