

tAltris
v1.0

Generated by Doxygen 1.8.13

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	3
2.1	File List	3
3	Data Structure Documentation	5
3.1	list Struct Reference	5
3.1.1	Detailed Description	5
3.1.2	Field Documentation	5
3.1.2.1	first	6
3.1.2.2	length	6
3.2	list_node Struct Reference	6
3.2.1	Detailed Description	6
3.2.2	Field Documentation	6
3.2.2.1	next	7
3.3	matrix Struct Reference	7
3.3.1	Detailed Description	7
3.3.2	Field Documentation	7
3.3.2.1	cols	7
3.3.2.2	data	7
3.3.2.3	rows	7

4	File Documentation	9
4.1	src/tAltris.c File Reference	9
4.1.1	Detailed Description	9
4.1.2	Function Documentation	10
4.1.2.1	main()	10
4.2	src/tAltris.h File Reference	10
4.2.1	Detailed Description	10
4.3	src/utlis/list.c File Reference	11
4.3.1	Detailed Description	12
4.3.2	Function Documentation	12
4.3.2.1	list_add()	12
4.3.2.2	list_advance()	12
4.3.2.3	list_append()	13
4.3.2.4	list_at()	14
4.3.2.5	list_concat()	15
4.3.2.6	list_del()	16
4.3.2.7	list_del_after()	16
4.3.2.8	list_del_at()	17
4.3.2.9	list_first()	17
4.3.2.10	list_init()	18
4.3.2.11	list_insert_after()	18
4.3.2.12	list_insert_at()	19
4.3.2.13	list_is_empty()	19
4.3.2.14	list_last()	20
4.3.2.15	list_length()	20
4.3.2.16	list_next()	21
4.3.2.17	list_print()	21
4.3.2.18	list_reverse()	22
4.3.2.19	list_sort()	22
4.3.2.20	list_split_at()	23

4.3.2.21	list_swap()	24
4.4	src/utls/list.h File Reference	24
4.4.1	Detailed Description	26
4.4.2	Macro Definition Documentation	26
4.4.2.1	list_elt	26
4.4.2.2	list_foreach	26
4.4.2.3	list_foreach_elt	27
4.4.2.4	list_foreach_elt_safe	28
4.4.2.5	list_foreach_safe	28
4.4.3	Function Documentation	29
4.4.3.1	list_add()	29
4.4.3.2	list_advance()	30
4.4.3.3	list_append()	30
4.4.3.4	list_at()	31
4.4.3.5	list_concat()	31
4.4.3.6	list_del()	32
4.4.3.7	list_del_after()	32
4.4.3.8	list_del_at()	33
4.4.3.9	list_first()	34
4.4.3.10	list_init()	34
4.4.3.11	list_insert_after()	35
4.4.3.12	list_insert_at()	35
4.4.3.13	list_is_empty()	36
4.4.3.14	list_last()	36
4.4.3.15	list_length()	37
4.4.3.16	list_next()	37
4.4.3.17	list_print()	38
4.4.3.18	list_reverse()	38
4.4.3.19	list_sort()	39
4.4.3.20	list_split_at()	39

4.4.3.21	list_swap()	40
4.5	src/utils/matrix.c File Reference	41
4.5.1	Detailed Description	41
4.5.2	Function Documentation	42
4.5.2.1	matrix_at()	42
4.5.2.2	matrix_cols()	42
4.5.2.3	matrix_copy()	43
4.5.2.4	matrix_create()	43
4.5.2.5	matrix_dot_product()	44
4.5.2.6	matrix_free()	44
4.5.2.7	matrix_product()	45
4.5.2.8	matrix_rows()	46
4.5.2.9	matrix_scale()	46
4.5.2.10	matrix_set()	47
4.5.2.11	matrix_sum()	47
4.5.2.12	matrix_transpose()	48
4.6	src/utils/matrix.h File Reference	49
4.6.1	Detailed Description	50
4.6.2	Function Documentation	50
4.6.2.1	matrix_at()	50
4.6.2.2	matrix_cols()	51
4.6.2.3	matrix_copy()	51
4.6.2.4	matrix_create()	52
4.6.2.5	matrix_dot_product()	52
4.6.2.6	matrix_free()	53
4.6.2.7	matrix_product()	53
4.6.2.8	matrix_rows()	54
4.6.2.9	matrix_scale()	55
4.6.2.10	matrix_set()	55
4.6.2.11	matrix_sum()	56
4.6.2.12	matrix_transpose()	57

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

list	5
list_node	6
matrix	7

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

src/ tAltris.c	
Main file	9
src/ tAltris.h	
Main file	10
src/utls/ list.c	
Intrusive list implement	11
src/utls/ list.h	
Intrusive list implement	24
src/utls/ matrix.c	
Matrix implement	41
src/utls/ matrix.h	
Matrix implement	49

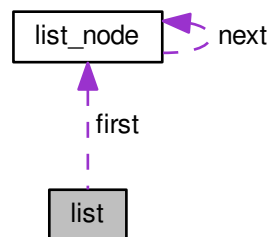
Chapter 3

Data Structure Documentation

3.1 list Struct Reference

```
#include <list.h>
```

Collaboration diagram for list:



Data Fields

- `size_t` **length**
- `struct list_node *` **first**

3.1.1 Detailed Description

Head of a singly-linked list.

3.1.2 Field Documentation

3.1.2.1 first

```
struct list_node* first
```

First node.

3.1.2.2 length

```
size_t length
```

List length.

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

- src/utils/ **list.h**

3.2 list_node Struct Reference

```
#include <list.h>
```

Collaboration diagram for list_node:



Data Fields

- struct **list_node** * **next**

3.2.1 Detailed Description

A node of a singly-linked list.

3.2.2 Field Documentation

3.2.2.1 next

```
struct list_node* next
```

Next node.

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

- src/utils/ **list.h**

3.3 matrix Struct Reference

```
#include <matrix.h>
```

Data Fields

- size_t **rows**
- size_t **cols**
- double * **data**

3.3.1 Detailed Description

Matrix structure

3.3.2 Field Documentation

3.3.2.1 cols

```
size_t cols
```

Columns

3.3.2.2 data

```
double* data
```

Values

3.3.2.3 rows

```
size_t rows
```

Rows

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

- src/utils/ **matrix.h**

Chapter 4

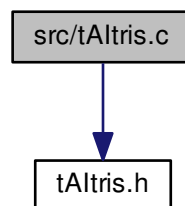
File Documentation

4.1 src/tAltris.c File Reference

Main file.

```
#include "tAltris.h"
```

Include dependency graph for tAltris.c:



Functions

- int **main** ()

4.1.1 Detailed Description

Main file.

Author

S4MasterRace

Version

1.0

4.1.2 Function Documentation

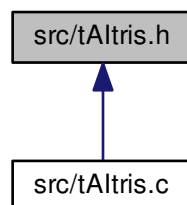
4.1.2.1 main()

```
int main ( )
```

4.2 src/tAltris.h File Reference

Main file.

This graph shows which files directly or indirectly include this file:



4.2.1 Detailed Description

Main file.

Author

S4MasterRace

Version

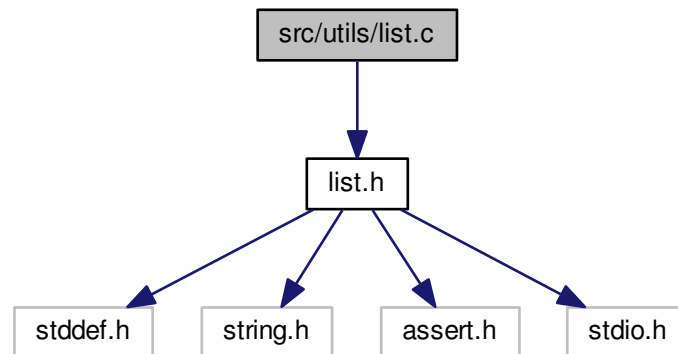
1.0

4.3 src/utls/list.c File Reference

Intrusive list implement.

```
#include "list.h"
```

Include dependency graph for list.c:



Functions

- void **list_init** (struct **list** * **list**)
- size_t **list_length** (const struct **list** * **list**)
- struct **list_node** * **list_first** (const struct **list** * **list**)
- struct **list_node** * **list_last** (const struct **list** * **list**)
- struct **list_node** * **list_next** (const struct **list_node** *node)
- struct **list_node** * **list_advance** (struct **list_node** *node, size_t distance)
- struct **list_node** * **list_at** (const struct **list** * **list**, size_t pos)
- void **list_reverse** (struct **list** * **list**)
- void **list_swap** (struct **list** *l1, struct **list** *l2)
- void **list_split_at** (struct **list** * **list**, size_t pos, struct **list** *right)
- void **list_concat** (struct **list** *l1, struct **list** *l2)
- void **list_sort** (struct **list** * **list**, int(*cmp)(struct **list_node** *, struct **list_node** *))
- int **list_is_empty** (const struct **list** * **list**)
- void **list_add** (struct **list** * **list**, struct **list_node** *node)
- void **list_append** (struct **list** * **list**, struct **list_node** *node)
- void **list_insert_after** (struct **list** * **list**, struct **list_node** *curr, struct **list_node** *node)
- void **list_insert_at** (struct **list** * **list**, struct **list_node** *node, size_t pos)
- void **list_del** (struct **list** * **list**)
- void **list_del_after** (struct **list** * **list**, struct **list_node** *node)
- void **list_del_at** (struct **list** * **list**, size_t pos)
- void **list_print** (const struct **list** * **list**)

4.3.1 Detailed Description

Intrusive list implement.

Author

S4MasterRace

Version

1.0

4.3.2 Function Documentation

4.3.2.1 `list_add()`

```
void list_add (  
    struct list * list,  
    struct list_node * node ) [inline]
```

Adds `node` in the front of `list`

Parameters

<i>list</i>	a list.
<i>node</i>	the new node.

Precondition

`list` must be not NULL.
`node` must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(1)

4.3.2.2 `list_advance()`

```
struct list_node* list_advance (  
    struct list_node * node,  
    size_t distance )
```

Returns the `nth`-node after the current one.

Parameters

<i>node</i>	a node.
<i>distance</i>	distance to move on.

Returns

the nth-node after *node*.

Precondition

node must be not NULL.

Remarks

Complexity: O(n)

4.3.2.3 list_append()

```
void list_append (
    struct list * list,
    struct list_node * node ) [inline]
```

Adds *node* at the end of *list*.

Parameters

<i>list</i>	a list.
<i>node</i>	the new node.

Precondition

list must be not NULL.
node must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(n)

4.3.2.4 list_at()

```
struct list_node* list_at (
    const struct list * list,
    size_t pos )
```

Returns node at the position `pos`.

Parameters

<i>list</i>	a list.
<i>pos</i>	position (0-based) of the node.

Returns

the node at the position `pos`.

Precondition

`list` must be not NULL.
`list` must be not empty.
`pos` must be in `[0; list_length(list)[`.

Remarks

Complexity: O(N)

4.3.2.5 list_concat()

```
void list_concat (
    struct list * l1,
    struct list * l2 ) [inline]
```

Concatenates two lists.

Parameters

<i>l1</i>	list 1.
<i>l2</i>	list 2.

Precondition

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

Postcondition

`l2` is reset to an empty list.

Remarks

Complexity: O(N)

4.3.2.6 list_del()

```
void list_del (
    struct list * list ) [inline]
```

Deletes the first node.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

list must be not NULL.
list must be not empty.

Postcondition

List size decreases by 1.

Remarks

Complexity: O(1)

4.3.2.7 list_del_after()

```
void list_del_after (
    struct list * list,
    struct list_node * node ) [inline]
```

Deletes the node at after the node *curr*.

Parameters

<i>list</i>	a list.
<i>node</i>	a node of <i>list</i> .

Precondition

list must be not NULL.
node must be not NULL.
list must be not empty.
node must a node of *list*.

Postcondition

List size decreases by 1.

Remarks

Complexity: O(1)

4.3.2.8 list_del_at()

```
void list_del_at (
    struct list * list,
    size_t pos ) [inline]
```

Deletes the node at the position *pos*.

Parameters

<i>list</i>	a list.
<i>pos</i>	index (0-based) of the node to delete.

Precondition

list must be not NULL.
list must be not empty.
pos must be in [0; list_length(*list*)].

Postcondition

List size decreases by 1.

Remarks

Complexity: O(n)

4.3.2.9 list_first()

```
struct list_node* list_first (
    const struct list * list )
```

Returns the first node.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the first node.

Precondition

`list` must be not NULL.
`list` must be not empty.

Remarks

Complexity: O(1)

4.3.2.10 list_init()

```
void list_init (
    struct list * list ) [inline]
```

Initializes the list.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

`list` must be not NULL.

Postcondition

`list` is empty.
`list` has a size of 0.

Remarks

Complexity: O(1)

4.3.2.11 list_insert_after()

```
void list_insert_after (
    struct list * list,
    struct list_node * curr,
    struct list_node * node ) [inline]
```

Inserts `node` at after the node `curr`.

Parameters

<i>list</i>	a list.
<i>curr</i>	a node of <code>list</code> .
<i>node</i>	new node.

Precondition

`list` must be not NULL.
`curr` must be not NULL.
`curr` must a node of `list`.
`node` must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(1)

4.3.2.12 list_insert_at()

```
void list_insert_at (
    struct list * list,
    struct list_node * node,
    size_t pos ) [inline]
```

Inserts `node` at the position `pos` in `list`.

Parameters

<i>list</i>	a list.
<i>node</i>	new node.
<i>pos</i>	position (0-based) where to insert the new node.

Precondition

`list` must be not NULL.
`node` must be not NULL.
`pos` must be in `[0; list_length(list)]`.

Postcondition

List size increases by 1.

Remarks

Complexity: O(n)

4.3.2.13 list_is_empty()

```
int list_is_empty (
    const struct list * list ) [inline]
```

Tests if a list is empty.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

1 if the list is empty, otherwise 0.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(1)

4.3.2.14 list_last()

```
struct list_node* list_last (
    const struct list * list )
```

Returns the last node.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the last node.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(N)

4.3.2.15 list_length()

```
size_t list_length (
    const struct list * list ) [inline]
```

Returns the size of the list.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the length of the list.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(1)

4.3.2.16 list_next()

```
struct list_node* list_next (  
    const struct list_node * node )
```

Returns the next node.

Parameters

<i>node</i>	a node.
-------------	---------

Returns

the next node.

Precondition

`node` must be not NULL.

Remarks

Complexity: O(1)

4.3.2.17 list_print()

```
void list_print (  
    const struct list * list )
```

Print the list

Parameters

<i>list</i>	a list
-------------	--------

4.3.2.18 list_reverse()

```
void list_reverse (
    struct list * list ) [inline]
```

Reverses the order of the elements in the list.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

list must be not NULL.

Remarks

Complexity: O(N)

4.3.2.19 list_sort()

```
void list_sort (
    struct list * list,
    int (*) (struct list_node *, struct list_node *) cmp ) [inline]
```

Sort a list using a comparison function.

The contents of the list are sorted in ascending order according to a comparison function which is called with two arguments that point to the node being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members compare as equal, their order in the sorted list is preserved.

Parameters

<i>list</i>	list to sort.
<i>cmp</i>	comparison function to use.

Precondition

`list` must be not NULL.
`cmp` must be not NULL.

Remarks

The sort is stable.
 Complexity: $O(N \log N)$
 Space complexity: $O(1)$

4.3.2.20 list_split_at()

```
void list_split_at (
    struct list * list,
    size_t pos,
    struct list * right ) [inline]
```

Splits a list in two parts at the position `pos`.

After the split:

- `list` contains nodes in `[0, pos[`
- `right` contains nodes in `[pos,length(list)[`

Examples:

```
list = [1, 2, 3]
list_split_at(list, 0, right) => ([], [1,2,3])
list_split_at(list, 1, right) => ([1], [2,3])
list_split_at(list, 2, right) => ([1,2], [3])
list_split_at(list, 3, right) => ([1,2,3], [])
list = []
list_split_at(list, 0, right) => ([], [])
```

Parameters

<i>list</i>	list to split.
<i>pos</i>	position (0-based) where to split the list.
<i>right</i>	an empty list to receive the part after <code>pos</code>

Precondition

`list` must be not NULL.
`right` must be not NULL.
`right` must be empty.
`list` must be different of `right`.

Remarks

Complexity: $O(N)$

4.3.2.21 list_swap()

```
void list_swap (
    struct list * l1,
    struct list * l2 ) [inline]
```

Swaps two lists.

Parameters

<i>l1</i>	list 1.
<i>l2</i>	list 2.

Precondition

l1 must be not NULL.
l2 must be not NULL.
l1 must be different of *l2*.

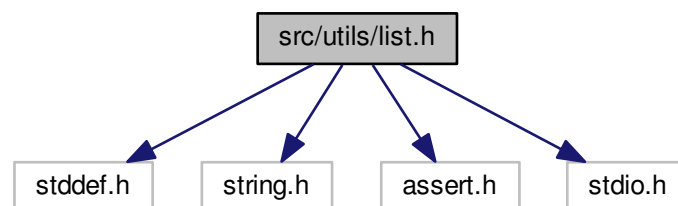
Remarks

Complexity: $O(1)$

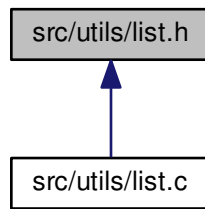
4.4 src/utils/list.h File Reference

Intrusive list implement.

```
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <stdio.h>
Include dependency graph for list.h:
```



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **list**
- struct **list_node**

Macros

- #define **list_elt**(node, type, fieldname) ((type*)((char*)(node) - offsetof(type, fieldname)))
- #define **list_foreach**(list, curr) for (curr = **list_first**(list); curr != NULL; curr = **list_next**(curr))
- #define **list_foreach_elt**(list, curr, type, fieldname)
- #define **list_foreach_safe**(list, curr, tmp)
- #define **list_foreach_elt_safe**(list, curr, tmp, type, fieldname)

Functions

- void **list_init** (struct **list** * list)
- size_t **list_length** (const struct **list** * list)
- struct **list_node** * **list_first** (const struct **list** * list)
- struct **list_node** * **list_last** (const struct **list** * list)
- struct **list_node** * **list_next** (const struct **list_node** * node)
- struct **list_node** * **list_advance** (struct **list_node** * node, size_t distance)
- struct **list_node** * **list_at** (const struct **list** * list, size_t pos)
- void **list_reverse** (struct **list** * list)
- void **list_swap** (struct **list** *l1, struct **list** *l2)
- void **list_split_at** (struct **list** * list, size_t pos, struct **list** *right)
- void **list_concat** (struct **list** *l1, struct **list** *l2)
- void **list_sort** (struct **list** * list, int(*cmp)(struct **list_node** *, struct **list_node** *))
- int **list_is_empty** (const struct **list** * list)
- void **list_add** (struct **list** * list, struct **list_node** * node)
- void **list_append** (struct **list** * list, struct **list_node** * node)
- void **list_insert_after** (struct **list** * list, struct **list_node** * curr, struct **list_node** * node)
- void **list_insert_at** (struct **list** * list, struct **list_node** * node, size_t pos)
- void **list_del** (struct **list** * list)
- void **list_del_after** (struct **list** * list, struct **list_node** * node)
- void **list_del_at** (struct **list** * list, size_t pos)
- void **list_print** (const struct **list** * list)

4.4.1 Detailed Description

Intrusive list implement.

Author

S4MasterRace

Version

1.0

4.4.2 Macro Definition Documentation

4.4.2.1 list_elt

```
#define list_elt(  
    node,  
    type,  
    fieldname ) ((type*)((char*)(node) - offsetof(type, fieldname)))
```

Returns a pointer to the structure which contains the node.

Parameters

<i>node</i>	a list node (struct list_node*).
<i>type</i>	type of the structure which contains the node.
<i>fieldname</i>	name of the node (field name) in the structure.

Precondition

node must be not NULL.

Remarks

Complexity: O(1)

4.4.2.2 list_foreach

```
#define list_foreach(  
    list,  
    curr ) for (curr = list_first( list); curr != NULL; curr = list_next(curr))
```

Iterates over list (nodes).

Parameters

<i>list</i>	a list (struct list*).
<i>curr</i>	a struct list_node* used to hold the current element.

Precondition

`list` must be not NULL.
`curr` must be not NULL.

Remarks

Complexity: O(N)

4.4.2.3 list_foreach_elt

```
#define list_foreach_elt(
    list,
    curr,
    type,
    fieldname )
```

Value:

```
for (curr = list_elt(list_first(list), type, fieldname); \
     curr != NULL; \
     curr = curr->fieldname.next == NULL ? NULL : \
     list_elt(list_next(&(curr->fieldname)), type, fieldname))
```

Iterates over list (elements)

Parameters

<i>list</i>	a list (struct list*).
<i>curr</i>	pointer (type*) used to hold the current element.
<i>type</i>	type of the structure which contains the node.
<i>fieldname</i>	name of the node (field name) in the structure.

Precondition

`list` must be not NULL.
`list` must be not empty.
`curr` must be not NULL.

Remarks

Complexity: O(N)

4.4.2.4 list_foreach_elt_safe

```
#define list_foreach_elt_safe(  
    list,  
    curr,  
    tmp,  
    type,  
    fieldname )
```

Value:

```
for (curr = list_elt(list_first(list), type, fieldname), \
    tmp = list_next(&(curr->fieldname)); \
    curr != NULL; \
    curr = tmp == NULL ? NULL : list_elt(tmp, type, fieldname), \
    tmp = tmp == NULL ? NULL : list_next(tmp))
```

Iterates over list (elements), allows deletion of the current element.

Parameters

<i>list</i>	a list (struct list*).
<i>curr</i>	pointer (type*) used to hold the current element.
<i>tmp</i>	a struct list_node* used as temporary storage.
<i>type</i>	type of the structure which contains the node.
<i>fieldname</i>	name of the node (field name) in the structure.

Precondition

```
list must be not NULL.  
list must be not empty.  
curr must be not NULL.
```

Remarks

Complexity: O(N)

4.4.2.5 list_foreach_safe

```
#define list_foreach_safe(  
    list,  
    curr,  
    tmp )
```

Value:

```
for (curr = list_first(list), tmp = list_next(curr); \
    curr != NULL; \
    curr = tmp, tmp = tmp == NULL ? NULL : list_next(tmp))
```

Iterates over list (nodes), allows deletion of the current node.

Parameters

<i>list</i>	a list (struct list*).
<i>curr</i>	a struct list_node* used to hold the current element.
<i>tmp</i>	a struct list_node* used as temporary storage.

Precondition

`list` must be not NULL.
`curr` must be not NULL.
`tmp` must be not NULL.

Remarks

Complexity: O(N)

4.4.3 Function Documentation**4.4.3.1 list_add()**

```
void list_add (  
    struct list * list,  
    struct list_node * node ) [inline]
```

Adds `node` in the front of `list`

Parameters

<i>list</i>	a list.
<i>node</i>	the new node.

Precondition

`list` must be not NULL.
`node` must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(1)

4.4.3.2 list_advance()

```
struct list_node* list_advance (
    struct list_node * node,
    size_t distance )
```

Returns the nth-node after the current one.

Parameters

<i>node</i>	a node.
<i>distance</i>	distance to move on.

Returns

the nth-node after *node*.

Precondition

node must be not NULL.

Remarks

Complexity: O(n)

4.4.3.3 list_append()

```
void list_append (
    struct list * list,
    struct list_node * node ) [inline]
```

Adds *node* at the end of *list*.

Parameters

<i>list</i>	a list.
<i>node</i>	the new node.

Precondition

list must be not NULL.
node must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(n)

4.4.3.4 list_at()

```
struct list_node* list_at (
    const struct list * list,
    size_t pos )
```

Returns node at the position *pos*.

Parameters

<i>list</i>	a list.
<i>pos</i>	position (0-based) of the node.

Returns

the node at the position *pos*.

Precondition

list must be not NULL.
list must be not empty.
pos must be in [0; list_length(list)].

Remarks

Complexity: O(N)

4.4.3.5 list_concat()

```
void list_concat (
    struct list * l1,
    struct list * l2 ) [inline]
```

Concatenates two lists.

Parameters

<i>l1</i>	list 1.
<i>l2</i>	list 2.

Precondition

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

Postcondition

`l2` is reset to an empty list.

Remarks

Complexity: O(N)

4.4.3.6 list_del()

```
void list_del (
    struct list * list ) [inline]
```

Deletes the first node.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

`list` must be not NULL.
`list` must be not empty.

Postcondition

List size decreases by 1.

Remarks

Complexity: O(1)

4.4.3.7 list_del_after()

```
void list_del_after (
    struct list * list,
    struct list_node * node ) [inline]
```

Deletes the node at after the node `curr`.

Parameters

<i>list</i>	a list.
<i>node</i>	a node of <code>list</code> .

Precondition

`list` must be not NULL.
`node` must be not NULL.
`list` must be not empty.
`node` must a node of `list`.

Postcondition

List size decreases by 1.

Remarks

Complexity: O(1)

4.4.3.8 list_del_at()

```
void list_del_at (  
    struct list * list,  
    size_t pos ) [inline]
```

Deletes the node at the position `pos`.

Parameters

<i>list</i>	a list.
<i>pos</i>	index (0-based) of the node to delete.

Precondition

`list` must be not NULL.
`list` must be not empty.
`pos` must be in `[0; list_length(list)[`.

Postcondition

List size decreases by 1.

Remarks

Complexity: O(n)

4.4.3.9 list_first()

```
struct list_node* list_first (
    const struct list * list )
```

Returns the first node.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the first node.

Precondition

`list` must be not NULL.
`list` must be not empty.

Remarks

Complexity: O(1)

4.4.3.10 list_init()

```
void list_init (
    struct list * list ) [inline]
```

Initializes the list.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

`list` must be not NULL.

Postcondition

`list` is empty.
`list` has a size of 0.

Remarks

Complexity: O(1)

4.4.3.11 list_insert_after()

```
void list_insert_after (
    struct list * list,
    struct list_node * curr,
    struct list_node * node ) [inline]
```

Inserts *node* at after the node *curr*.

Parameters

<i>list</i>	a list.
<i>curr</i>	a node of <i>list</i> .
<i>node</i>	new node.

Precondition

list must be not NULL.
curr must be not NULL.
curr must a node of *list*.
node must be not NULL.

Postcondition

List size increases by 1.

Remarks

Complexity: O(1)

4.4.3.12 list_insert_at()

```
void list_insert_at (
    struct list * list,
    struct list_node * node,
    size_t pos ) [inline]
```

Inserts *node* at the position *pos* in *list*.

Parameters

<i>list</i>	a list.
<i>node</i>	new node.
<i>pos</i>	position (0-based) where to insert the new node.

Precondition

`list` must be not NULL.
`node` must be not NULL.
`pos` must be in `[0; list_length(list)]`.

Postcondition

List size increases by 1.

Remarks

Complexity: $O(n)$

4.4.3.13 `list_is_empty()`

```
int list_is_empty (
    const struct list * list ) [inline]
```

Tests if a list is empty.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

1 if the list is empty, otherwise 0.

Precondition

`list` must be not NULL.

Remarks

Complexity: $O(1)$

4.4.3.14 `list_last()`

```
struct list_node* list_last (
    const struct list * list )
```

Returns the last node.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the last node.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(N)

4.4.3.15 list_length()

```
size_t list_length (  
    const struct list * list ) [inline]
```

Returns the size of the list.

Parameters

<i>list</i>	a list.
-------------	---------

Returns

the length of the list.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(1)

4.4.3.16 list_next()

```
struct list_node* list_next (  
    const struct list_node * node )
```

Returns the next node.

Parameters

<i>node</i>	a node.
-------------	---------

Returns

the next node.

Precondition

node must be not NULL.

Remarks

Complexity: O(1)

4.4.3.17 list_print()

```
void list_print (
    const struct list * list )
```

Print the list

Parameters

<i>list</i>	a list
-------------	--------

4.4.3.18 list_reverse()

```
void list_reverse (
    struct list * list ) [inline]
```

Reverses the order of the elements in the list.

Parameters

<i>list</i>	a list.
-------------	---------

Precondition

list must be not NULL.

Remarks

Complexity: $O(N)$

4.4.3.19 list_sort()

```
void list_sort (
    struct list * list,
    int(*) (struct list_node *, struct list_node *) cmp ) [inline]
```

Sort a list using a comparison function.

The contents of the list are sorted in ascending order according to a comparison function which is called with two arguments that point to the node being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members compare as equal, their order in the sorted list is preserved.

Parameters

<i>list</i>	list to sort.
<i>cmp</i>	comparison function to use.

Precondition

list must be not NULL.
cmp must be not NULL.

Remarks

The sort is stable.
Complexity: $O(N \log N)$
Space complexity: $O(1)$

4.4.3.20 list_split_at()

```
void list_split_at (
    struct list * list,
    size_t pos,
    struct list * right ) [inline]
```

Splits a list in two parts at the position *pos*.

After the split:

- `list` contains nodes in `[0, pos[`
- `right` contains nodes in `[pos,length(list)[`

Examples:

```
list = [1, 2, 3]
list_split_at(list, 0, right) => ([], [1,2,3])
list_split_at(list, 1, right) => ([1], [2,3])
list_split_at(list, 2, right) => ([1,2], [3])
list_split_at(list, 3, right) => ([1,2,3], [])
list = []
list_split_at(list, 0, right) => ([], [])
```

Parameters

<i>list</i>	list to split.
<i>pos</i>	position (0-based) where to split the list.
<i>right</i>	an empty list to receive the part after <code>pos</code>

Precondition

`list` must be not NULL.
`right` must be not NULL.
`right` must be empty.
`list` must be different of `right`.

Remarks

Complexity: O(N)

4.4.3.21 list_swap()

```
void list_swap (
    struct list * l1,
    struct list * l2 ) [inline]
```

Swaps two lists.

Parameters

<i>l1</i>	list 1.
<i>l2</i>	list 2.

Precondition

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

Remarks

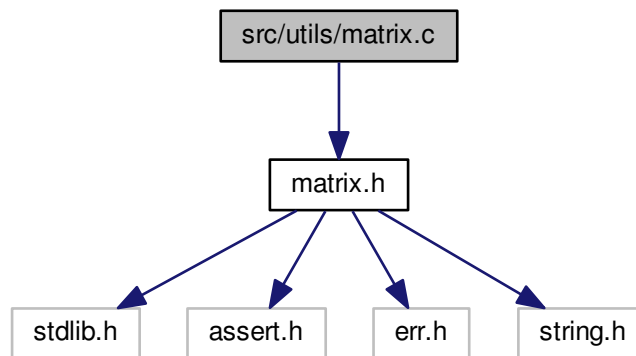
Complexity: $O(1)$

4.5 src/utls/matrix.c File Reference

Matrix implement.

```
#include "matrix.h"
```

Include dependency graph for matrix.c:



Functions

- `struct matrix * matrix_create` (`size_t` rows, `size_t` cols)
- `void matrix_free` (`struct matrix *mx`)
- `size_t matrix_rows` (`const struct matrix *mx`)
- `size_t matrix_cols` (`const struct matrix *mx`)
- `double matrix_at` (`const struct matrix *mx`, `size_t` rows, `size_t` cols)
- `void matrix_set` (`struct matrix *mx`, `size_t` rows, `size_t` cols, `double` value)
- `struct matrix * matrix_copy` (`const struct matrix *mx`)
- `void matrix_transpose` (`const struct matrix *mx`, `struct matrix *tmx`)
- `void matrix_sum` (`const struct matrix *mx1`, `const struct matrix *mx2`, `struct matrix *sum`)
- `void matrix_product` (`const struct matrix *mx1`, `const struct matrix *mx2`, `struct matrix *prod`)
- `void matrix_scale` (`const struct matrix *mx`, `double` scale, `struct matrix *smx`)
- `double matrix_dot_product` (`const struct matrix *v1`, `const struct matrix *v2`)

4.5.1 Detailed Description

Matrix implement.

Author

S4MasterRace

Version

1.0

4.5.2 Function Documentation

4.5.2.1 `matrix_at()`

```
double matrix_at (
    const struct matrix * mx,
    size_t rows,
    size_t cols ) [inline]
```

Get value at `rows` rows and `cols` columns of `mx`

Parameters

<i>mx</i>	a matrix
<i>rows</i>	rows
<i>cols</i>	columns

Returns

the value at `rows` rows and `cols` columns of `mx`

Precondition

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx) [`
`cols` must be between `[0, matrix_cols(mx) [`

Remarks

Complexity: O(1)

4.5.2.2 `matrix_cols()`

```
size_t matrix_cols (
    const struct matrix * mx ) [inline]
```

Get the number of columns of `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the number of columns `mx`

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.5.2.3 matrix_copy()

```
struct matrix* matrix_copy (  
    const struct matrix * mx )
```

Copy the matrix `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the copy of `mx`

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.5.2.4 matrix_create()

```
struct matrix* matrix_create (  
    size_t rows,  
    size_t cols )
```

Create a matrix of size `rows` rows and `cols` columns

Parameters

<i>rows</i>	number of rows
<i>cols</i>	number of columns

Returns

the initialized matrix of size `rows` rows and `cols` columns

Precondition

`rows` must be greater than zero

`cols` must be greater than zero

Remarks

Complexity: $O(1)$

4.5.2.5 matrix_dot_product()

```
double matrix_dot_product (
    const struct matrix * v1,
    const struct matrix * v2 )
```

Do the dot product of vector `v1` with vector `v2`

Parameters

<code>v1</code>	a vector
<code>v2</code>	a vector

Returns

the dot product of vector `v1` with vector `v2`

Precondition

`v1` must be not NULL

`v2` must be not NULL

`matrix_cols(v1)` and `matrix_cols(v2)` must be equal to one

`matrix_rows(v1)` must be equal to `matrix_rows(v2)`

Remarks

Complexity: $O(N)$

4.5.2.6 matrix_free()

```
void matrix_free (
    struct matrix * mx ) [inline]
```

Free the matrix `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Precondition

mx must be not NULL

Postcondition

mx is freed

Remarks

Complexity: O(1)

4.5.2.7 matrix_product()

```
void matrix_product (
    const struct matrix * mx1,
    const struct matrix * mx2,
    struct matrix * prod )
```

Multiply the matrix *mx1* with *mx2*

Parameters

<i>mx1</i>	a matrix
<i>mx2</i>	a matrix
<i>prod</i>	a matrix

Precondition

mx1 must be not NULL
mx2 must be not NULL
prod must be not NULL
prod must be not equal to *mx1*
prod must be not equal to *mx2*
matrix_cols(*mx1*) must be equal to *matrix_rows*(*mx2*)
matrix_rows(*prod*) must be equal to *matrix_rows*(*mx1*)
matrix_cols(*prod*) must be equal to *matrix_cols*(*mx2*)

Postcondition

prod is the product of *mx1* with *mx2*

Remarks

Complexity: O(nmp)

4.5.2.8 `matrix_rows()`

```
size_t matrix_rows (
    const struct matrix * mx ) [inline]
```

Get the number of rows `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the number of rows of `mx`

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.5.2.9 `matrix_scale()`

```
void matrix_scale (
    const struct matrix * mx,
    double scale,
    struct matrix * smx )
```

Scale the matrix `mx` with `scale`

Parameters

<i>mx</i>	a matrix
<i>scale</i>	the scale factor
<i>smx</i>	a matrix

Precondition

`mx` must be not NULL

`smx` must be not NULL

`matrix_rows (smx)` must be equal to `matrix_rows (mx)`

`matrix_cols (smx)` must be equal to `matrix_cols (mx)`

Postcondition

`smx` is the `scale` scaled matrix of `mx`

Remarks

Complexity: O(N)

4.5.2.10 matrix_set()

```
void matrix_set (
    struct matrix * mx,
    size_t rows,
    size_t cols,
    double value ) [inline]
```

Set the value at rows *rows* and cols columns with value of *mx*

Parameters

<i>mx</i>	a matrix
<i>rows</i>	rows
<i>cols</i>	columns
<i>value</i>	a value

Precondition

mx must be not NULL
rows must be between [0, matrix_rows(*mx*) [
cols must be between [0, matrix_cols(*mx*) [

Postcondition

the value at rows *rows* and cols columns is *value*

Remarks

Complexity: O(1)

4.5.2.11 matrix_sum()

```
void matrix_sum (
    const struct matrix * mx1,
    const struct matrix * mx2,
    struct matrix * sum )
```

Sum the matrix *mx1* with *mx2*

Parameters

<i>mx1</i>	a matrix
<i>mx2</i>	a matrix
<i>sum</i>	a matrix

Precondition

`mx1` must be not NULL
`mx2` must be not NULL
`sum` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(sum)` must be equal to `matrix_rows(mx1)`
`matrix_cols(sum)` must be equal to `matrix_cols(mx1)`

Postcondition

`sum` is the sum of matrix `mx1` with `mx2`

Remarks

Complexity: $O(N)$

4.5.2.12 matrix_transpose()

```
void matrix_transpose (
    const struct matrix * mx,
    struct matrix * tmx )
```

Transpose the matrix `mx`

Parameters

<i>mx</i>	a matrix
<i>tmx</i>	a matrix

Precondition

`mx` must be not NULL
`tmx` must be not NULL
`tmx` must be not equal to `mx`
`matrix_rows(tmx)` must be equal to `matrix_cols(mx)`
`matrix_cols(tmx)` must be equal to `matrix_rows(mx)`

Postcondition

`tmx` is the transposed matrix of `mx`

Remarks

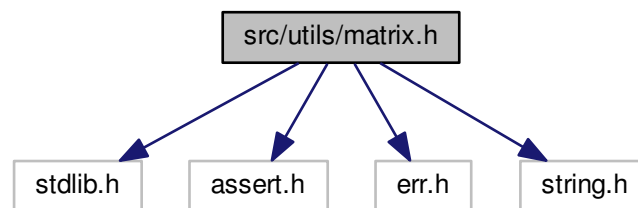
Complexity: $O(N)$

4.6 src/utils/matrix.h File Reference

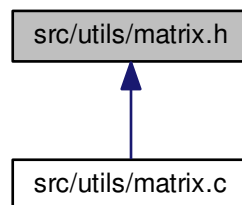
Matrix implement.

```
#include <stdlib.h>
#include <assert.h>
#include <err.h>
#include <string.h>
```

Include dependency graph for matrix.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **matrix**

Functions

- struct **matrix** * **matrix_create** (size_t rows, size_t cols)
- void **matrix_free** (struct **matrix** *mx)
- size_t **matrix_rows** (const struct **matrix** *mx)
- size_t **matrix_cols** (const struct **matrix** *mx)
- double **matrix_at** (const struct **matrix** *mx, size_t rows, size_t cols)

- void **matrix_set** (struct **matrix** *mx, size_t rows, size_t cols, double value)
- struct **matrix** * **matrix_copy** (const struct **matrix** *mx)
- void **matrix_transpose** (const struct **matrix** *mx, struct **matrix** *tmx)
- void **matrix_sum** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *sum)
- void **matrix_product** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *prod)
- void **matrix_scale** (const struct **matrix** *mx, double scale, struct **matrix** *smx)
- double **matrix_dot_product** (const struct **matrix** *v1, const struct **matrix** *v2)

4.6.1 Detailed Description

Matrix implement.

Author

S4MasterRace

Version

1.0

4.6.2 Function Documentation

4.6.2.1 matrix_at()

```
double matrix_at (
    const struct matrix * mx,
    size_t rows,
    size_t cols ) [inline]
```

Get value at `rows` rows and `cols` columns of `mx`

Parameters

<i>mx</i>	a matrix
<i>rows</i>	rows
<i>cols</i>	columns

Returns

the value at `rows` rows and `cols` columns of `mx`

Precondition

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx) [`
`cols` must be between `[0, matrix_cols(mx) [`

Remarks

Complexity: O(1)

4.6.2.2 matrix_cols()

```
size_t matrix_cols (  
    const struct matrix * mx ) [inline]
```

Get the number of columns of *mx*

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the number of columns *mx*

Precondition

mx must be not NULL

Remarks

Complexity: O(1)

4.6.2.3 matrix_copy()

```
struct matrix* matrix_copy (  
    const struct matrix * mx )
```

Copy the matrix *mx*

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the copy of *mx*

Precondition

mx must be not NULL

Remarks

Complexity: O(1)

4.6.2.4 matrix_create()

```
struct matrix* matrix_create (
    size_t rows,
    size_t cols )
```

Create a matrix of size `rows` rows and `cols` columns

Parameters

<i>rows</i>	number of rows
<i>cols</i>	number of columns

Returns

the initialized matrix of size `rows` rows and `cols` columns

Precondition

`rows` must be greater than zero
`cols` must be greater than zero

Remarks

Complexity: O(1)

4.6.2.5 matrix_dot_product()

```
double matrix_dot_product (
    const struct matrix * v1,
    const struct matrix * v2 )
```

Do the dot product of vector `v1` with vector `v2`

Parameters

<i>v1</i>	a vector
<i>v2</i>	a vector

Returns

the dot product of vector `v1` with vector `v2`

Precondition

`v1` must be not NULL

`v2` must be not NULL

`matrix_cols(v1)` and `matrix_cols(v2)` must be equal to one

`matrix_rows(v1)` must be equal to `matrix_rows(v2)`

Remarks

Complexity: $O(N)$

4.6.2.6 matrix_free()

```
void matrix_free (
    struct matrix * mx ) [inline]
```

Free the matrix `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Precondition

`mx` must be not NULL

Postcondition

`mx` is freed

Remarks

Complexity: $O(1)$

4.6.2.7 matrix_product()

```
void matrix_product (
    const struct matrix * mx1,
    const struct matrix * mx2,
    struct matrix * prod )
```

Multiply the matrix `mx1` with `mx2`

Parameters

<i>mx1</i>	a matrix
<i>mx2</i>	a matrix
<i>prod</i>	a matrix

Precondition

`mx1` must be not NULL
`mx2` must be not NULL
`prod` must be not NULL
`prod` must be not equal to `mx1`
`prod` must be not equal to `mx2`
`matrix_cols(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_rows(prod)` must be equal to `matrix_rows(mx1)`
`matrix_cols(prod)` must be equal to `matrix_cols(mx2)`

Postcondition

`prod` is the product of `mx1` with `mx2`

Remarks

Complexity: O(nmp)

4.6.2.8 matrix_rows()

```
size_t matrix_rows (
    const struct matrix * mx ) [inline]
```

Get the number of rows `mx`

Parameters

<i>mx</i>	a matrix
-----------	----------

Returns

the number of rows of `mx`

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.6.2.9 matrix_scale()

```
void matrix_scale (
    const struct matrix * mx,
    double scale,
    struct matrix * smx )
```

Scale the matrix *mx* with *scale*

Parameters

<i>mx</i>	a matrix
<i>scale</i>	the scale factor
<i>smx</i>	a matrix

Precondition

mx must be not NULL
smx must be not NULL
matrix_rows(*smx*) must be equal to *matrix_rows*(*mx*)
matrix_cols(*smx*) must be equal to *matrix_cols*(*mx*)

Postcondition

smx is the *scale* scaled matrix of *mx*

Remarks

Complexity: O(N)

4.6.2.10 matrix_set()

```
void matrix_set (
    struct matrix * mx,
    size_t rows,
    size_t cols,
    double value ) [inline]
```

Set the value at *rows* rows and *cols* columns with *value* of *mx*

Parameters

<i>mx</i>	a matrix
<i>rows</i>	rows
<i>cols</i>	columns
<i>value</i>	a value

Precondition

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx) [`
`cols` must be between `[0, matrix_cols(mx) [`

Postcondition

the value at `rows` `rows` and `cols` columns is `value`

Remarks

Complexity: O(1)

4.6.2.11 matrix_sum()

```
void matrix_sum (
    const struct matrix * mx1,
    const struct matrix * mx2,
    struct matrix * sum )
```

Sum the matrix `mx1` with `mx2`

Parameters

<i>mx1</i>	a matrix
<i>mx2</i>	a matrix
<i>sum</i>	a matrix

Precondition

`mx1` must be not NULL
`mx2` must be not NULL
`sum` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(sum)` must be equal to `matrix_rows(mx1)`
`matrix_cols(sum)` must be equal to `matrix_cols(mx1)`

Postcondition

`sum` is the sum of matrix `mx1` with `mx2`

Remarks

Complexity: O(N)

4.6.2.12 matrix_transpose()

```
void matrix_transpose (
    const struct matrix * mx,
    struct matrix * tmx )
```

Transpose the matrix *mx*

Parameters

<i>mx</i>	a matrix
<i>tmx</i>	a matrix

Precondition

mx must be not NULL
tmx must be not NULL
tmx must be not equal to *mx*
`matrix_rows(tmx)` must be equal to `matrix_cols(mx)`
`matrix_cols(tmx)` must be equal to `matrix_rows(mx)`

Postcondition

tmx is the transposed matrix of *mx*

Remarks

Complexity: O(N)

Index

- cols
 - matrix, 7
- data
 - matrix, 7
- first
 - list, 5
- length
 - list, 6
- list, 5
 - first, 5
 - length, 6
- list.c
 - list_add, 12
 - list_advance, 12
 - list_append, 13
 - list_at, 13
 - list_concat, 15
 - list_del, 15
 - list_del_after, 16
 - list_del_at, 17
 - list_first, 17
 - list_init, 18
 - list_insert_after, 18
 - list_insert_at, 19
 - list_is_empty, 19
 - list_last, 20
 - list_length, 20
 - list_next, 21
 - list_print, 21
 - list_reverse, 22
 - list_sort, 22
 - list_split_at, 23
 - list_swap, 24
- list.h
 - list_add, 29
 - list_advance, 29
 - list_append, 30
 - list_at, 31
 - list_concat, 31
 - list_del, 32
 - list_del_after, 32
 - list_del_at, 33
 - list_elt, 26
 - list_first, 33
 - list_foreach, 26
 - list_foreach_elt, 27
 - list_foreach_elt_safe, 27
 - list_foreach_safe, 28
 - list_init, 34
 - list_insert_after, 34
 - list_insert_at, 35
 - list_is_empty, 36
 - list_last, 36
 - list_length, 37
 - list_next, 37
 - list_print, 38
 - list_reverse, 38
 - list_sort, 39
 - list_split_at, 39
 - list_swap, 40
- list_add
 - list.c, 12
 - list.h, 29
- list_advance
 - list.c, 12
 - list.h, 29
- list_append
 - list.c, 13
 - list.h, 30
- list_at
 - list.c, 13
 - list.h, 31
- list_concat
 - list.c, 15
 - list.h, 31
- list_del
 - list.c, 15
 - list.h, 32
- list_del_after
 - list.c, 16
 - list.h, 32
- list_del_at
 - list.c, 17
 - list.h, 33
- list_elt
 - list.h, 26
- list_first
 - list.c, 17
 - list.h, 33
- list_foreach
 - list.h, 26
- list_foreach_elt
 - list.h, 27
- list_foreach_elt_safe
 - list.h, 27
- list_foreach_safe
 - list.h, 27

- list.h, 28
- list_init
 - list.c, 18
 - list.h, 34
- list_insert_after
 - list.c, 18
 - list.h, 34
- list_insert_at
 - list.c, 19
 - list.h, 35
- list_is_empty
 - list.c, 19
 - list.h, 36
- list_last
 - list.c, 20
 - list.h, 36
- list_length
 - list.c, 20
 - list.h, 37
- list_next
 - list.c, 21
 - list.h, 37
- list_node, 6
 - next, 6
- list_print
 - list.c, 21
 - list.h, 38
- list_reverse
 - list.c, 22
 - list.h, 38
- list_sort
 - list.c, 22
 - list.h, 39
- list_split_at
 - list.c, 23
 - list.h, 39
- list_swap
 - list.c, 24
 - list.h, 40
- main
 - tAltris.c, 10
- matrix, 7
 - cols, 7
 - data, 7
 - rows, 7
- matrix.c
 - matrix_at, 42
 - matrix_cols, 42
 - matrix_copy, 43
 - matrix_create, 43
 - matrix_dot_product, 44
 - matrix_free, 44
 - matrix_product, 45
 - matrix_rows, 45
 - matrix_scale, 46
 - matrix_set, 47
 - matrix_sum, 47
 - matrix_transpose, 48
- matrix.h
 - matrix_at, 50
 - matrix_cols, 51
 - matrix_copy, 51
 - matrix_create, 52
 - matrix_dot_product, 52
 - matrix_free, 53
 - matrix_product, 53
 - matrix_rows, 54
 - matrix_scale, 54
 - matrix_set, 55
 - matrix_sum, 56
 - matrix_transpose, 56
- matrix_at
 - matrix.c, 42
 - matrix.h, 50
- matrix_cols
 - matrix.c, 42
 - matrix.h, 51
- matrix_copy
 - matrix.c, 43
 - matrix.h, 51
- matrix_create
 - matrix.c, 43
 - matrix.h, 52
- matrix_dot_product
 - matrix.c, 44
 - matrix.h, 52
- matrix_free
 - matrix.c, 44
 - matrix.h, 53
- matrix_product
 - matrix.c, 45
 - matrix.h, 53
- matrix_rows
 - matrix.c, 45
 - matrix.h, 54
- matrix_scale
 - matrix.c, 46
 - matrix.h, 54
- matrix_set
 - matrix.c, 47
 - matrix.h, 55
- matrix_sum
 - matrix.c, 47
 - matrix.h, 56
- matrix_transpose
 - matrix.c, 48
 - matrix.h, 56
- next
 - list_node, 6
- rows
 - matrix, 7
- src/tAltris.c, 9
- src/tAltris.h, 10
- src/utlis/list.c, 11

src/utls/list.h, 24
src/utls/matrix.c, 41
src/utls/matrix.h, 49

tAltris.c
 main, 10