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 Function Documentation . . . 9

4.1.1.1 main() . . . 9

4.2 src/tAltris.h File Reference . . . 10

4.3 src/utils/list.c File Reference . . . 10

4.3.1 Function Documentation . . . 11

4.3.1.1 list_add() . . . 11

4.3.1.2 list_advance() . . . 11

4.3.1.3 list_append() . . . 12

4.3.1.4 list_at() . . . 13

4.3.1.5 list_concat() . . . 14

4.3.1.6 list_del() . . . 15

4.3.1.7 list_del_after() . . . 15

4.3.1.8 list_del_at() . . . 16

4.3.1.9 list_first() . . . 16

4.3.1.10 list_init() . . . 17

4.3.1.11 list_insert_after() . . . 17

4.3.1.12 list_insert_at() . . . 18

4.3.1.13 list_is_empty() . . . 18

4.3.1.14 list_last() . . . 19

4.3.1.15 list_length() . . . 19

4.3.1.16 list_next() . . . 20

4.3.1.17 list_print() . . . 20

4.3.1.18 list_reverse() . . . 21

4.3.1.19 list_sort() . . . 21

4.3.1.20 list_split_at() . . . 22

4.3.1.21 list_swap() . . . 23

4.4 src/utils/list.h File Reference . . . 23

4.4.1 Detailed Description . . . 25

4.4.2 Macro Definition Documentation . . . 25
4.4.2.1 list_elt . . . 25
4.4.2.2 list_foreach . . . 25
4.4.2.3 list_foreach_elt . . . 26
4.4.2.4 list_foreach_elt_safe . . . 27
4.4.2.5 list_foreach_safe . . . 27
4.4.3 Function Documentation . . . 28
4.4.3.1 list_add() . . . 28
4.4.3.2 list_advance() . . . 29
4.4.3.3 list_append() . . . 29
4.4.3.4 list_at() . . . 30
4.4.3.5 list_concat() . . . 30
4.4.3.6 list_del() . . . 31
4.4.3.7 list_del_after() . . . 31
4.4.3.8 list_del_at() . . . 32
4.4.3.9 list_first() . . . 33
4.4.3.10 list_init() . . . 33
4.4.3.11 list_insert_after() . . . 34
4.4.3.12 list_insert_at() . . . 34
4.4.3.13 list_is_empty() . . . 35
4.4.3.14 list_last() . . . 35
4.4.3.15 list_length() . . . 36
4.4.3.16 list_next() . . . 36
4.4.3.17 list_print() . . . 37
4.4.3.18 list_reverse() . . . 37
4.4.3.19 list_sort() . . . 38
4.4.3.20 list_split_at() . . . 38
4.4.3.21 list_swap() . . . 39
4.5 src/utils/matrix.c File Reference . . . 40
4.5.1 Function Documentation . . . 40

4.5.1.1 matrix_at() . . . 40
4.5.1.2 matrix_cols() . . . 41
4.5.1.3 matrix_copy() . . . 42
4.5.1.4 matrix_create() . . . 43
4.5.1.5 matrix_free() . . . 43
4.5.1.6 matrix_init() . . . 44
4.5.1.7 matrix_product() . . . 45
4.5.1.8 matrix_rows() . . . 45
4.5.1.9 matrix_set() . . . 46
4.5.1.10 matrix_sum() . . . 46
4.5.1.11 matrix_transpose() . . . 47
4.6 src/utils/matrix.h File Reference . . . 47
4.6.1 Detailed Description . . . 49
4.6.2 Function Documentation . . . 49
4.6.2.1 matrix_at() . . . 49
4.6.2.2 matrix_cols() . . . 50
4.6.2.3 matrix_copy() . . . 51
4.6.2.4 matrix_create() . . . 51
4.6.2.5 matrix_free() . . . 52
4.6.2.6 matrix_init() . . . 53
4.6.2.7 matrix_product() . . . 53
4.6.2.8 matrix_rows() . . . 54
4.6.2.9 matrix_set() . . . 54
4.6.2.10 matrix_sum() . . . 55
4.6.2.11 matrix_transpose() . . . 56

Index **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** 9
src/ **tAltris.h** 10
src/utils/ **list.c** 10
src/utils/ **list.h**
Intrusive list implement 23
src/utils/ **matrix.c** 40
src/utils/ **matrix.h**
Matrix implement 47

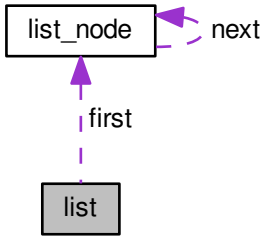
Chapter 3

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.

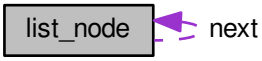
The documentation for this struct was generated from the following file:

- 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**

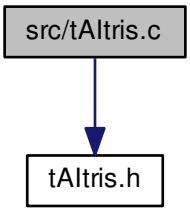
Chapter 4

File Documentation

4.1 src/tAItris.c File Reference

```
#include "tAItris.h"
```

Include dependency graph for tAItris.c:



Functions

- int **main** ()

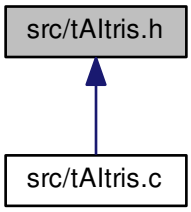
4.1.1 Function Documentation

4.1.1.1 main()

```
int main ( )
```

4.2 src/tAltris.h File Reference

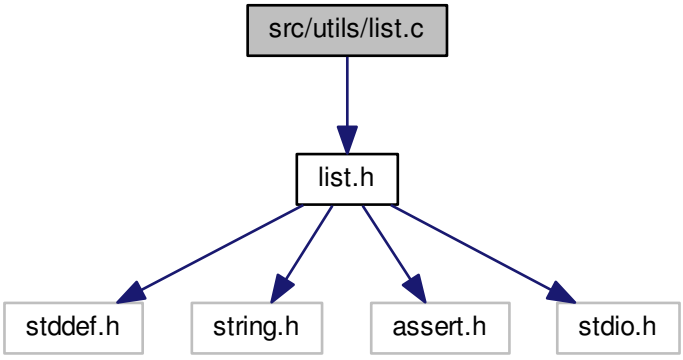
This graph shows which files directly or indirectly include this file:



4.3 src/utils/list.c File Reference

```
#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 Function Documentation

4.3.1.1 list_add()

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

Adds `node` in the front of `list`

Parameters

list	a list.
node	the new node.

Precondition

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

Postcondition

List size increases by 1.

Remarks

Complexity: O(1)

4.3.1.2 list_advance()

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

Returns the nth-node after the current one.

Parameters

node	a node.
distance	distance to move on.

Returns

the nth-node after `node`.

Precondition

`node` must be not NULL.

Remarks

Complexity: O(n)

4.3.1.3 list_append()

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

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

Parameters

list	a list.
node	the new node.

Precondition

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

Postcondition

List size increases by 1.

Remarks

Complexity: O(n)

4.3.1.4 list_at()

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

Returns node at the position pos.

Parameters

list	a list.
pos	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)

4.3.1.5 list_concat()

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

Concatenates two lists.

Parameters

l1	list 1.
l2	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)

4.3.1.6 list_del()

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

Deletes the first node.

Parameters

list	a list.

Precondition

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

Postcondition

List size decreases by 1.

Remarks

Complexity: O(1)

4.3.1.7 list_del_after()

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

Deletes the node at after the node `curr`.

Parameters

list	a list.
node	a node of `list`.

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)

4.3.1.8 list_del_at()

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

Deletes the node at the position `pos`.

Parameters

list	a list.
pos	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)

4.3.1.9 list_first()

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

Returns the first node.

Parameters

list	a list.

Returns

the first node.

Precondition

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

Remarks

Complexity: O(1)

4.3.1.10 list_init()

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

Initializes the list.

Parameters

list	a list.

Precondition

`list` must be not NULL.

Postcondition

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

Remarks

Complexity: O(1)

4.3.1.11 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

list	a list.
curr	a node of `list`.
node	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)

4.3.1.12 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

list	a list.
node	new node.
pos	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)

4.3.1.13 list_is_empty()

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

Tests if a list is empty.

Parameters

list	a list.

Returns

1 if the list is empty, otherwise 0.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(1)

4.3.1.14 list_last()

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

Returns the last node.

Parameters

list	a list.

Returns

the last node.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(N)

4.3.1.15 list_length()

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

Returns the size of the list.

Parameters

list	a list.

Returns

the length of the list.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(1)

4.3.1.16 list_next()

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

Returns the next node.

Parameters

node	a node.

Returns

the next node.

Precondition

`node` must be not NULL.

Remarks

Complexity: O(1)

4.3.1.17 list_print()

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

Print the list

Parameters

list	a list

4.3.1.18 list_reverse()

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

Reverses the order of the elements in the list.

Parameters

list	a list.

Precondition

`list` must be not NULL.

Remarks

Complexity: O(N)

4.3.1.19 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

list	list to sort.
cmp	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)

4.3.1.20 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

list	list to split.
pos	position (0-based) where to split the list.
right	an empty list to receive the part after `pos`

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)

4.3.1.21 list_swap()

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

Swaps two lists.

Parameters

l1	list 1.
l2	list 2.

Precondition

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

Remarks

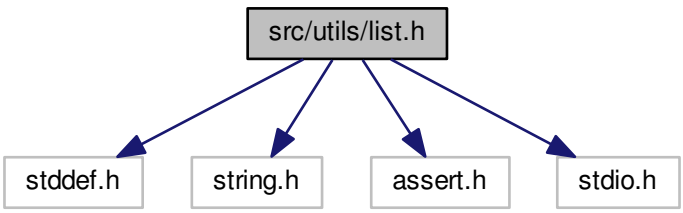
Complexity: O(1)

4.4 src/utils/list.h File Reference

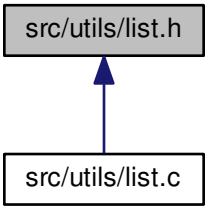
Intrusive list implement.

```
#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

node	a list node (struct list_node*).
type	type of the structure which contains the node.
fieldname	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

list	a list (struct list∗).
curr	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

list	a list (struct list∗).
curr	pointer (type∗) used to hold the current element.
type	type of the structure which contains the node.
fieldname	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

list	a list (struct list∗).
curr	pointer (type∗) used to hold the current element.
tmp	a struct list_node∗ used as temporary storage.
type	type of the structure which contains the node.
fieldname	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

list	a list (struct list∗).
curr	a struct list_node∗ used to hold the current element.
tmp	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

list	a list.
node	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

node	a node.
distance	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

list	a list.
node	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

list	a list.
pos	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

l1	list 1.
l2	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

list	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

list	a list.
node	a node of `list`.

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

list	a list.
pos	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

list	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

list	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

list	a list.
curr	a node of `list`.
node	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

list	a list.
node	new node.
pos	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

list	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

list	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

list	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

node	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

list	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

list	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

list	list to sort.
cmp	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

list	list to split.
pos	position (0-based) where to split the list.
right	an empty list to receive the part after `pos`

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

l1	list 1.
l2	list 2.

Precondition

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

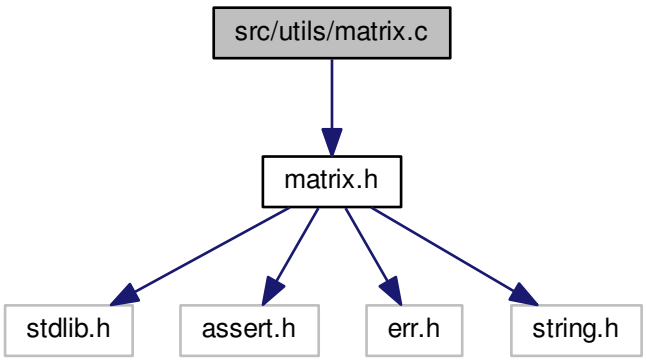
Remarks

Complexity: O(1)

4.5 src/utils/matrix.c File Reference

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

Include dependency graph for matrix.c:



Functions

- void **matrix_init** (struct **matrix** *mx, size_t rows, size_t cols)
- 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)
- struct **matrix** * **matrix_transpose** (const struct **matrix** *mx)
- struct **matrix** * **matrix_sum** (const struct **matrix** *mx1, const struct **matrix** *mx2)
- struct **matrix** * **matrix_product** (const struct **matrix** *mx1, const struct **matrix** *mx2)

4.5.1 Function Documentation

4.5.1.1 matrix_at()

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

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

Parameters

mx	a matrix
rows	rows
cols	columns

Returns

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

Precondition

`mx` must be not NULL
`mx->data` must be not NULL
`rows` must be between zero and `matrix_rows(mx)`
`cols` must be between zero and `matrix_cols(mx)`

Remarks

Complexity: O(1)

4.5.1.2 matrix_cols()

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

Get the number of columns

Parameters

mx	a matrix

Returns

the number of columns

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.5.1.3 matrix_copy()

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

Copy the matrix

Parameters

mx	a matrix

Returns

the copy of `mx`

Precondition

`mx` must be not NULL
`mx->data` must be not NULL

Remarks

Complexity: O(1)

4.5.1.4 matrix_create()

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

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

Parameters

rows	number of rows
cols	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)

4.5.1.5 matrix_free()

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

Free the matrix

Parameters

mx	a matrix

Precondition

`mx` must be not NULL
`mx->data` must be not NULL

Postcondition

`mx` is freed
`mx->data` is freed

Remarks

Complexity: O(1)

4.5.1.6 matrix_init()

```
void matrix_init (
            struct  matrix * mx,
            size_t rows,
            size_t cols )  [inline]
```

Initialize the matrix

Parameters

mx	a matrix
rows	number of rows
cols	number of columns

Precondition

`mx` must be not NULL
`rows` must be greater than zero
`cols` must be greater than zero

Postcondition

`mx` is size of `rows` rows and `cols` columns
`mx->data` is initialized with the right size

Remarks

Complexity: O(1)

4.5.1.7 matrix_product()

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

Multiply the matrix `mx1` with `mx2`

Parameters

mx1	a matrix
mx2	a matrix

Returns

the product of `mx1` with `mx2`

Precondition

`mx1` must be not NULL
`mx1->data` must be not NULL
`mx2` must be not NULL
`mx2->data` must be not NULL
`matrix_cols(mx1)` must be equal to `matrix_rows(mx2)`

Remarks

the result is a matrix of `matrix_rows(mx1)` rows and `matrix_cols(mx2)` cols
Complexity: O(nmp)

4.5.1.8 matrix_rows()

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

Get the number of rows

Parameters

mx	a matrix

Returns

the number of rows

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.5.1.9 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`

Parameters

mx	a matrix
rows	rows
cols	columns
value	a value

Precondition

`mx` must be not NULL
`mx->data` must be not NULL
`rows` must be between zero and `matrix_rows(mx)`
`cols` must be between zero and `matrix_cols(mx)`

Postcondition

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

Remarks

Complexity: O(1)

4.5.1.10 matrix_sum()

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

Sum the matrix `mx1` with `mx2`

Parameters

mx1	a matrix
mx2	a matrix

Returns

the sum of `mx1` and `mx2`

Precondition

`mx1` must be not NULL
`mx1->data` must be not NULL
`mx2` must be not NULL
`mx2->data` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`

Remarks

Complexity: O(N)

4.5.1.11 matrix_transpose()

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

Transpose the matrix

Parameters

mx	a matrix

Returns

the transposed matrix

Precondition

`mx` must be not NULL
`mx->data` must be not NULL

Remarks

transposed matrix is size of `matrix_cols(mx)` rows and `matrix_rows(mx)` columns
Complexity: O(N)

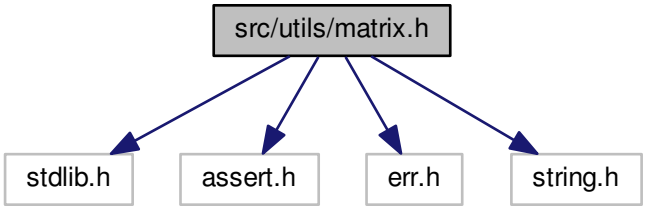
4.6 src/utils/matrix.h File Reference

Matrix implement.

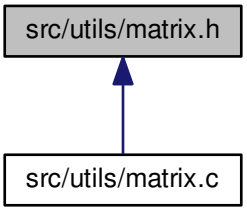
```
#include <stdlib.h>
#include <assert.h>
```

```
#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

- void **matrix_init** (struct **matrix** *mx, size_t rows, size_t cols)
- 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)
- struct **matrix** * **matrix_transpose** (const struct **matrix** *mx)
- struct **matrix** * **matrix_sum** (const struct **matrix** *mx1, const struct **matrix** *mx2)
- struct **matrix** * **matrix_product** (const struct **matrix** *mx1, const struct **matrix** *mx2)

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

Parameters

mx	a matrix
rows	rows
cols	columns

Returns

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

Precondition

`mx` must be not NULL
`mx->data` must be not NULL
`rows` must be between zero and `matrix_rows(mx)`
`cols` must be between zero and `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

Parameters

mx	a matrix

Returns

the number of columns

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

Parameters

mx	a matrix

Returns

the copy of `mx`

Precondition

`mx` must be not NULL
`mx->data` 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

rows	number of rows
cols	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_free()

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

Free the matrix

Parameters

mx	a matrix

Precondition

`mx` must be not NULL
`mx->data` must be not NULL

Postcondition

`mx` is freed
`mx->data` is freed

Remarks

Complexity: O(1)

4.6.2.6 matrix_init()

```
void matrix_init (
            struct  matrix * mx,
            size_t rows,
            size_t cols )  [inline]
```

Initialize the matrix

Parameters

mx	a matrix
rows	number of rows
cols	number of columns

Precondition

`mx` must be not NULL
`rows` must be greater than zero
`cols` must be greater than zero

Postcondition

`mx` is size of `rows` rows and `cols` columns
`mx->data` is initialized with the right size

Remarks

Complexity: O(1)

4.6.2.7 matrix_product()

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

Multiply the matrix `mx1` with `mx2`

Parameters

mx1	a matrix
mx2	a matrix

Returns

the product of `mx1` with `mx2`

Precondition

`mx1` must be not NULL
`mx1->data` must be not NULL
`mx2` must be not NULL
`mx2->data` must be not NULL
`matrix_cols(mx1)` must be equal to `matrix_rows(mx2)`

Remarks

the result is a matrix of `matrix_rows(mx1)` rows and `matrix_cols(mx2)` cols
Complexity: O(nmp)

4.6.2.8 matrix_rows()

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

Get the number of rows

Parameters

mx	a matrix

Returns

the number of rows

Precondition

`mx` must be not NULL

Remarks

Complexity: O(1)

4.6.2.9 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`

Parameters

mx	a matrix
rows	rows
cols	columns
value	a value

Precondition

`mx` must be not NULL
`mx->data` must be not NULL
`rows` must be between zero and `matrix_rows(mx)`
`cols` must be between zero and `matrix_cols(mx)`

Postcondition

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

Remarks

Complexity: O(1)

4.6.2.10 matrix_sum()

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

Sum the matrix `mx1` with `mx2`

Parameters

mx1	a matrix
mx2	a matrix

Returns

the sum of `mx1` and `mx2`

Precondition

`mx1` must be not NULL
`mx1->data` must be not NULL
`mx2` must be not NULL
`mx2->data` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`

Remarks

Complexity: O(N)

4.6.2.11 matrix_transpose()

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

Transpose the matrix

Parameters

mx	a matrix

Returns

the transposed matrix

Precondition

`mx` must be not NULL
`mx->data` must be not NULL

Remarks

transposed matrix is size of `matrix_cols(mx)` rows and `matrix_rows(mx)` columns
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, 11
    list_advance, 11
    list_append, 12
    list_at, 12
    list_concat, 14
    list_del, 14
    list_del_after, 15
    list_del_at, 16
    list_first, 16
    list_init, 17
    list_insert_after, 17
    list_insert_at, 18
    list_is_empty, 18
    list_last, 19
    list_length, 19
    list_next, 20
    list_print, 20
    list_reverse, 21
    list_sort, 21
    list_split_at, 22
    list_swap, 23
list.h
    list_add, 28
    list_advance, 28
    list_append, 29
    list_at, 30
    list_concat, 30
    list_del, 31
    list_del_after, 31
    list_del_at, 32
    list_elt, 25
    list_first, 32
    list_foreach, 25
    list_foreach_elt, 26
    list_foreach_elt_safe, 26
    list_foreach_safe, 27
    list_init, 33
    list_insert_after, 33
    list_insert_at, 34
    list_is_empty, 35
    list_last, 35
    list_length, 36
    list_next, 36
    list_print, 37
    list_reverse, 37
    list_sort, 38
    list_split_at, 38
    list_swap, 39
list_add
    list.c, 11
    list.h, 28
list_advance
    list.c, 11
    list.h, 28
list_append
    list.c, 12
    list.h, 29
list_at
    list.c, 12
    list.h, 30
list_concat
    list.c, 14
    list.h, 30
list_del
    list.c, 14
    list.h, 31
list_del_after
    list.c, 15
    list.h, 31
list_del_at
    list.c, 16
    list.h, 32
list_elt
    list.h, 25
list_first
    list.c, 16
    list.h, 32
list_foreach
    list.h, 25
list_foreach_elt
    list.h, 26
list_foreach_elt_safe
    list.h, 26
list_foreach_safe

list.h, 27
list_init
list.c, 17
list.h, 33
list_insert_after
list.c, 17
list.h, 33
list_insert_at
list.c, 18
list.h, 34
list_is_empty
list.c, 18
list.h, 35
list_last
list.c, 19
list.h, 35
list_length
list.c, 19
list.h, 36
list_next
list.c, 20
list.h, 36
list_node, 6
next, 6
list_print
list.c, 20
list.h, 37
list_reverse
list.c, 21
list.h, 37
list_sort
list.c, 21
list.h, 38
list_split_at
list.c, 22
list.h, 38
list_swap
list.c, 23
list.h, 39

main
tAItris.c, 9
matrix, 7
cols, 7
data, 7
rows, 7
matrix.c
matrix_at, 40
matrix_cols, 41
matrix_copy, 41
matrix_create, 43
matrix_free, 43
matrix_init, 44
matrix_product, 44
matrix_rows, 45
matrix_set, 46
matrix_sum, 46
matrix_transpose, 47
matrix.h
matrix_at, 49
matrix_cols, 49
matrix_copy, 51
matrix_create, 51
matrix_free, 52
matrix_init, 52
matrix_product, 53
matrix_rows, 54
matrix_set, 54
matrix_sum, 55
matrix_transpose, 56
matrix_at
matrix.c, 40
matrix.h, 49
matrix_cols
matrix.c, 41
matrix.h, 49
matrix_copy
matrix.c, 41
matrix.h, 51
matrix_create
matrix.c, 43
matrix.h, 51
matrix_free
matrix.c, 43
matrix.h, 52
matrix_init
matrix.c, 44
matrix.h, 52
matrix_product
matrix.c, 44
matrix.h, 53
matrix_rows
matrix.c, 45
matrix.h, 54
matrix_set
matrix.c, 46
matrix.h, 54
matrix_sum
matrix.c, 46
matrix.h, 55
matrix_transpose
matrix.c, 47
matrix.h, 56

next
list_node, 6

rows
matrix, 7

src/tAItris.c, 9
src/tAItris.h, 10
src/utils/list.c, 10
src/utils/list.h, 23
src/utils/matrix.c, 40
src/utils/matrix.h, 47

tAItris.c
main, 9