

Dagor-in-Erain

v0.0.0

Generated by Doxygen 1.10.0

1 Dagor-in-Erain	1
2 Namespace Index	3
2.1 Namespace List	3
3 Class Index	5
3.1 Class List	5
4 File Index	7
4.1 File List	7
5 Namespace Documentation	9
5.1 Dagor Namespace Reference	9
5.2 Dagor::BitBoard Namespace Reference	9
5.2.1 Function Documentation	10
5.2.1.1 operator&()	10
5.2.1.2 operator<<()	10
5.2.1.3 operator==()	10
5.2.1.4 operator" ()	10
5.2.1.5 operator~()	10
5.2.2 Variable Documentation	10
5.2.2.1 edgesOnly	10
5.3 Dagor::Board Namespace Reference	11
5.3.1 Enumeration Type Documentation	11
5.3.1.1 CompassOffsets	11
5.3.1.2 Square	12
5.3.2 Function Documentation	13
5.3.2.1 file()	13
5.3.2.2 file_name()	14
5.3.2.3 index()	14
5.3.2.4 rank()	14
5.3.3 Variable Documentation	15
5.3.3.1 size	15
5.3.3.2 width	15
5.4 Dagor::MoveTables Namespace Reference	15
5.4.1 Variable Documentation	16
5.4.1.1 bishopHashes	16
5.4.1.2 kingMoves	16
5.4.1.3 knightMoves	16
5.4.1.4 pawnAttacks	16
5.4.1.5 rookHashes	16
5.4.1.6 slidingMoves	16
6 Class Documentation	17

6.1 Dagor::BitBoard::BitBoard Class Reference	17
6.1.1 Detailed Description	18
6.1.2 Constructor & Destructor Documentation	18
6.1.2.1 BitBoard() [1/2]	18
6.1.2.2 BitBoard() [2/2]	18
6.1.3 Member Function Documentation	18
6.1.3.1 as_uint()	18
6.1.3.2 findFirstSet()	18
6.1.3.3 is_empty()	19
6.1.3.4 is_set()	19
6.1.3.5 operator&=()	19
6.1.3.6 operator" =()	19
6.1.3.7 popcount()	20
6.1.3.8 set_bit()	20
6.1.3.9 set_bit_if_index_valid()	20
6.1.3.10 single_square_set()	21
6.1.3.11 unset_bit()	21
6.2 Dagor::MoveTables::BlockerHash Class Reference	21
6.2.1 Detailed Description	22
6.2.2 Constructor & Destructor Documentation	22
6.2.2.1 BlockerHash()	22
6.2.3 Member Function Documentation	22
6.2.3.1 hash()	22
6.2.3.2 lookUp()	22
6.2.4 Member Data Documentation	23
6.2.4.1 blockerMask	23
6.2.4.2 downShift	23
6.2.4.3 magic	23
6.2.4.4 tableOffset	23
7 File Documentation	25
7.1 bitboard.cpp File Reference	25
7.2 bitboard.h File Reference	25
7.3 bitboard.h	26
7.4 board.h File Reference	27
7.5 board.h	28
7.6 constants.h File Reference	29
7.6.1 Enumeration Type Documentation	29
7.6.1.1 Color	29
7.7 constants.h	30
7.8 generate_movetables.cpp File Reference	30
7.8.1 Detailed Description	31

7.8.2 Function Documentation	31
7.8.2.1 bishopBlockers()	31
7.8.2.2 bishopMoveRay()	32
7.8.2.3 bishopMoves()	32
7.8.2.4 findPerfectHash()	32
7.8.2.5 generatePossibleBlockers()	33
7.8.2.6 kingMove()	33
7.8.2.7 knightMove()	33
7.8.2.8 main()	34
7.8.2.9 pawnAttack()	34
7.8.2.10 randomFewBitsSet()	34
7.8.2.11 randomLong()	34
7.8.2.12 rookBlockers()	35
7.8.2.13 rookMoveRay()	35
7.8.2.14 rookMoves()	36
7.8.2.15 spreadBitsInMask()	36
7.8.2.16 writeHash()	37
7.8.2.17 writeHashes()	37
7.8.2.18 writeKingMoves()	37
7.8.2.19 writeKnightMoves()	37
7.8.2.20 writePawnAttacks()	39
7.9 main.cpp File Reference	39
7.9.1 Function Documentation	39
7.9.1.1 main()	39
7.10 movetables.cpp File Reference	39
7.11 movetables.h File Reference	40
7.12 movetables.h	40
7.13 print.cpp File Reference	41
7.14 print.h File Reference	41
7.15 print.h	41
7.16 Readme.md File Reference	41
Index	43

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:

Dagor	9
Dagor::BitBoard	9
Dagor::Board	11
Dagor::MoveTables	15

Chapter 3

Class Index

3.1 Class List

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

[Dagor::BitBoard::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

17

[Dagor::MoveTables::BlockerHash](#)

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

21

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

bitboard.cpp	25
bitboard.h	25
board.h	27
constants.h	29
generate_movetables.cpp	30
main.cpp	39
movetables.cpp	39
movetables.h	40
print.cpp	41
print.h	41

Chapter 5

Namespace Documentation

5.1 Dagor Namespace Reference

Namespaces

- namespace [BitBoard](#)
- namespace [Board](#)
- namespace [MoveTables](#)

5.2 Dagor::BitBoard 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)`
- `BitBoard 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&()

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

5.2.1.2 operator<<()

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

5.2.1.3 operator==()

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

5.2.1.4 operator" | ()

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

5.2.1.5 operator~()

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

5.2.2 Variable Documentation

5.2.2.1 edgesOnly

```
const BitBoard Dagor::BitBoard::edgesOnly {0xff8181818181ff} [inline]
```

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

```

8 | @ @ @ @ @ @ @ @
7 | @ . . . . . @
6 | @ . . . . . @
5 | @ . . . . . @
4 | @ . . . . . @
3 | @ . . . . . @
2 | @ . . . . . @
1 | @ @ @ @ @ @ @ @
-----
a b c d e f g h
```

as decimal: 18411139144890810879
as hex: 0xff8181818181ff

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

<i>square</i>	the index of the square.
---------------	--------------------------

Returns

its rank.

5.3.2.2 file_name()

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

Parameters

<i>file</i>	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

<i>file</i>	file (i. e. column)
<i>rank</i>	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

<i>square</i>	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::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::BitBoard](#) `knightMoves` [[Board::size](#)]
The moves a knight can make on a given square.
- const [BitBoard::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 [BitBoard::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 BitBoard::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 BitBoard::BitBoard Dagor::MoveTables::knightMoves
```

The moves a knight can make on a given square.

5.4.1.4 pawnAttacks

```
const BitBoard::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 BitBoard::BitBoard Dagor::MoveTables::slidingMoves [extern]
```

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::BitBoard::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::BitBoard::BitBoard::BitBoard ( )
```

constructs an empty [BitBoard](#).

6.1.2.2 BitBoard() [2/2]

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

Parameters

<i>bitboard</i>	a uint64 as returned by the <code>as_uint</code> function.
-----------------	--

6.1.3 Member Function Documentation

6.1.3.1 as_uint()

```
std::uint64_t Dagor::BitBoard::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::BitBoard::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::BitBoard::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::BitBoard::BitBoard::is_set (
    int square ) const [inline], [constexpr]
```

Checks whether a particular square is set.

Parameters

<i>square</i>	the square to check.
---------------	----------------------

Returns

true, iff the square is set.

6.1.3.5 operator&=()

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

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

Parameters

<i>other</i>	
--------------	--

Returns

this bitboard after the modification

6.1.3.6 operator" |=()

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

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

Parameters

<i>other</i>	
--------------	--

Returns

this bitboard after the modification

6.1.3.7 popcount()

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

Adds the given square to the bitboard.

Parameters

<i>square</i>	the square to add.
---------------	--------------------

6.1.3.9 set_bit_if_index_valid()

```
void Dagor::BitBoard::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

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

6.1.3.10 single_square_set()

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

Constructs a bitboard with only a single square set.

Parameters

<i>square</i>	the square to be set
---------------	----------------------

Returns

the bitboard

6.1.3.11 unset_bit()

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

Removes a given square from the bitboard.

Parameters

<i>square</i>	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](#) ([BitBoard::BitBoard](#) mask, [BitBoard::BitBoard](#) magic, unsigned [downShift](#), unsigned [tableOffset](#))
- unsigned [hash](#) ([BitBoard::BitBoard](#) blockers) const
Computes the hash for a configuration of blocking pieces.
- [BitBoard::BitBoard](#) [lookUp](#) ([BitBoard::BitBoard](#) blockers) const
Looks up the possible moves for a bishop/rook with the specified blocking pieces.

Public Attributes

- const [BitBoard::BitBoard](#) `blockerMask`
The mask singling out the blocking pieces that actually matter to the figure under consideration.
- const [BitBoard::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 (
    BitBoard::BitBoard mask,
    BitBoard::BitBoard magic,
    unsigned downShift,
    unsigned tableOffset ) [inline]
```

6.2.3 Member Function Documentation

6.2.3.1 hash()

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

Computes the hash for a configuration of blocking pieces.

Parameters

<i>blockers</i>	pieces blocking the bishop's/rook's movement.
-----------------	---

Returns

the hash.

6.2.3.2 lookUp()

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

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

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

Parameters

<i>blockers</i>	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 BitBoard::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 BitBoard::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](#)

Chapter 7

File Documentation

7.1 bitboard.cpp File Reference

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

Namespaces

- namespace [Dagor](#)
- namespace [Dagor::BitBoard](#)

Functions

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

7.2 bitboard.h File Reference

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

Classes

- class [Dagor::BitBoard::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::BitBoard](#)

Functions

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

Variables

- `const BitBoard Dagor::BitBoard::edgesOnly {0xff8181818181ff}`
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::BitBoard {
00011
00012 static_assert(sizeof(std::uint64_t) == 8,
00013               "For its BitBoards, this program assumes 64 bit integers.");
00014
00015 class BitBoard {
00016 private:
00017     std::uint64_t board;
00018
00019 public:
00020     BitBoard();
00021     BitBoard(std::uint64_t bitboard);
00022
00023     std::uint64_t as_uint() const { return board; }
00024
00025     BitBoard &operator&=(BitBoard other) {
00026         board &= other.board;
00027         return *this;
00028     }
00029
00030     BitBoard &operator|=(BitBoard other) {
00031         board |= other.board;
00032         return *this;
00033     }
00034
00035     static inline BitBoard single_square_set(int square) {
00036         return BitBoard(static_cast<std::uint64_t>(1) << square);
00037     }
00038
00039     constexpr bool is_empty() const { return board == 0; }
00040
00041     constexpr bool is_set(int square) const {
00042         return (board & (static_cast<std::uint64_t>(1) << square)) != 0;
00043     }
00044
00045     void set_bit(int square) { *this |= single_square_set(square); }
00046
00047     void set_bit_if_index_valid(int file, int rank) {
00048         if (0 <= file && file < Board::width && 0 <= rank && rank < Board::width) {
00049             set_bit(Board::index(file, rank));
00050         }
00051     }
00052
00053     void unset_bit(int square) { board &= ~single_square_set(square).board; }
00054
00055     constexpr int popcount() const { return __builtin_popcountll(board); }
00056
00057     constexpr int findFirstSet() const { return __builtin_ctzll(board); }
00058

```



```

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 BitBoard operator==(BitBoard a, BitBoard b) {
00104     return a.as_uint() == b.as_uint();
00105 }
00106
00107 std::ostream &operator<<(std::ostream &out, const BitBoard &printer);
00108
00122 inline const BitBoard edgesOnly{0xff8181818181ff};
00123
00124 } // namespace Dagor::BitBoard
00125
00126 #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 <fstream>
#include <ios>
#include <iostream>
#include <random>
#include <vector>
#include "bitboard.h"
#include "board.h"
#include "constants.h"
#include "movetables.h"
```

Functions

- [BitBoard::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::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::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::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::BitBoard bishopMoveRay](#) (int square, bool fileUp, bool rankUp, [BitBoard::BitBoard](#) blockers)
Computes one ray of a bishops movement.
- [BitBoard::BitBoard bishopMoves](#) (int square, [BitBoard::BitBoard](#) blockers)
Computes the moves for a bishop.
- [BitBoard::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::BitBoard rookMoveRay](#) (int square, int addFile, int addRank, [BitBoard::BitBoard](#) blockers)
- [BitBoard::BitBoard rookMoves](#) (int square, [BitBoard::BitBoard](#) blockers)

- *Computes the possible move for a rook.*
- `BitBoard::BitBoard spreadBitsInMask` (unsigned bitsToSpread, `BitBoard::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.*
- `std::vector< BitBoard::BitBoard > generatePossibleBlockers` (`BitBoard::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.*
- `MoveTables::BlockerHash findPerfectHash` (`BitBoard::BitBoard` mask)
- *Finds the configuration for a perfect hash function for the powerset of the given mask.*
- `void writeHash` (`std::ofstream` &f, `MoveTables::BlockerHash` &hash, unsigned offset)
- *Write the representation of a hash function to a file.*
- `void writeHashes` (`std::ofstream` &f)
- *Writes the hashes and moves for the leaping pieces (bishop and rook) to a file.*
- `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::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 | . . . . . . . .
-----
a b c d e f g h
```

as decimal: 9592139778506752
as hex: 0x22140014224000

Parameters

<code>square</code>	the position of the bishop.
---------------------	-----------------------------

Returns

a bitboard with the relevant squares set.

7.8.2.2 bishopMoveRay()

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

Computes one ray of a bishops movement.

Parameters

<i>square</i>	the position of the bishop
<i>fileUp</i>	if <code>true</code> , the ray will travel to the right of the board, otherwise to the left.
<i>rankUp</i>	if <code>true</code> , the ray will travel to the top of the board, otherwise to the bottom.
<i>blockers</i>	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::BitBoard bishopMoves (
    int square,
    BitBoard::BitBoard blockers )
```

Computes the moves for a bishop.

Parameters

<i>square</i>	the position of the bishop.
<i>blockers</i>	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 (
    BitBoard::BitBoard mask )
```

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

Parameters

<i>mask</i>	
-------------	--

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()

```
std::vector< BitBoard::BitBoard > generatePossibleBlockers (  
    BitBoard::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

<i>mask</i>	the locations where other pieces could potentially block our movement
-------------	---

Returns

the powerset of that mask.

7.8.2.6 kingMove()

```
BitBoard::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

<i>square</i>	the position of the king.
---------------	---------------------------

Returns

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

7.8.2.7 knightMove()

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

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

Parameters

<i>square</i>	position of the knight
---------------	------------------------

Returns

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

7.8.2.8 main()

```
int main ( )
```

7.8.2.9 pawnAttack()

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

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

Parameters

<i>square</i>	the position of the pawn
<i>color</i>	the color of the pawn (<code>enum Color</code>). White pawns move upwards, black pawns move downwards.

Returns

a bitboard with the attacked squares set.

7.8.2.10 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.

7.8.2.11 randomLong()

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

Generates an uniformly distributed uint64.

Returns

the random number.

7.8.2.12 rookBlockers()

```
BitBoard::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 | . . . . . . . .
-----
a b c d e f g h      as decimal: 2261102847592448
                        as hex:      0x8087608080800
```

Parameters

<i>square</i>	the position of the rook.
---------------	---------------------------

Returns

a bitboard with the relevant squares set.

7.8.2.13 rookMoveRay()

```
BitBoard::BitBoard rookMoveRay (
    int square,
    int addFile,
    int addRank,
    BitBoard::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 left
- (0, +1) = rook goes to the top
- (0, -1) = rook goes to the bottom

Other values should not be used.

Parameters

<i>square</i>	the position of the rook
<i>addFile</i>	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.
<i>addRank</i>	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.
<i>blockers</i>	the pieces blocking the rook's movement.

Returns

a bitboard with the moves of this ray set.

7.8.2.14 rookMoves()

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

Computes the possible move for a rook.

Parameters

<i>square</i>	the position of the rook.
<i>blockers</i>	the pieces blocking the rook's movement.

Returns

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

7.8.2.15 spreadBitsInMask()

```
BitBoard::BitBoard spreadBitsInMask (
    unsigned bitsToSpread,
    BitBoard::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 | . . . . .
7 | @ . . . . .
6 | @ . . . . .
5 | @ . . . . .
4 | @ . . . . .
3 | @ . . . . .
2 | @ . . . . .
1 | . @ @ @ @ @ @ .
-----
   a b c d e f g h

8 | . . . . .
7 | . . . . .
6 | @ . . . . .
5 | . . . . .
4 | @ . . . . .
3 | . . . . .
2 | @ . . . . .
1 | . @ . @ . @ .
-----
   a b c d e f g h
```

Parameters

<i>bitsToSpread</i>	the binary data to fill the mask with.
<i>mask</i>	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.

7.8.2.16 writeHash()

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

Write the representation of a hash function to a file.

Parameters

<i>f</i>	
<i>hash</i>	
<i>offset</i>	a new offset, potentially differing from the one specified in the hash function.

7.8.2.17 writeHashes()

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

Writes the hashes and moves for the leaping pieces (bishop and rook) to a file.

Parameters

<i>f</i>	
----------	--

7.8.2.18 writeKingMoves()

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

writes the king moves to a file.

Parameters

<i>f</i>	
----------	--

7.8.2.19 writeKnightMoves()

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

writes the knight moves to a file.

Parameters

<i>f</i>	
----------	--

7.8.2.20 writePawnAttacks()

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

writes all possible pawn attacks into a file.

Parameters

<i>f</i>	
----------	--

7.9 main.cpp File Reference

```
#include <iostream>
#include "bitboard.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::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::BitBoard Dagor::MoveTables::knightMoves](#) [[Board::size](#)]
The moves a knight can make on a given square.
- const [BitBoard::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 [BitBoard::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 BitBoard::BitBoard pawnAttacks[2][Board::size];
00014
00016 extern const BitBoard::BitBoard knightMoves[Board::size];
00017
00020 extern const BitBoard::BitBoard kingMoves[Board::size];
```

```

00021
00025 extern const BitBoard::BitBoard slidingMoves[];
00026
00031 class BlockerHash {
00032 public:
00035     const BitBoard::BitBoard blockerMask;
00037     const BitBoard::BitBoard magic;
00039     const unsigned downShift;
00043     const unsigned tableOffset;
00044
00045     BlockerHash(BitBoard::BitBoard mask, BitBoard::BitBoard magic,
00046               unsigned downShift, unsigned tableOffset)
00047         : blockerMask{mask},
00048           magic{magic},
00049           downShift{downShift},
00050           tableOffset{tableOffset} {}
00051
00055     unsigned hash(BitBoard::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     BitBoard::BitBoard lookUp(BitBoard::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 print.cpp File Reference

7.14 print.h File Reference

```

#include <array>
#include <string>
#include "bitboard.h"

```

7.15 print.h

[Go to the documentation of this file.](#)

```

00001 #ifndef PRINT_H
00002 #define PRINT_H
00003
00004 #include <array>
00005 #include <string>
00006
00007 #include "bitboard.h"
00008
00009 #endif

```

7.16 Readme.md File Reference

Index

a1
 Dagor::Board, [12](#)
a2
 Dagor::Board, [12](#)
a3
 Dagor::Board, [12](#)
a4
 Dagor::Board, [12](#)
a5
 Dagor::Board, [13](#)
a6
 Dagor::Board, [13](#)
a7
 Dagor::Board, [13](#)
a8
 Dagor::Board, [13](#)
as_uint
 Dagor::BitBoard::BitBoard, [18](#)

b1
 Dagor::Board, [12](#)
b2
 Dagor::Board, [12](#)
b3
 Dagor::Board, [12](#)
b4
 Dagor::Board, [12](#)
b5
 Dagor::Board, [13](#)
b6
 Dagor::Board, [13](#)
b7
 Dagor::Board, [13](#)
b8
 Dagor::Board, [13](#)
bishopBlockers
 generate_movetables.cpp, [31](#)
bishopHashes
 Dagor::MoveTables, [16](#)
bishopMoveRay
 generate_movetables.cpp, [32](#)
bishopMoves
 generate_movetables.cpp, [32](#)
BitBoard
 Dagor::BitBoard::BitBoard, [18](#)
bitboard.cpp, [25](#)
bitboard.h, [25](#)
black
 constants.h, [29](#)
BlockerHash
 Dagor::MoveTables::BlockerHash, [22](#)
blockerMask
 Dagor::MoveTables::BlockerHash, [23](#)
board.h, [27](#)

c1
 Dagor::Board, [12](#)
c2
 Dagor::Board, [12](#)
c3
 Dagor::Board, [12](#)
c4
 Dagor::Board, [12](#)
c5
 Dagor::Board, [13](#)
c6
 Dagor::Board, [13](#)
c7
 Dagor::Board, [13](#)
c8
 Dagor::Board, [13](#)
Color
 constants.h, [29](#)
CompassOffsets
 Dagor::Board, [11](#)
constants.h, [29](#)
 black, [29](#)
 Color, [29](#)
 white, [29](#)

d1
 Dagor::Board, [12](#)
d2
 Dagor::Board, [12](#)
d3
 Dagor::Board, [12](#)
d4
 Dagor::Board, [12](#)
d5
 Dagor::Board, [13](#)
d6
 Dagor::Board, [13](#)
d7
 Dagor::Board, [13](#)
d8
 Dagor::Board, [13](#)
Dagor, [9](#)
Dagor-in-Erain, [1](#)
Dagor::BitBoard, [9](#)
 edgesOnly, [10](#)

- operator<<, 10
- operator==, 10
- operator&, 10
- operator~, 10
- operator|, 10
- Dagor::BitBoard::BitBoard, 17
 - as_uint, 18
 - BitBoard, 18
 - findFirstSet, 18
 - is_empty, 18
 - is_set, 19
 - operator&=, 19
 - operator|=, 19
 - popcount, 20
 - set_bit, 20
 - set_bit_if_index_valid, 20
 - single_square_set, 20
 - unset_bit, 21
- Dagor::Board, 11
 - a1, 12
 - a2, 12
 - a3, 12
 - a4, 12
 - a5, 13
 - a6, 13
 - a7, 13
 - a8, 13
 - b1, 12
 - b2, 12
 - b3, 12
 - b4, 12
 - b5, 13
 - b6, 13
 - b7, 13
 - b8, 13
 - c1, 12
 - c2, 12
 - c3, 12
 - c4, 12
 - c5, 13
 - c6, 13
 - c7, 13
 - c8, 13
 - CompassOffsets, 11
 - d1, 12
 - d2, 12
 - d3, 12
 - d4, 12
 - d5, 13
 - d6, 13
 - d7, 13
 - d8, 13
 - e1, 12
 - e2, 12
 - e3, 12
 - e4, 12
 - e5, 13
 - e6, 13
 - e7, 13
 - e8, 13
 - east, 12
 - f1, 12
 - f2, 12
 - f3, 12
 - f4, 12
 - f5, 13
 - f6, 13
 - f7, 13
 - f8, 13
 - file, 13
 - file_name, 14
 - g1, 12
 - g2, 12
 - g3, 12
 - g4, 12
 - g5, 13
 - g6, 13
 - g7, 13
 - g8, 13
 - h1, 12
 - h2, 12
 - h3, 12
 - h4, 13
 - h5, 13
 - h6, 13
 - h7, 13
 - h8, 13
 - index, 14
 - no_square, 13
 - north, 12
 - north_east, 12
 - north_west, 12
 - rank, 14
 - size, 15
 - south, 12
 - south_east, 12
 - south_west, 12
 - Square, 12
 - west, 12
 - width, 15
- Dagor::MoveTables, 15
 - bishopHashes, 16
 - kingMoves, 16
 - knightMoves, 16
 - pawnAttacks, 16
 - rookHashes, 16
 - slidingMoves, 16
- Dagor::MoveTables::BlockerHash, 21
 - BlockerHash, 22
 - blockerMask, 23
 - downShift, 23
 - hash, 22
 - lookUp, 22
 - magic, 23
 - tableOffset, 23
- downShift

- Dagor::MoveTables::BlockerHash, 23
- e1
 - Dagor::Board, 12
- e2
 - Dagor::Board, 12
- e3
 - Dagor::Board, 12
- e4
 - Dagor::Board, 12
- e5
 - Dagor::Board, 13
- e6
 - Dagor::Board, 13
- e7
 - Dagor::Board, 13
- e8
 - Dagor::Board, 13
- east
 - Dagor::Board, 12
- edgesOnly
 - Dagor::BitBoard, 10
- f1
 - Dagor::Board, 12
- f2
 - Dagor::Board, 12
- f3
 - Dagor::Board, 12
- f4
 - Dagor::Board, 12
- f5
 - Dagor::Board, 13
- f6
 - Dagor::Board, 13
- f7
 - Dagor::Board, 13
- f8
 - Dagor::Board, 13
- file
 - Dagor::Board, 13
- file_name
 - Dagor::Board, 14
- findFirstSet
 - Dagor::BitBoard::BitBoard, 18
- findPerfectHash
 - generate_movetables.cpp, 32
- g1
 - Dagor::Board, 12
- g2
 - Dagor::Board, 12
- g3
 - Dagor::Board, 12
- g4
 - Dagor::Board, 12
- g5
 - Dagor::Board, 13
- g6
 - Dagor::Board, 13
- g7
 - Dagor::Board, 13
- g8
 - Dagor::Board, 13
- generate_movetables.cpp, 30
 - bishopBlockers, 31
 - bishopMoveRay, 32
 - bishopMoves, 32
 - findPerfectHash, 32
 - generatePossibleBlockers, 33
 - kingMove, 33
 - knightMove, 33
 - main, 34
 - pawnAttack, 34
 - randomFewBitsSet, 34
 - randomLong, 34
 - rookBlockers, 34
 - rookMoveRay, 35
 - rookMoves, 36
 - spreadBitsInMask, 36
 - writeHash, 37
 - writeHashes, 37
 - writeKingMoves, 37
 - writeKnightMoves, 37
 - writePawnAttacks, 39
- generatePossibleBlockers
 - generate_movetables.cpp, 33
- h1
 - Dagor::Board, 12
- h2
 - Dagor::Board, 12
- h3
 - Dagor::Board, 12
- h4
 - Dagor::Board, 13
- h5
 - Dagor::Board, 13
- h6
 - Dagor::Board, 13
- h7
 - Dagor::Board, 13
- h8
 - Dagor::Board, 13
- hash
 - Dagor::MoveTables::BlockerHash, 22
- index
 - Dagor::Board, 14
- is_empty
 - Dagor::BitBoard::BitBoard, 18
- is_set
 - Dagor::BitBoard::BitBoard, 19
- kingMove
 - generate_movetables.cpp, 33
- kingMoves
 - Dagor::MoveTables, 16

- knightMove
 - generate_movetables.cpp, 33
- knightMoves
 - Dagor::MoveTables, 16
- lookUp
 - Dagor::MoveTables::BlockerHash, 22
- magic
 - Dagor::MoveTables::BlockerHash, 23
- main
 - generate_movetables.cpp, 34
 - main.cpp, 39
- main.cpp, 39
 - main, 39
- movetables.cpp, 39
- movetables.h, 40
- no_square
 - Dagor::Board, 13
- north
 - Dagor::Board, 12
- north_east
 - Dagor::Board, 12
- north_west
 - Dagor::Board, 12
- operator<<
 - Dagor::BitBoard, 10
- operator==
 - Dagor::BitBoard, 10
- operator&
 - Dagor::BitBoard, 10
- operator&=
 - Dagor::BitBoard::BitBoard, 19
- operator~
 - Dagor::BitBoard, 10
- operator|
 - Dagor::BitBoard, 10
- operator|=
 - Dagor::BitBoard::BitBoard, 19
- pawnAttack
 - generate_movetables.cpp, 34
- pawnAttacks
 - Dagor::MoveTables, 16
- popcount
 - Dagor::BitBoard::BitBoard, 20
- print.cpp, 41
- print.h, 41
- randomFewBitsSet
 - generate_movetables.cpp, 34
- randomLong
 - generate_movetables.cpp, 34
- rank
 - Dagor::Board, 14
- Readme.md, 41
- rookBlockers
 - generate_movetables.cpp, 34
- rookHashes
 - Dagor::MoveTables, 16
- rookMoveRay
 - generate_movetables.cpp, 35
- rookMoves
 - generate_movetables.cpp, 36
- set_bit
 - Dagor::BitBoard::BitBoard, 20
- set_bit_if_index_valid
 - Dagor::BitBoard::BitBoard, 20
- single_square_set
 - Dagor::BitBoard::BitBoard, 20
- size
 - Dagor::Board, 15
- slidingMoves
 - Dagor::MoveTables, 16
- south
 - Dagor::Board, 12
- south_east
 - Dagor::Board, 12
- south_west
 - Dagor::Board, 12
- spreadBitsInMask
 - generate_movetables.cpp, 36
- Square
 - Dagor::Board, 12
- tableOffset
 - Dagor::MoveTables::BlockerHash, 23
- unset_bit
 - Dagor::BitBoard::BitBoard, 21
- west
 - Dagor::Board, 12
- white
 - constants.h, 29
- width
 - Dagor::Board, 15
- writeHash
 - generate_movetables.cpp, 37
- writeHashes
 - generate_movetables.cpp, 37
- writeKingMoves
 - generate_movetables.cpp, 37
- writeKnightMoves
 - generate_movetables.cpp, 37
- writePawnAttacks
 - generate_movetables.cpp, 39