

cdatastruct

Generated by Doxygen 1.8.1.2

Fri Sep 6 2013 20:12:51



# Contents

<b>1</b>	<b>Data Structure Index</b>	<b>1</b>
1.1	Data Structures . . . . .	1
<b>2</b>	<b>File Index</b>	<b>3</b>
2.1	File List . . . . .	3
<b>3</b>	<b>Data Structure Documentation</b>	<b>5</b>
3.1	dl_list_node_t Struct Reference . . . . .	5
3.1.1	Field Documentation . . . . .	5
3.1.1.1	data . . . . .	5
3.1.1.2	next . . . . .	5
3.1.1.3	prev . . . . .	5
3.2	dl_list_t Struct Reference . . . . .	6
3.2.1	Field Documentation . . . . .	6
3.2.1.1	back . . . . .	6
3.2.1.2	cfunc . . . . .	6
3.2.1.3	front . . . . .	6
3.2.1.4	length . . . . .	6
3.3	sl_list_node_t Struct Reference . . . . .	7
3.3.1	Field Documentation . . . . .	7
3.3.1.1	data . . . . .	7
3.3.1.2	next . . . . .	7
3.4	sl_list_t Struct Reference . . . . .	7
3.4.1	Field Documentation . . . . .	8
3.4.1.1	cfunc . . . . .	8
3.4.1.2	front . . . . .	8
3.4.1.3	length . . . . .	8
<b>4</b>	<b>File Documentation</b>	<b>9</b>
4.1	cdatastruct.h File Reference . . . . .	9
4.1.1	Detailed Description . . . . .	9
4.2	cds_common.h File Reference . . . . .	9

4.2.1	Detailed Description	10
4.2.2	Enumeration Type Documentation	10
4.2.2.1	cds_error	10
4.3	cds_dl_list.h File Reference	10
4.3.1	Detailed Description	12
4.3.2	Function Documentation	13
4.3.2.1	dl_list_append	13
4.3.2.2	dl_list_data	13
4.3.2.3	dl_list_delete_at	13
4.3.2.4	dl_list_find_index	13
4.3.2.5	dl_list_find_itr	13
4.3.2.6	dl_list_first	14
4.3.2.7	dl_list_free	14
4.3.2.8	dl_list_init	14
4.3.2.9	dl_list_insert_after	14
4.3.2.10	dl_list_insert_at	14
4.3.2.11	dl_list_insert_before	15
4.3.2.12	dl_list_isempty	15
4.3.2.13	dl_list_itr_from_index	15
4.3.2.14	dl_list_last	15
4.3.2.15	dl_list_length	16
4.3.2.16	dl_list_next	16
4.3.2.17	dl_list_prepend	16
4.3.2.18	dl_list_prev	16
4.4	cds_general.h File Reference	16
4.4.1	Detailed Description	17
4.4.2	Function Documentation	18
4.4.2.1	cds_compare_int	18
4.4.2.2	cds_compare_long	18
4.4.2.3	cds_compare_string	18
4.4.2.4	cds_compare_uint	18
4.4.2.5	cds_compare_ulong	18
4.4.2.6	cds_new_int	19
4.4.2.7	cds_new_long	19
4.4.2.8	cds_new_string	19
4.4.2.9	cds_new_uint	19
4.4.2.10	cds_new_ulong	19
4.5	cds_queue.h File Reference	20
4.5.1	Detailed Description	21
4.5.2	Function Documentation	21

4.5.2.1	<a href="#">queue_free</a>	21
4.5.2.2	<a href="#">queue_init</a>	21
4.5.2.3	<a href="#">queue_isempty</a>	21
4.5.2.4	<a href="#">queue_length</a>	21
4.5.2.5	<a href="#">queue_pop</a>	22
4.5.2.6	<a href="#">queue_pushback</a>	22
4.6	<a href="#">cds_sl_list.h File Reference</a>	22
4.6.1	<a href="#">Detailed Description</a>	24
4.6.2	<a href="#">Function Documentation</a>	24
4.6.2.1	<a href="#">sl_list_data</a>	24
4.6.2.2	<a href="#">sl_list_delete_at</a>	24
4.6.2.3	<a href="#">sl_list_find_index</a>	24
4.6.2.4	<a href="#">sl_list_find_itr</a>	25
4.6.2.5	<a href="#">sl_list_first</a>	25
4.6.2.6	<a href="#">sl_list_free</a>	25
4.6.2.7	<a href="#">sl_list_init</a>	25
4.6.2.8	<a href="#">sl_list_insert_after</a>	25
4.6.2.9	<a href="#">sl_list_insert_at</a>	26
4.6.2.10	<a href="#">sl_list_isempty</a>	26
4.6.2.11	<a href="#">sl_list_itr_from_index</a>	26
4.6.2.12	<a href="#">sl_list_length</a>	26
4.6.2.13	<a href="#">sl_list_next</a>	27
4.6.2.14	<a href="#">sl_list_prepend</a>	27
4.7	<a href="#">cds_stack.h File Reference</a>	27
4.7.1	<a href="#">Detailed Description</a>	28
4.7.2	<a href="#">Function Documentation</a>	28
4.7.2.1	<a href="#">stack_free</a>	28
4.7.2.2	<a href="#">stack_init</a>	29
4.7.2.3	<a href="#">stack_isempty</a>	29
4.7.2.4	<a href="#">stack_length</a>	29
4.7.2.5	<a href="#">stack_pop</a>	29
4.7.2.6	<a href="#">stack_push</a>	29
4.8	<a href="#">dl_list.c File Reference</a>	30
4.8.1	<a href="#">Detailed Description</a>	31
4.8.2	<a href="#">Function Documentation</a>	31
4.8.2.1	<a href="#">dl_list_append</a>	31
4.8.2.2	<a href="#">dl_list_data</a>	32
4.8.2.3	<a href="#">dl_list_delete_at</a>	32
4.8.2.4	<a href="#">dl_list_find</a>	32
4.8.2.5	<a href="#">dl_list_find_index</a>	32

4.8.2.6	<a href="#">dl_list_find_itr</a>	32
4.8.2.7	<a href="#">dl_list_first</a>	33
4.8.2.8	<a href="#">dl_list_free</a>	33
4.8.2.9	<a href="#">dl_list_free_node</a>	33
4.8.2.10	<a href="#">dl_list_init</a>	33
4.8.2.11	<a href="#">dl_list_insert_after</a>	33
4.8.2.12	<a href="#">dl_list_insert_at</a>	34
4.8.2.13	<a href="#">dl_list_insert_before</a>	34
4.8.2.14	<a href="#">dl_list_insert_node_after_mid</a>	34
4.8.2.15	<a href="#">dl_list_insert_node_back</a>	34
4.8.2.16	<a href="#">dl_list_insert_node_before_mid</a>	34
4.8.2.17	<a href="#">dl_list_insert_node_front</a>	35
4.8.2.18	<a href="#">dl_list_isempty</a>	35
4.8.2.19	<a href="#">dl_list_itr_from_index</a>	35
4.8.2.20	<a href="#">dl_list_last</a>	35
4.8.2.21	<a href="#">dl_list_length</a>	35
4.8.2.22	<a href="#">dl_list_new_node</a>	36
4.8.2.23	<a href="#">dl_list_next</a>	36
4.8.2.24	<a href="#">dl_list_prepend</a>	36
4.8.2.25	<a href="#">dl_list_prev</a>	36
4.8.2.26	<a href="#">dl_list_remove_at</a>	36
4.8.2.27	<a href="#">dl_list_remove_node_back</a>	36
4.8.2.28	<a href="#">dl_list_remove_node_front</a>	37
4.8.2.29	<a href="#">dl_list_remove_node_mid</a>	37
4.9	<a href="#">dl_list.h File Reference</a>	37
4.9.1	<a href="#">Detailed Description</a>	39
4.9.2	<a href="#">Function Documentation</a>	39
4.9.2.1	<a href="#">dl_list_find</a>	39
4.9.2.2	<a href="#">dl_list_free_node</a>	39
4.9.2.3	<a href="#">dl_list_insert_node_after_mid</a>	40
4.9.2.4	<a href="#">dl_list_insert_node_back</a>	40
4.9.2.5	<a href="#">dl_list_insert_node_before_mid</a>	40
4.9.2.6	<a href="#">dl_list_insert_node_front</a>	40
4.9.2.7	<a href="#">dl_list_new_node</a>	40
4.9.2.8	<a href="#">dl_list_remove_at</a>	40
4.9.2.9	<a href="#">dl_list_remove_node_back</a>	41
4.9.2.10	<a href="#">dl_list_remove_node_front</a>	41
4.9.2.11	<a href="#">dl_list_remove_node_mid</a>	41
4.10	<a href="#">general.c File Reference</a>	41
4.10.1	<a href="#">Detailed Description</a>	42

4.10.2	Function Documentation	43
4.10.2.1	<code>cds_compare_int</code>	43
4.10.2.2	<code>cds_compare_long</code>	43
4.10.2.3	<code>cds_compare_string</code>	43
4.10.2.4	<code>cds_compare_uint</code>	43
4.10.2.5	<code>cds_compare_ulong</code>	43
4.10.2.6	<code>cds_new_int</code>	44
4.10.2.7	<code>cds_new_long</code>	44
4.10.2.8	<code>cds_new_string</code>	44
4.10.2.9	<code>cds_new_uint</code>	44
4.10.2.10	<code>cds_new_ulong</code>	44
4.11	<code>queue.c</code> File Reference	45
4.11.1	Detailed Description	46
4.11.2	Function Documentation	46
4.11.2.1	<code>queue_free</code>	46
4.11.2.2	<code>queue_init</code>	46
4.11.2.3	<code>queue_isempty</code>	46
4.11.2.4	<code>queue_length</code>	46
4.11.2.5	<code>queue_pop</code>	46
4.11.2.6	<code>queue_pushback</code>	47
4.12	<code>sl_list.c</code> File Reference	47
4.12.1	Detailed Description	48
4.12.2	Function Documentation	48
4.12.2.1	<code>sl_list_data</code>	48
4.12.2.2	<code>sl_list_delete_at</code>	49
4.12.2.3	<code>sl_list_find</code>	49
4.12.2.4	<code>sl_list_find_index</code>	49
4.12.2.5	<code>sl_list_find_itr</code>	49
4.12.2.6	<code>sl_list_first</code>	49
4.12.2.7	<code>sl_list_free</code>	50
4.12.2.8	<code>sl_list_free_node</code>	50
4.12.2.9	<code>sl_list_init</code>	50
4.12.2.10	<code>sl_list_insert_after</code>	50
4.12.2.11	<code>sl_list_insert_at</code>	50
4.12.2.12	<code>sl_list_isempty</code>	51
4.12.2.13	<code>sl_list_itr_from_index</code>	51
4.12.2.14	<code>sl_list_length</code>	51
4.12.2.15	<code>sl_list_new_node</code>	51
4.12.2.16	<code>sl_list_next</code>	51
4.12.2.17	<code>sl_list_prepend</code>	51

---

4.12.2.18 <code>sl_list_remove_at</code> . . . . .	52
4.13 <code>sl_list.h</code> File Reference . . . . .	52
4.13.1 Detailed Description . . . . .	53
4.13.2 Function Documentation . . . . .	54
4.13.2.1 <code>sl_list_find</code> . . . . .	54
4.13.2.2 <code>sl_list_free_node</code> . . . . .	54
4.13.2.3 <code>sl_list_new_node</code> . . . . .	54
4.13.2.4 <code>sl_list_remove_at</code> . . . . .	54
4.14 <code>stack.c</code> File Reference . . . . .	54
4.14.1 Detailed Description . . . . .	55
4.14.2 Function Documentation . . . . .	55
4.14.2.1 <code>stack_free</code> . . . . .	55
4.14.2.2 <code>stack_init</code> . . . . .	56
4.14.2.3 <code>stack_isempty</code> . . . . .	56
4.14.2.4 <code>stack_length</code> . . . . .	56
4.14.2.5 <code>stack_pop</code> . . . . .	56
4.14.2.6 <code>stack_push</code> . . . . .	56



# Chapter 1

## Data Structure Index

### 1.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">dl_list_node_t</a>	Struct for double linked list node . . . . .	5
<a href="#">dl_list_t</a>	Struct to contain a list . . . . .	6
<a href="#">sl_list_node_t</a>	Struct for singly linked list node . . . . .	7
<a href="#">sl_list_t</a>	Struct to contain a list . . . . .	7



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">cdatastruct.h</a>	Interface to generic C data structures . . . . .	9
<a href="#">cds_common.h</a>	Common data types and data for C data structures library . . . . .	9
<a href="#">cds_dl_list.h</a>	User interface to doubly linked list data structure . . . . .	10
<a href="#">cds_general.h</a>	Interface to general data structure helper functions . . . . .	16
<a href="#">cds_queue.h</a>	User interface to queue data structure . . . . .	20
<a href="#">cds_sl_list.h</a>	User interface to singly linked list data structure . . . . .	22
<a href="#">cds_stack.h</a>	User interface to stack data structure . . . . .	27
<a href="#">dl_list.c</a>	Implementation of doubly linked list data structure . . . . .	30
<a href="#">dl_list.h</a>	Developer interface to double linked list data structure . . . . .	37
<a href="#">general.c</a>	Implementation of general data structure helper functions . . . . .	41
<a href="#">queue.c</a>	Implementation of queue data structure . . . . .	45
<a href="#">sl_list.c</a>	Implementation of singly linked list data structure . . . . .	47
<a href="#">sl_list.h</a>	Developer interface to singly linked list data structure . . . . .	52
<a href="#">stack.c</a>	Implementation of stack data structure . . . . .	54



## Chapter 3

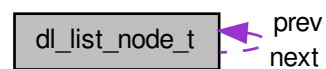
# Data Structure Documentation

### 3.1 dl\_list\_node\_t Struct Reference

Struct for double linked list node.

```
#include <cds_dl_list.h>
```

Collaboration diagram for dl\_list\_node\_t:



#### Data Fields

- void \* [data](#)
- struct [dl\\_list\\_node\\_t](#) \* [next](#)
- struct [dl\\_list\\_node\\_t](#) \* [prev](#)

#### 3.1.1 Field Documentation

##### 3.1.1.1 void\* dl\_list\_node\_t::data

Pointer to data

##### 3.1.1.2 struct dl\_list\_node\_t\* dl\_list\_node\_t::next

Pointer to next node

##### 3.1.1.3 struct dl\_list\_node\_t\* dl\_list\_node\_t::prev

Pointer to previous node

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

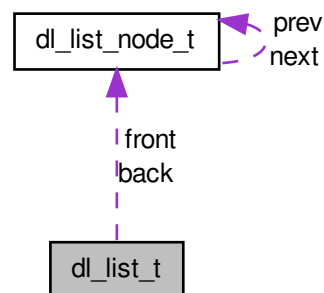
- [cds\\_dl\\_list.h](#)

## 3.2 dl\_list\_t Struct Reference

Struct to contain a list.

```
#include <dl_list.h>
```

Collaboration diagram for dl\_list\_t:



### Data Fields

- struct [dl\\_list\\_node\\_t](#) \* [front](#)
- struct [dl\\_list\\_node\\_t](#) \* [back](#)
- [size\\_t](#) [length](#)
- [int](#)(\* [cfunc](#) )()

### 3.2.1 Field Documentation

#### 3.2.1.1 struct [dl\\_list\\_node\\_t](#)\* [dl\\_list\\_t::back](#)

Pointer to last node

#### 3.2.1.2 [int](#)(\* [dl\\_list\\_t::cfunc](#))()

Pointer to compare function

#### 3.2.1.3 struct [dl\\_list\\_node\\_t](#)\* [dl\\_list\\_t::front](#)

Pointer to first node

#### 3.2.1.4 [size\\_t](#) [dl\\_list\\_t::length](#)

Length of list

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

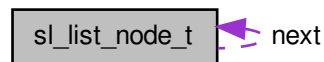
- [dl\\_list.h](#)

### 3.3 `sl_list_node_t` Struct Reference

Struct for singly linked list node.

```
#include <cds_sl_list.h>
```

Collaboration diagram for `sl_list_node_t`:



#### Data Fields

- `void * data`
- `struct sl\_list\_node\_t * next`

#### 3.3.1 Field Documentation

##### 3.3.1.1 `void* sl_list_node_t::data`

Pointer to data

##### 3.3.1.2 `struct sl_list_node_t* sl_list_node_t::next`

Pointer to next node

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

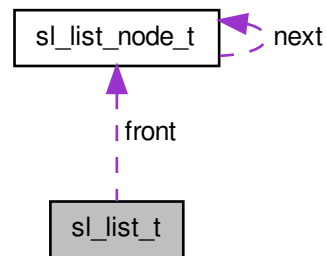
- [cds\\_sl\\_list.h](#)

### 3.4 `sl_list_t` Struct Reference

Struct to contain a list.

```
#include <sl_list.h>
```

Collaboration diagram for `sl_list_t`:



## Data Fields

- struct `sl_list_node_t` \* `front`
- `size_t` `length`
- `int(* cfunc )()`

### 3.4.1 Field Documentation

#### 3.4.1.1 `int(* sl_list_t::cfunc)()`

Pointer to compare function

#### 3.4.1.2 `struct sl_list_node_t* sl_list_t::front`

Pointer to first node

#### 3.4.1.3 `size_t sl_list_t::length`

Length of list

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

- [sl\\_list.h](#)



## Chapter 4

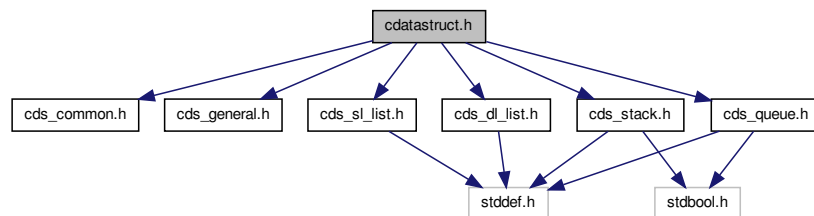
# File Documentation

### 4.1 cdatastruct.h File Reference

Interface to generic C data structures.

```
#include "cds_common.h"
#include "cds_general.h"
#include "cds_sl_list.h"
#include "cds_dl_list.h"
#include "cds_stack.h"
#include "cds_queue.h"
```

Include dependency graph for cdatastruct.h:



#### 4.1.1 Detailed Description

Interface to generic C data structures.

##### Author

Paul Griffiths

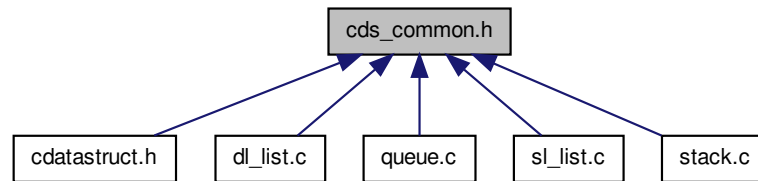
##### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.2 cds\_common.h File Reference

Common data types and data for C data structures library.

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



## Typedefs

- typedef enum `cds_error` `cds_error`  
*Enumeration of return error codes.*

## Enumerations

- enum `cds_error` { `CDSERR_ERROR` = -1, `CDSERR_OUTOFRANGE` = -2, `CDSERR_NOTFOUND` = -3, `CDSERR_BADITERATOR` = -4 }  
*Enumeration of return error codes.*

### 4.2.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.2.2 Enumeration Type Documentation

#### 4.2.2.1 enum `cds_error`

##### Enumerator:

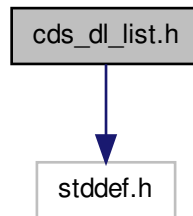
**`CDSERR_ERROR`** Unspecified error  
**`CDSERR_OUTOFRANGE`** Index out of range  
**`CDSERR_NOTFOUND`** Data element not found  
**`CDSERR_BADITERATOR`** Invalid iterator

## 4.3 `cds_dl_list.h` File Reference

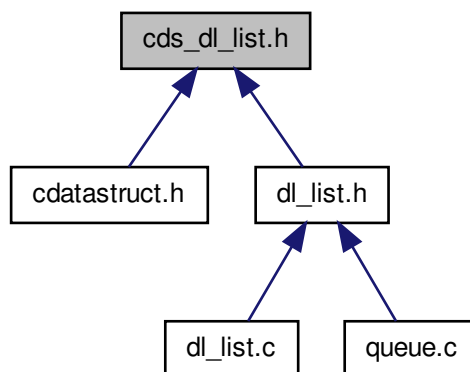
User interface to doubly linked list data structure.

```
#include <stddef.h>
```

Include dependency graph for cds\_dl\_list.h:



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



## Data Structures

- struct [dl\\_list\\_node\\_t](#)  
*Struct for double linked list node.*

## Typedefs

- typedef struct [dl\\_list\\_node\\_t](#) [dl\\_list\\_node\\_t](#)  
*Struct for double linked list node.*
- typedef struct [dl\\_list\\_t](#) \* [dl\\_list](#)  
*Typedef for list pointer.*
- typedef struct [dl\\_list\\_node\\_t](#) \* [dl\\_list\\_itr](#)  
*Typedef for list iterator.*

## Functions

- `dl_list dl_list_init` (int(\*cfunc)(const void \*, const void \*))  
*Initializes a new doubly linked list.*
- `void dl_list_free` (dl\_list list)  
*Frees the resources associated with a list.*
- `size_t dl_list_length` (const dl\_list list)  
*Returns the number of elements in a list.*
- `bool dl_list_isempty` (const dl\_list list)  
*Checks if a list is empty.*
- `void dl_list_prepend` (dl\_list list, void \*data)  
*Inserts an element at the beginning of a list.*
- `void dl_list_append` (dl\_list list, void \*data)  
*Inserts an element at the end of a list.*
- `int dl_list_insert_before` (dl\_list list, const dl\_list\_itr itr, void \*data)  
*Inserts an element before a provided iterator.*
- `int dl_list_insert_at` (dl\_list list, const size\_t index, void \*data)  
*Inserts an element at the specified index of a list.*
- `int dl_list_insert_after` (dl\_list list, const dl\_list\_itr itr, void \*data)  
*Inserts an element after a provided iterator.*
- `int dl_list_delete_at` (dl\_list list, const size\_t index)  
*Deletes a list element at a specified index.*
- `int dl_list_find_index` (const dl\_list list, const void \*data)  
*Finds the index of the specified data in a list.*
- `dl_list_itr dl_list_find_itr` (const dl\_list list, const void \*data)  
*Gets an iterator to the specified data in a list.*
- `void * dl_list_data` (const dl\_list list, const size\_t index)  
*Returns a pointer to the data at a specified index.*
- `dl_list_itr dl_list_first` (const dl\_list list)  
*Returns an iterator to the first element of a list.*
- `dl_list_itr dl_list_last` (const dl\_list list)  
*Returns an iterator to the last element of a list.*
- `dl_list_itr dl_list_next` (const dl\_list\_itr itr)  
*Advances a list iterator by one element.*
- `dl_list_itr dl_list_prev` (const dl\_list\_itr itr)  
*Backs up a list iterator by one element.*
- `dl_list_itr dl_list_itr_from_index` (const dl\_list list, const size\_t index)  
*Return an iterator to a specified element of a list.*

### 4.3.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

## 4.3.2 Function Documentation

### 4.3.2.1 void dl\_list\_append ( dl\_list list, void \* data )

#### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

### 4.3.2.2 void\* dl\_list\_data ( const dl\_list list, const size\_t index )

#### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

#### Returns

A pointer to the data, or NULL if the index is out of range.

### 4.3.2.3 int dl\_list\_delete\_at ( dl\_list list, const size\_t index )

#### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

#### Returns

0 on success, CDSERR\_OUTOFRANGE if the the index is out of range.

### 4.3.2.4 int dl\_list\_find\_index ( const dl\_list list, const void \* data )

#### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

#### Returns

The index of the element, if found, or CDSERR\_NOTFOUND if it is not in the list.

### 4.3.2.5 dl\_list\_itr dl\_list\_find\_itr ( const dl\_list list, const void \* data )

#### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

#### Returns

An iterator to the found element, or NULL is the element is not in the list.

#### 4.3.2.6 `dl_list_itr dl_list_first ( const dl_list list )`

##### Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

##### Returns

An iterator to the first element.

#### 4.3.2.7 `void dl_list_free ( dl_list list )`

##### Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

#### 4.3.2.8 `dl_list dl_list_init ( int(*)(const void *, const void *) cfunc )`

##### Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
--------------	---

##### Returns

A pointer to the new list.

#### 4.3.2.9 `int dl_list_insert_after ( dl_list list, const dl_list_itr itr, void * data )`

Note that `dl_list_first()` may return a NULL iterator when the list is empty. One reasonable behavior for this function would be to add a new node to the list in that case. However, an iterator may also become NULL when advanced to the end of the list. One possible way to modify this function would be to check the length of this list when the iterator is NULL, and if it is zero, add the first node to the list. However, the semantic meaning of adding an element *after* an iterator breaks down if that that iterator does not point to an existing element. Therefore, it is simpler for this function to simply refuse to handle NULL iterators. It is unlikely a user would want to call this function unless there are already elements in a list, and a valid iterator has been returned, e.g. through a find function.

##### Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

##### Returns

0 on success, CDSERR\_BADITERATOR if `itr` is a NULL pointer.

#### 4.3.2.10 `int dl_list_insert_at ( dl_list list, const size_t index, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_OUTOFRANGE if `index` exceeds the length of the list.

4.3.2.11 `int dl_list_insert_before ( dl_list list, const dl_list_itr itr, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_BADITERATOR if `itr` is a NULL pointer.

4.3.2.12 `bool dl_list_isempty ( const dl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

`true` if the list is empty, otherwise `false`.

4.3.2.13 `dl_list_itr dl_list_itr_from_index ( const dl_list list, const size_t index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

## Returns

The iterator, or NULL if `index` is out of range.

4.3.2.14 `dl_list_itr dl_list_last ( const dl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

**Returns**

An iterator to the first element.

**4.3.2.15** `size_t dl_list_length ( const dl_list list )`

**Parameters**

<i>list</i>	A pointer to the list.
-------------	------------------------

**4.3.2.16** `dl_list_itr dl_list_next ( const dl_list_itr itr )`

**Parameters**

<i>itr</i>	The iterator to advance
------------	-------------------------

**Returns**

The advanced iterator.

**4.3.2.17** `void dl_list_prepend ( dl_list list, void * data )`

**Parameters**

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

**4.3.2.18** `dl_list_itr dl_list_prev ( const dl_list_itr itr )`

**Parameters**

<i>itr</i>	The iterator to back up.
------------	--------------------------

**Returns**

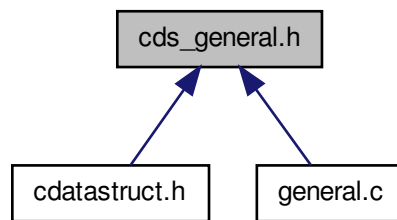
The backed up iterator.

## 4.4 cds\_general.h File Reference

Interface to general data structure helper functions.



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



## Functions

- void \* [cds\\_new\\_int](#) (const int n)  
*Dynamically allocates memory for a new int.*
- void \* [cds\\_new\\_uint](#) (const unsigned int n)  
*Dynamically allocates memory for a new unsigned int.*
- void \* [cds\\_new\\_long](#) (const long n)  
*Dynamically allocates memory for a new long.*
- void \* [cds\\_new\\_ulong](#) (const unsigned long n)  
*Dynamically allocates memory for a new unsigned long.*
- void \* [cds\\_new\\_string](#) (const char \*str)  
*Dynamically allocates memory for a new string.*
- int [cds\\_compare\\_int](#) (const void \*data, const void \*cmp)  
*Compares two int via void pointers.*
- int [cds\\_compare\\_uint](#) (const void \*data, const void \*cmp)  
*Compares two unsigned int via void pointers.*
- int [cds\\_compare\\_long](#) (const void \*data, const void \*cmp)  
*Compares two long via void pointers.*
- int [cds\\_compare\\_ulong](#) (const void \*data, const void \*cmp)  
*Compares two unsigned long via void pointers.*
- int [cds\\_compare\\_string](#) (const void \*data, const void \*cmp)  
*Compares two strings via void pointers.*

### 4.4.1 Detailed Description

Interface to general data structure helper functions.

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

## 4.4.2 Function Documentation

### 4.4.2.1 `int cds_compare_int ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.4.2.2 `int cds_compare_long ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.4.2.3 `int cds_compare_string ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.4.2.4 `int cds_compare_uint ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.4.2.5 `int cds_compare_ulong ( const void * data, const void * cmp )`

## Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

## Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.4.2.6 void\* cds\_new\_int ( const int *n* )

## Parameters

<i>n</i>	The new int for which to allocate.
----------	------------------------------------

## Returns

A void pointer to the allocated memory.

4.4.2.7 void\* cds\_new\_long ( const long *n* )

## Parameters

<i>n</i>	The new long for which to allocate.
----------	-------------------------------------

## Returns

A void pointer to the allocated memory.

4.4.2.8 void\* cds\_new\_string ( const char \* *str* )

## Parameters

<i>str</i>	The new string for which to allocate.
------------	---------------------------------------

## Returns

A void pointer to the allocated memory.

4.4.2.9 void\* cds\_new\_uint ( const unsigned int *n* )

## Parameters

<i>n</i>	The new unsigned int for which to allocate.
----------	---

## Returns

A void pointer to the allocated memory.

4.4.2.10 void\* cds\_new\_ulong ( const unsigned long *n* )

**Parameters**

<i>n</i>	The new unsigned long for which to allocate.
----------	--

**Returns**

A void pointer to the allocated memory.

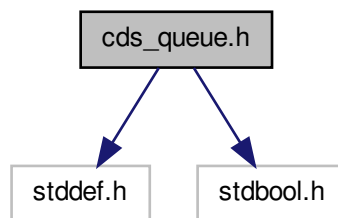
**4.5 cds\_queue.h File Reference**

User interface to queue data structure.

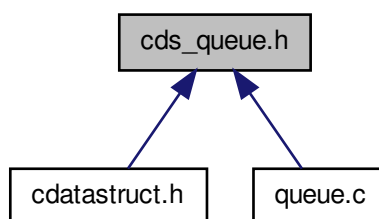
```
#include <stddef.h>
```

```
#include <stdbool.h>
```

Include dependency graph for cds\_queue.h:



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

**Typedefs**

- typedef struct dl\_list\_t \* queue  
*Typedef for queue pointer.*

**Functions**

- queue queue\_init (void)

- Initializes a new queue.*
- void `queue_free` (`queue` `que`)  
*Frees memory and releases resources used by a queue.*
- size\_t `queue_length` (const `queue` `que`)  
*Gets the number of items in a queue.*
- bool `queue_isempty` (const `queue` `que`)  
*Checks if a queue is empty.*
- void \* `queue_pop` (`queue` `que`)  
*Pops a data item from the queue.*
- void `queue_pushback` (`queue` `que`, void \*`data`)  
*Pushes a data item onto the back of the queue.*

### 4.5.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.5.2 Function Documentation

#### 4.5.2.1 void queue\_free ( queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

#### 4.5.2.2 queue queue\_init ( void )

##### Returns

A pointer to the new queue.

#### 4.5.2.3 bool queue\_isempty ( const queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

##### Returns

`true` is the queue is empty, `false` if not.

#### 4.5.2.4 size\_t queue\_length ( const queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

**Returns**

The number of items in the queue.

**4.5.2.5 void\* queue\_pop ( queue *que* )**

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

**Parameters**

<i>que</i>	A pointer to the queue.
------------	-------------------------

**Returns**

A `void` pointer to the popped data item.

**4.5.2.6 void queue\_pushback ( queue *que*, void \* *data* )**

The provided pointer should point to dynamically allocated memory.

**Parameters**

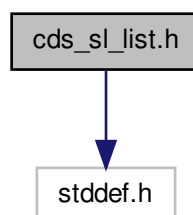
<i>que</i>	A pointer to the queue.
<i>data</i>	A pointer to the data item to be pushed.

**4.6 cds\_sl\_list.h File Reference**

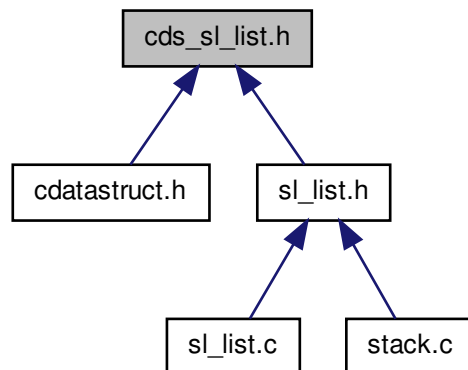
User interface to singly linked list data structure.

```
#include <stddef.h>
```

Include dependency graph for `cds_sl_list.h`:



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



## Data Structures

- struct [sl\\_list\\_node\\_t](#)  
*Struct for singly linked list node.*

## Typedefs

- typedef struct [sl\\_list\\_node\\_t](#) [sl\\_list\\_node\\_t](#)  
*Struct for singly linked list node.*
- typedef struct [sl\\_list\\_t](#) \* [sl\\_list](#)  
*Typedef for list pointer.*
- typedef struct [sl\\_list\\_node\\_t](#) \* [sl\\_list\\_itr](#)  
*Typedef for list iterator.*

## Functions

- [sl\\_list sl\\_list\\_init](#) (int(\*cfunc)(const void \*, const void \*))  
*Initializes a new singly linked list.*
- void [sl\\_list\\_free](#) ([sl\\_list](#) list)  
*Frees the resources associated with a list.*
- size\_t [sl\\_list\\_length](#) (const [sl\\_list](#) list)  
*Returns the number of elements in a list.*
- bool [sl\\_list\\_isempty](#) (const [sl\\_list](#) list)  
*Checks if a list is empty.*
- void [sl\\_list\\_prepend](#) ([sl\\_list](#) list, void \*data)  
*Inserts an element at the beginning of a list.*
- int [sl\\_list\\_insert\\_at](#) ([sl\\_list](#) list, const size\_t index, void \*data)  
*Inserts an element at the specified index of a list.*
- int [sl\\_list\\_insert\\_after](#) ([sl\\_list](#) list, const [sl\\_list\\_itr](#) itr, void \*data)  
*Inserts an element after a provided iterator.*

- `int sl_list_delete_at (sl_list list, const size_t index)`  
*Deletes a list element at a specified index.*
- `int sl_list_find_index (const sl_list list, const void *data)`  
*Gets an index to the specified data in a list.*
- `sl_list_itr sl_list_find_itr (const sl_list list, const void *data)`  
*Gets an iterator to the specified data in a list.*
- `void * sl_list_data (const sl_list list, const size_t index)`  
*Returns a pointer to the data at a specified index.*
- `sl_list_itr sl_list_first (const sl_list list)`  
*Returns an iterator to the first element of a list.*
- `sl_list_itr sl_list_next (const sl_list_itr itr)`  
*Advances a list iterator by one element.*
- `sl_list_itr sl_list_itr_from_index (const sl_list list, const size_t index)`  
*Return an iterator to a specified element of a list.*

#### 4.6.1 Detailed Description

##### Author

Paul Griffiths

##### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

#### 4.6.2 Function Documentation

##### 4.6.2.1 `void* sl_list_data ( const sl_list list, const size_t index )`

###### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

###### Returns

A pointer to the data, or NULL if the index is out of range.

##### 4.6.2.2 `int sl_list_delete_at ( sl_list list, const size_t index )`

###### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

###### Returns

0 on success, CDSERR\_OUTOFRANGE if the the index is out of range.

##### 4.6.2.3 `int sl_list_find_index ( const sl_list list, const void * data )`



## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

## Returns

The index of the found element, or CDSERR\_NOTFOUND if the element is not in the list.

4.6.2.4 `sl_list_itr sl_list_find_itr ( const sl_list list, const void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

## Returns

An iterator to the found element, or NULL if the element is not in the list.

4.6.2.5 `sl_list_itr sl_list_first ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

An iterator to the first element.

4.6.2.6 `void sl_list_free ( sl_list list )`

## Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

4.6.2.7 `sl_list sl_list_init ( int(*) (const void *, const void *) cfunc )`

## Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
--------------	---

## Returns

A pointer to the new list.

4.6.2.8 `int sl_list_insert_after ( sl_list list, const sl_list_itr itr, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_BADITERATOR if `itr` is a NULL pointer.

4.6.2.9 `int sl_list_insert_at ( sl_list list, const size_t index, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_OUTOFRANGE if `index` exceeds the length of the list.

4.6.2.10 `bool sl_list_isempty ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

`true` if the list is empty, otherwise `false`.

4.6.2.11 `sl_list_itr sl_list_itr_from_index ( const sl_list list, const size_t index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

## Returns

The iterator, or NULL if `index` is out of range.

4.6.2.12 `size_t sl_list_length ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

#### 4.6.2.13 `sl_list_itr sl_list_next ( const sl_list_itr itr )`

##### Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

##### Returns

The advanced iterator.

#### 4.6.2.14 `void sl_list_prepend ( sl_list list, void * data )`

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

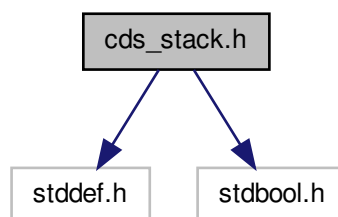
## 4.7 cds\_stack.h File Reference

User interface to stack data structure.

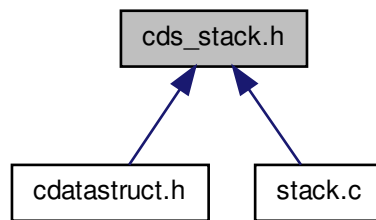
```
#include <stddef.h>
```

```
#include <stdbool.h>
```

Include dependency graph for cds\_stack.h:



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



## Typedefs

- typedef struct [sl\\_list\\_t](#) \* [stack](#)  
*Typedef for stack pointer.*

## Functions

- [stack\\_init](#) (void)  
*Initializes a new stack.*
- void [stack\\_free](#) ([stack](#) stk)  
*Frees memory and releases resources used by a stack.*
- size\_t [stack\\_length](#) (const [stack](#) stk)  
*Gets the number of items in a stack.*
- bool [stack\\_isempty](#) (const [stack](#) stk)  
*Checks if a stack is empty.*
- void \* [stack\\_pop](#) ([stack](#) stk)  
*Pops a data item from the stack.*
- void [stack\\_push](#) ([stack](#) stk, void \*data)  
*Pushes a data item onto the stack.*

### 4.7.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.7.2 Function Documentation

#### 4.7.2.1 void [stack\\_free](#) ( [stack](#) *stk* )

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

4.7.2.2 `stack stack_init ( void )`

## Returns

A pointer to the new stack.

4.7.2.3 `bool stack_isempty ( const stack stk )`

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

`true` is the stack is empty, `false` if not.

4.7.2.4 `size_t stack_length ( const stack stk )`

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

The number of items in the stack.

4.7.2.5 `void* stack_pop ( stack stk )`

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

A `void` pointer to the popped data item.

4.7.2.6 `void stack_push ( stack stk, void * data )`

The provided pointer should point to dynamically allocated memory.

## Parameters

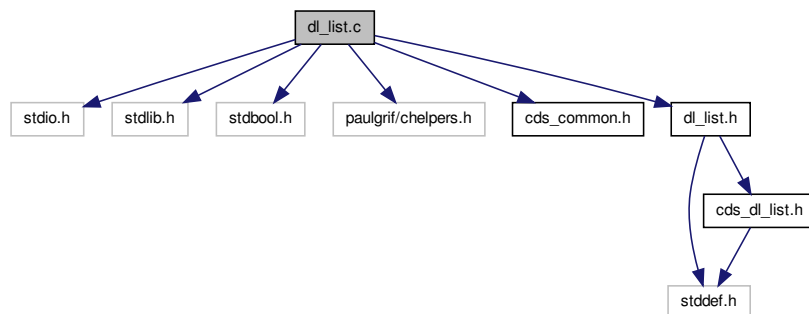
<i>stk</i>	A pointer to the stack.
<i>data</i>	A pointer to the data item to be pushed.

## 4.8 dl\_list.c File Reference

Implementation of doubly linked list data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "dl_list.h"
```

Include dependency graph for dl\_list.c:



## Functions

- [dl\\_list dl\\_list\\_init](#) (int(\*cfunc)(const void \*, const void \*))  
*Initializes a new doubly linked list.*
- void [dl\\_list\\_free](#) (dl\_list list)  
*Frees the resources associated with a list.*
- size\_t [dl\\_list\\_length](#) (const dl\_list list)  
*Returns the number of elements in a list.*
- bool [dl\\_list\\_isempty](#) (const dl\_list list)  
*Checks if a list is empty.*
- void [dl\\_list\\_prepend](#) (dl\_list list, void \*data)  
*Inserts an element at the beginning of a list.*
- void [dl\\_list\\_append](#) (dl\_list list, void \*data)  
*Inserts an element at the end of a list.*
- int [dl\\_list\\_insert\\_before](#) (dl\_list list, const dl\_list\_itr itr, void \*data)  
*Inserts an element before a provided iterator.*
- int [dl\\_list\\_insert\\_at](#) (dl\_list list, const size\_t index, void \*data)  
*Inserts an element at the specified index of a list.*
- int [dl\\_list\\_insert\\_after](#) (dl\_list list, const dl\_list\_itr itr, void \*data)  
*Inserts an element after a provided iterator.*
- int [dl\\_list\\_delete\\_at](#) (dl\_list list, const size\_t index)  
*Deletes a list element at a specified index.*
- int [dl\\_list\\_find\\_index](#) (const dl\_list list, const void \*data)  
*Finds the index of the specified data in a list.*
- [dl\\_list\\_itr dl\\_list\\_find\\_itr](#) (const dl\_list list, const void \*data)  
*Gets an iterator to the specified data in a list.*

- `void * dl_list_data` (const `dl_list` list, const `size_t` index)  
*Returns a pointer to the data at a specified index.*
- `dl_list_itr dl_list_first` (const `dl_list` list)  
*Returns an iterator to the first element of a list.*
- `dl_list_itr dl_list_last` (const `dl_list` list)  
*Returns an iterator to the last element of a list.*
- `dl_list_itr dl_list_next` (const `dl_list_itr` itr)  
*Advances a list iterator by one element.*
- `dl_list_itr dl_list_prev` (const `dl_list_itr` itr)  
*Backs up a list iterator by one element.*
- `dl_list_itr dl_list_itr_from_index` (const `dl_list` list, const `size_t` index)  
*Return an iterator to a specified element of a list.*
- `dl_list_node dl_list_new_node` (void \*data)  
*Creates a new list node.*
- `void dl_list_free_node` (`dl_list_node` node)  
*Frees resources for a node and any data.*
- `void dl_list_insert_node_front` (`dl_list` list, `dl_list_node` node)  
*Inserts a node at the front of a list.*
- `void dl_list_insert_node_before_mid` (`dl_list` list, `dl_list_itr` itr, `dl_list_node` node)  
*Inserts a node in the middle of a list before a specified iterator.*
- `void dl_list_insert_node_after_mid` (`dl_list` list, `dl_list_itr` itr, `dl_list_node` node)  
*Inserts a node in the middle of a list after a specified iterator.*
- `void dl_list_insert_node_back` (`dl_list` list, `dl_list_node` node)  
*Inserts a node at the back of a list.*
- `dl_list_node dl_list_remove_at` (`dl_list` list, const `size_t` index)  
*Removes, but does not delete, an element at an index.*
- `dl_list_node dl_list_remove_node_front` (`dl_list` list)  
*Removes the first node of a list.*
- `dl_list_node dl_list_remove_node_mid` (`dl_list` list, `dl_list_node` node)  
*Removes a specified node from the middle of a list.*
- `dl_list_node dl_list_remove_node_back` (`dl_list` list)  
*Removes the last node of a list.*
- `void dl_list_find` (const `dl_list` list, const void \*data, `dl_list_itr` \*p\_itr, int \*p\_index)  
*Finds the index of, and a pointer to, the first node in the list containing the specified data.*

### 4.8.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.8.2 Function Documentation

#### 4.8.2.1 void dl\_list\_append ( dl\_list list, void \* data )

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

#### 4.8.2.2 void\* dl\_list\_data ( const dl\_list list, const size\_t index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.

##### Returns

A pointer to the data, or NULL if the index is out of range.

#### 4.8.2.3 int dl\_list\_delete\_at ( dl\_list list, const size\_t index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

##### Returns

0 on success, CDSERR\_OUTOFRANGE if the the index is out of range.

#### 4.8.2.4 void dl\_list\_find ( const dl\_list list, const void \* data, dl\_list\_itr \* p\_itr, int \* p\_index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This is set to CDSERR_NOTFOUND if the data was not found.
<i>p_index</i>	A pointer to an integer the populate with the result. This is set to NULL if the data was not found.

#### 4.8.2.5 int dl\_list\_find\_index ( const dl\_list list, const void \* data )

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

##### Returns

The index of the element, if found, or CDSERR\_NOTFOUND if it is not in the list.

#### 4.8.2.6 dl\_list\_itr dl\_list\_find\_itr ( const dl\_list list, const void \* data )

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

##### Returns

An iterator to the found element, or NULL is the element is not in the list.



## 4.8.2.7 dl\_list\_itr dl\_list\_first ( const dl\_list list )

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

An iterator to the first element.

## 4.8.2.8 void dl\_list\_free ( dl\_list list )

## Parameters

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

## 4.8.2.9 void dl\_list\_free\_node ( dl\_list\_node node )

## Parameters

<i>node</i>	A pointer to the node to free.
-------------	--------------------------------

## 4.8.2.10 dl\_list dl\_list\_init ( int(\*)(const void \*, const void \*) cfunc )

## Parameters

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
--------------	---

## Returns

A pointer to the new list.

## 4.8.2.11 int dl\_list\_insert\_after ( dl\_list list, const dl\_list\_itr itr, void \* data )

Note that `dl_list_first()` may return a NULL iterator when the list is empty. One reasonable behavior for this function would be to add a new node to the list in that case. However, an iterator may also become NULL when advanced to the end of the list. One possible way to modify this function would be to check the length of this list when the iterator is NULL, and if it is zero, add the first node to the list. However, the semantic meaning of adding an element *after* an iterator breaks down if that that iterator does not point to an existing element. Therefore, it is simpler for this function to simply refuse to handle NULL iterators. It is unlikely a user would want to call this function unless there are already elements in a list, and a valid iterator has been returned, e.g. through a find function.

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_BADITERATOR if `itr` is a NULL pointer.

**4.8.2.12** `int dl_list_insert_at ( dl_list list, const size_t index, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_OUTOFRANGE if `index` exceeds the length of the list.

**4.8.2.13** `int dl_list_insert_before ( dl_list list, const dl_list_itr itr, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, CDSERR\_BADITERATOR if `itr` is a NULL pointer.

**4.8.2.14** `void dl_list_insert_node_after_mid ( dl_list list, dl_list_itr itr, dl_list_node node )`

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

**4.8.2.15** `void dl_list_insert_node_back ( dl_list list, dl_list_node node )`

## Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

**4.8.2.16** `void dl_list_insert_node_before_mid ( dl_list list, dl_list_itr itr, dl_list_node node )`

## Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator before which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

4.8.2.17 void dl\_list\_insert\_node\_front ( dl\_list *list*, dl\_list\_node *node* )

## Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

4.8.2.18 bool dl\_list\_isempty ( const dl\_list *list* )

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

`true` if the list is empty, otherwise `false`.

4.8.2.19 dl\_list\_itr dl\_list\_itr\_from\_index ( const dl\_list *list*, const size\_t *index* )

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

## Returns

The iterator, or NULL if `index` is out of range.

4.8.2.20 dl\_list\_itr dl\_list\_last ( const dl\_list *list* )

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

An iterator to the first element.

4.8.2.21 size\_t dl\_list\_length ( const dl\_list *list* )

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

#### 4.8.2.22 `dl_list_node dl_list_new_node ( void * data )`

##### Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

##### Returns

A pointer to the newly created node.

#### 4.8.2.23 `dl_list_itr dl_list_next ( const dl_list_itr itr )`

##### Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

##### Returns

The advanced iterator.

#### 4.8.2.24 `void dl_list_prepend ( dl_list list, void * data )`

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

#### 4.8.2.25 `dl_list_itr dl_list_prev ( const dl_list_itr itr )`

##### Parameters

<i>itr</i>	The iterator to back up.
------------	--------------------------

##### Returns

The backed up iterator.

#### 4.8.2.26 `dl_list_node dl_list_remove_at ( dl_list list, const size_t index )`

##### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

##### Returns

A pointer to the removed node. This should be `free()`d by calling [dl\\_list\\_free\\_node\(\)](#).

#### 4.8.2.27 `dl_list_node dl_list_remove_node_back ( dl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

A pointer to the removed node.

## 4.8.2.28 dl\_list\_node dl\_list\_remove\_node\_front ( dl\_list list )

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

A pointer to the removed node.

## 4.8.2.29 dl\_list\_node dl\_list\_remove\_node\_mid ( dl\_list list, dl\_list\_node node )

## Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to remove. As this is removing from the middle, this node should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.

## Returns

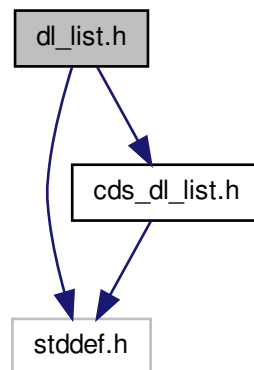
A pointer to the removed node, i.e. equal to `itr`.

## 4.9 dl\_list.h File Reference

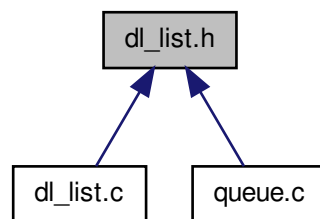
Developer interface to double linked list data structure.

```
#include <stddef.h>
#include "cds_dl_list.h"
```

Include dependency graph for `dl_list.h`:



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



## Data Structures

- struct [dl\\_list\\_t](#)  
*Struct to contain a list.*

## Typedefs

- typedef struct [dl\\_list\\_t](#) [dl\\_list\\_t](#)  
*Struct to contain a list.*
- typedef struct [dl\\_list\\_node\\_t](#) \* [dl\\_list\\_node](#)  
*Typedef for list node.*

## Functions

- [dl\\_list\\_node](#) [dl\\_list\\_new\\_node](#) (void \*data)

- Creates a new list node.*
- void `dl_list_free_node` (`dl_list_node` node)
- Frees resources for a node and any data.*
- void `dl_list_insert_node_front` (`dl_list` list, `dl_list_node` node)
- Inserts a node at the front of a list.*
- void `dl_list_insert_node_before_mid` (`dl_list` list, `dl_list_itr` itr, `dl_list_node` node)
- Inserts a node in the middle of a list before a specified iterator.*
- void `dl_list_insert_node_after_mid` (`dl_list` list, `dl_list_itr` itr, `dl_list_node` node)
- Inserts a node in the middle of a list after a specified iterator.*
- void `dl_list_insert_node_back` (`dl_list` list, `dl_list_node` node)
- Inserts a node at the back of a list.*
- `dl_list_node` `dl_list_remove_at` (`dl_list` list, const `size_t` index)
- Removes, but does not delete, an element at an index.*
- `dl_list_node` `dl_list_remove_node_front` (`dl_list` list)
- Removes the first node of a list.*
- `dl_list_node` `dl_list_remove_node_mid` (`dl_list` list, `dl_list_itr` itr)
- Removes a specified node from the middle of a list.*
- `dl_list_node` `dl_list_remove_node_back` (`dl_list` list)
- Removes the last node of a list.*
- void `dl_list_find` (const `dl_list` list, const void \*data, `dl_list_itr` \*p\_itr, int \*p\_index)
- Finds the index of, and a pointer to, the first node in the list containing the specified data.*

### 4.9.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.9.2 Function Documentation

#### 4.9.2.1 void dl\_list\_find ( const dl\_list list, const void \* data, dl\_list\_itr \* p\_itr, int \* p\_index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This is set to CDSERR_NOTFOUND if the data was not found.
<i>p_index</i>	A pointer to an integer the populate with the result. This is set to NULL if the data was not found.

#### 4.9.2.2 void dl\_list\_free\_node ( dl\_list\_node node )

##### Parameters

<i>node</i>	A pointer to the node to free.
-------------	--------------------------------

#### 4.9.2.3 void dl\_list\_insert\_node\_after\_mid ( dl\_list list, dl\_list\_itr itr, dl\_list\_node node )

##### Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

#### 4.9.2.4 void dl\_list\_insert\_node\_back ( dl\_list list, dl\_list\_node node )

##### Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

#### 4.9.2.5 void dl\_list\_insert\_node\_before\_mid ( dl\_list list, dl\_list\_itr itr, dl\_list\_node node )

##### Parameters

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator before which to insert. As this is inserting in the middle, this iterator should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.
<i>node</i>	A pointer to the node to insert.

#### 4.9.2.6 void dl\_list\_insert\_node\_front ( dl\_list list, dl\_list\_node node )

##### Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.

#### 4.9.2.7 dl\_list\_node dl\_list\_new\_node ( void \* data )

##### Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

##### Returns

A pointer to the newly created node.

#### 4.9.2.8 dl\_list\_node dl\_list\_remove\_at ( dl\_list list, const size\_t index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.



**Returns**

A pointer to the removed node. This should be `free()` d by calling `dl_list_free_node()`.

**4.9.2.9 `dl_list_node dl_list_remove_node_back ( dl_list list )`****Parameters**

<i>list</i>	A pointer to the list.
-------------	------------------------

**Returns**

A pointer to the removed node.

**4.9.2.10 `dl_list_node dl_list_remove_node_front ( dl_list list )`****Parameters**

<i>list</i>	A pointer to the list.
-------------	------------------------

**Returns**

A pointer to the removed node.

**4.9.2.11 `dl_list_node dl_list_remove_node_mid ( dl_list list, dl_list_node node )`****Parameters**

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to remove. As this is removing from the middle, this node should not be either the front or the back of the list, i.e. both the <code>prev</code> and <code>next</code> members should be non-NULL.

**Returns**

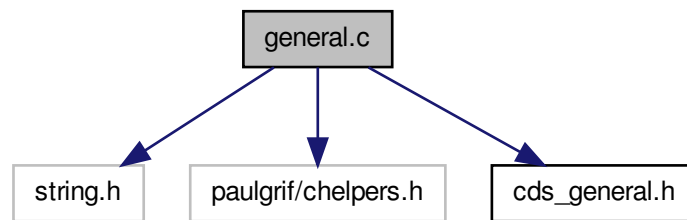
A pointer to the removed node, i.e. equal to `itr`.

**4.10 general.c File Reference**

Implementation of general data structure helper functions.

```
#include <string.h>
#include <paulgrif/chelpers.h>
#include "cds_general.h"
```

Include dependency graph for general.c:



## Functions

- void \* `cds_new_int` (const int n)  
*Dynamically allocates memory for a new int.*
- void \* `cds_new_uint` (const unsigned int n)  
*Dynamically allocates memory for a new unsigned int.*
- void \* `cds_new_long` (const long n)  
*Dynamically allocates memory for a new long.*
- void \* `cds_new_ulong` (const unsigned long n)  
*Dynamically allocates memory for a new unsigned long.*
- void \* `cds_new_string` (const char \*str)  
*Dynamically allocates memory for a new string.*
- int `cds_compare_int` (const void \*data, const void \*cmp)  
*Compares two int via void pointers.*
- int `cds_compare_uint` (const void \*data, const void \*cmp)  
*Compares two unsigned int via void pointers.*
- int `cds_compare_long` (const void \*data, const void \*cmp)  
*Compares two long via void pointers.*
- int `cds_compare_ulong` (const void \*data, const void \*cmp)  
*Compares two unsigned long via void pointers.*
- int `cds_compare_string` (const void \*data, const void \*cmp)  
*Compares two strings via void pointers.*

### 4.10.1 Detailed Description

Implementation of general data structure helper functions.

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

## 4.10.2 Function Documentation

### 4.10.2.1 `int cds_compare_int ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.10.2.2 `int cds_compare_long ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.10.2.3 `int cds_compare_string ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.10.2.4 `int cds_compare_uint ( const void * data, const void * cmp )`

#### Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

#### Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

### 4.10.2.5 `int cds_compare_ulong ( const void * data, const void * cmp )`

## Parameters

<i>data</i>	Pointer to the data to which to compare.
<i>cmp</i>	Pointer to the comparison data.

## Returns

-1 if the comparison data is greater than the data, 1 if the comparison data is less than the data, and 0 if the comparison data is equal to the data.

4.10.2.6 `void* cds_new_int ( const int n )`

## Parameters

<i>n</i>	The new <code>int</code> for which to allocate.
----------	---

## Returns

A `void` pointer to the allocated memory.

4.10.2.7 `void* cds_new_long ( const long n )`

## Parameters

<i>n</i>	The new <code>long</code> for which to allocate.
----------	--

## Returns

A `void` pointer to the allocated memory.

4.10.2.8 `void* cds_new_string ( const char * str )`

## Parameters

<i>str</i>	The new string for which to allocate.
------------	---------------------------------------

## Returns

A `void` pointer to the allocated memory.

4.10.2.9 `void* cds_new_uint ( const unsigned int n )`

## Parameters

<i>n</i>	The new unsigned <code>int</code> for which to allocate.
----------	--

## Returns

A `void` pointer to the allocated memory.

4.10.2.10 `void* cds_new_ulong ( const unsigned long n )`

## Parameters

<i>n</i>	The new unsigned long for which to allocate.
----------	--

## Returns

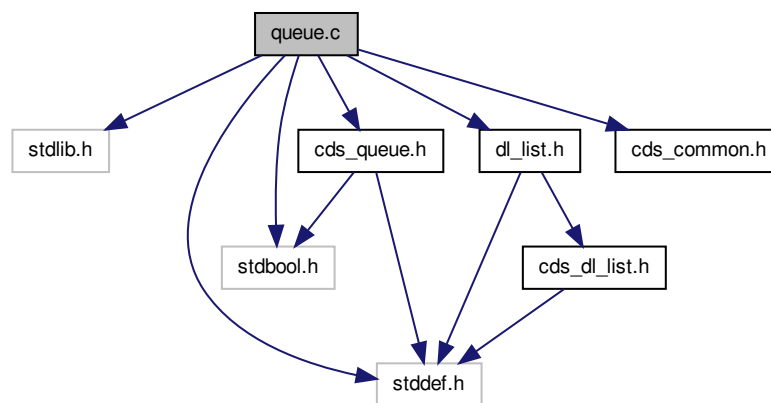
A void pointer to the allocated memory.

## 4.11 queue.c File Reference

Implementation of queue data structure.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include "cds_queue.h"
#include "dl_list.h"
#include "cds_common.h"
```

Include dependency graph for queue.c:



## Functions

- `queue queue_init` (void)  
*Initializes a new queue.*
- `void queue_free` (queue que)  
*Frees memory and releases resources used by a queue.*
- `size_t queue_length` (const queue que)  
*Gets the number of items in a queue.*
- `bool queue_isempty` (const queue que)  
*Checks if a queue is empty.*
- `void * queue_pop` (queue que)  
*Pops a data item from the queue.*
- `void queue_pushback` (queue que, void \*data)  
*Pushes a data item onto the back of the queue.*

### 4.11.1 Detailed Description

Implemented in terms of a doubly linked, double-ended list data structure.

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.11.2 Function Documentation

#### 4.11.2.1 void queue\_free ( queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

#### 4.11.2.2 queue queue\_init ( void )

##### Returns

A pointer to the new queue.

#### 4.11.2.3 bool queue\_isempty ( const queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

##### Returns

`true` if the queue is empty, `false` if not.

#### 4.11.2.4 size\_t queue\_length ( const queue *que* )

##### Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

##### Returns

The number of items in the queue.

#### 4.11.2.5 void\* queue\_pop ( queue *que* )

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

## Parameters

<i>que</i>	A pointer to the queue.
------------	-------------------------

## Returns

A `void` pointer to the popped data item.

4.11.2.6 void queue\_pushback ( queue *que*, void \* *data* )

The provided pointer should point to dynamically allocated memory.

## Parameters

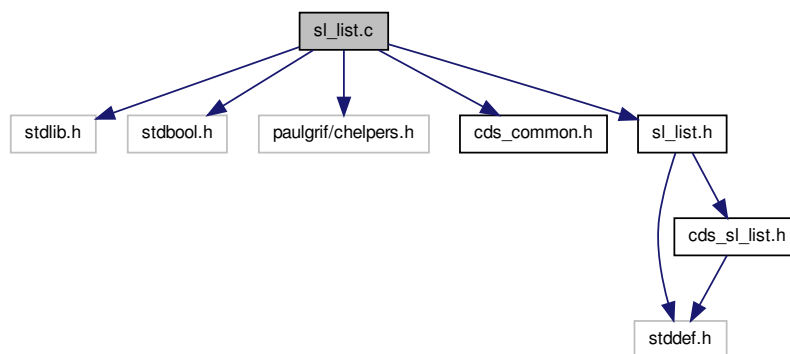
<i>que</i>	A pointer to the queue.
<i>data</i>	A pointer to the data item to be pushed.

## 4.12 sl\_list.c File Reference

Implementation of singly linked list data structure.

```
#include <stdlib.h>
#include <stdbool.h>
#include <paulgrif/chelpers.h>
#include "cds_common.h"
#include "sl_list.h"
```

Include dependency graph for `sl_list.c`:



## Functions

- `sl_list sl_list_init` (int(\*cfunc)(const void \*, const void \*))  
*Initializes a new singly linked list.*
- void `sl_list_free` (sl\_list list)  
*Frees the resources associated with a list.*
- size\_t `sl_list_length` (const sl\_list list)  
*Returns the number of elements in a list.*
- bool `sl_list_isempty` (const sl\_list list)

- Checks if a list is empty.*

  - void `sl_list_prepend` (`sl_list` list, void \*data)

*Inserts an element at the beginning of a list.*
- int `sl_list_insert_at` (`sl_list` list, const size\_t index, void \*data)

*Inserts an element at the specified index of a list.*
- int `sl_list_insert_after` (`sl_list` list, const `sl_list_itr` itr, void \*data)

*Inserts an element after a provided iterator.*
- int `sl_list_delete_at` (`sl_list` list, const size\_t index)

*Deletes a list element at a specified index.*
- int `sl_list_find_index` (const `sl_list` list, const void \*data)

*Gets an index to the specified data in a list.*
- `sl_list_itr` `sl_list_find_itr` (const `sl_list` list, const void \*data)

*Gets an iterator to the specified data in a list.*
- void \* `sl_list_data` (const `sl_list` list, const size\_t index)

*Returns a pointer to the data at a specified index.*
- `sl_list_itr` `sl_list_first` (const `sl_list` list)

*Returns an iterator to the first element of a list.*
- `sl_list_itr` `sl_list_next` (const `sl_list_itr` itr)

*Advances a list iterator by one element.*
- `sl_list_itr` `sl_list_itr_from_index` (const `sl_list` list, const size\_t index)

*Return an iterator to a specified element of a list.*
- `sl_list_node` `sl_list_new_node` (void \*data)

*Creates a new list node.*
- void `sl_list_free_node` (`sl_list_node` node)

*Frees resources for a node and any data.*
- `sl_list_node` `sl_list_remove_at` (`sl_list` list, const size\_t index)

*Removes, but does not delete, an element at an index.*
- void `sl_list_find` (const `sl_list` list, const void \*data, `sl_list_itr` \*p\_itr, int \*p\_index)

*Gets an index and iterator to a specified piece of data.*

#### 4.12.1 Detailed Description

##### Author

Paul Griffiths

##### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

#### 4.12.2 Function Documentation

##### 4.12.2.1 void\* sl\_list\_data ( const sl\_list list, const size\_t index )

##### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the data.



## Returns

A pointer to the data, or NULL if the index is out of range.

4.12.2.2 `int sl_list_delete_at ( sl_list list, const size_t index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to delete.

## Returns

0 on success, CDSERR\_OUTOFRANGE if the the index is out of range.

4.12.2.3 `void sl_list_find ( const sl_list list, const void * data, sl_list_itr * p_itr, int * p_index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This parameter is ignored if set to NULL.
<i>p_index</i>	A pointer to an integer index to populate with the result. This parameter is ignored if set to NULL.

4.12.2.4 `int sl_list_find_index ( const sl_list list, const void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

## Returns

The index of the found element, or CDSERR\_NOTFOUND if the element is not in the list.

4.12.2.5 `sl_list_itr sl_list_find_itr ( const sl_list list, const void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.

## Returns

An iterator to the found element, or NULL is the element is not in the list.

4.12.2.6 `sl_list_itr sl_list_first ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

**Returns**

An iterator to the first element.

**4.12.2.7 void sl\_list\_free ( sl\_list list )****Parameters**

<i>list</i>	A pointer to the list to free.
-------------	--------------------------------

**4.12.2.8 void sl\_list\_free\_node ( sl\_list\_node node )****Parameters**

<i>node</i>	A pointer to the node to free.
-------------	--------------------------------

**4.12.2.9 sl\_list sl\_list\_init ( int (\*)(const void \*, const void \*) cfunc )****Parameters**

<i>cfunc</i>	A pointer to a compare function. The function should return <code>int</code> and accept two parameters of type <code>void *</code> . It should return less than 1 if the first parameter is less than the second, greater than 1 if the first parameter is greater than the second, and zero if the parameters are equal.
--------------	---

**Returns**

A pointer to the new list.

**4.12.2.10 int sl\_list\_insert\_after ( sl\_list list, const sl\_list\_itr itr, void \* data )****Parameters**

<i>list</i>	A pointer to the list.
<i>itr</i>	The iterator after which to insert.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

**Returns**

0 on success, `CDSERR_BADITERATOR` if `itr` is a NULL pointer.

**4.12.2.11 int sl\_list\_insert\_at ( sl\_list list, const size\_t index, void \* data )****Parameters**

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert. Setting this equal to the length of the list (i.e. to one element past the zero-based index of the last element) inserts the element at the end of the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

## Returns

0 on success, `CDSERR_OUTOFRANGE` if `index` exceeds the length of the list.

4.12.2.12 `bool sl_list_isempty ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

## Returns

`true` if the list is empty, otherwise `false`.

4.12.2.13 `sl_list_itr sl_list_itr_from_index ( const sl_list list, const size_t index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The specified index.

## Returns

The iterator, or `NULL` if `index` is out of range.

4.12.2.14 `size_t sl_list_length ( const sl_list list )`

## Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

4.12.2.15 `sl_list_node sl_list_new_node ( void * data )`

## Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

## Returns

A pointer to the newly created node.

4.12.2.16 `sl_list_itr sl_list_next ( const sl_list_itr itr )`

## Parameters

<i>itr</i>	The iterator to advance
------------	-------------------------

## Returns

The advanced iterator.

4.12.2.17 `void sl_list_prepend ( sl_list list, void * data )`

## Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to add. The memory pointed to by this parameter must be dynamically allocated, as an attempt will be made to <code>free()</code> it when deleting the list.

4.12.2.18 `sl_list_node sl_list_remove.at ( sl_list list, const size_t index )`

## Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

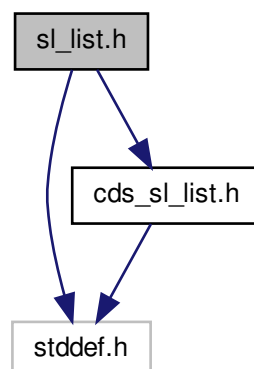
## Returns

A pointer to the removed node. This should be `free()` d by calling [sl\\_list\\_free\\_node\(\)](#).

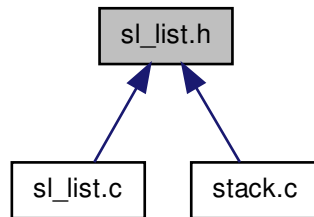
4.13 `sl_list.h` File Reference

Developer interface to singly linked list data structure.

```
#include <stddef.h>
#include "cds_sl_list.h"
Include dependency graph for sl_list.h:
```



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



## Data Structures

- struct `sl_list_t`  
*Struct to contain a list.*

## Typedefs

- typedef struct `sl_list_t` `sl_list_t`  
*Struct to contain a list.*
- typedef struct `sl_list_node_t` \* `sl_list_node`  
*Typedef for list node.*

## Functions

- `sl_list_node` `sl_list_new_node` (void \*data)  
*Creates a new list node.*
- void `sl_list_free_node` (`sl_list_node` node)  
*Frees resources for a node and any data.*
- `sl_list_node` `sl_list_remove_at` (`sl_list` list, const size\_t index)  
*Removes, but does not delete, an element at an index.*
- void `sl_list_find` (const `sl_list` list, const void \*data, `sl_list_itr` \*p\_itr, int \*p\_index)  
*Gets an index and iterator to a specified piece of data.*

### 4.13.1 Detailed Description

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.13.2 Function Documentation

4.13.2.1 `void sl_list_find ( const sl_list list, const void * data, sl_list_itr * p_itr, int * p_index )`

#### Parameters

<i>list</i>	A pointer to the list.
<i>data</i>	A pointer to the data to find.
<i>p_itr</i>	A pointer to an iterator to populate with the result. This parameter is ignored if set to NULL.
<i>p_index</i>	A pointer to an integer index to populate with the result. This parameter is ignored if set to NULL.

4.13.2.2 `void sl_list_free_node ( sl_list_node node )`

#### Parameters

<i>node</i>	A pointer to the node to free.
-------------	--------------------------------

4.13.2.3 `sl_list_node sl_list_new_node ( void * data )`

#### Parameters

<i>data</i>	The data for the new node.
-------------	----------------------------

#### Returns

A pointer to the newly created node.

4.13.2.4 `sl_list_node sl_list_remove_at ( sl_list list, const size_t index )`

#### Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the element to be removed.

#### Returns

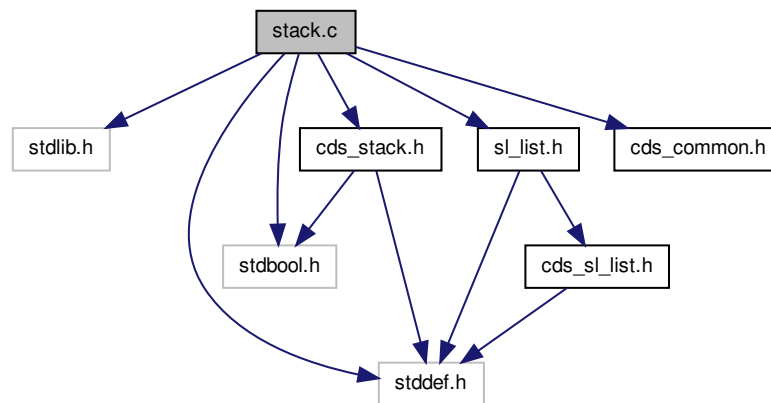
A pointer to the removed node. This should be `free()` d by calling [sl\\_list\\_free\\_node\(\)](#).

## 4.14 stack.c File Reference

Implementation of stack data structure.

```
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include "cds_stack.h"
#include "sl_list.h"
#include "cds_common.h"
```

Include dependency graph for stack.c:



## Functions

- `stack stack_init` (void)  
*Initializes a new stack.*
- void `stack_free` (stack stk)  
*Frees memory and releases resources used by a stack.*
- size\_t `stack_length` (const stack stk)  
*Gets the number of items in a stack.*
- bool `stack_isempty` (const stack stk)  
*Checks if a stack is empty.*
- void \* `stack_pop` (stack stk)  
*Pops a data item from the stack.*
- void `stack_push` (stack stk, void \*data)  
*Pushes a data item onto the stack.*

### 4.14.1 Detailed Description

Implemented in terms of a singly linked, singled-ended list data structure.

#### Author

Paul Griffiths

#### Copyright

Copyright 2013 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

### 4.14.2 Function Documentation

#### 4.14.2.1 void stack\_free ( stack stk )

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

4.14.2.2 `stack stack_init ( void )`

## Returns

A pointer to the new stack.

4.14.2.3 `bool stack_isempty ( const stack stk )`

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

`true` is the stack is empty, `false` if not.

4.14.2.4 `size_t stack_length ( const stack stk )`

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

The number of items in the stack.

4.14.2.5 `void* stack_pop ( stack stk )`

The item returned was previously allocated using `malloc()`, so the user must `free()` the returned pointer when done.

## Parameters

<i>stk</i>	A pointer to the stack.
------------	-------------------------

## Returns

A `void` pointer to the popped data item.

4.14.2.6 `void stack_push ( stack stk, void * data )`

The provided pointer should point to dynamically allocated memory.

## Parameters

<i>stk</i>	A pointer to the stack.
<i>data</i>	A pointer to the data item to be pushed.



# Index

back

dl\_list\_t, 6

CDSERR\_BADITERATOR

cds\_common.h, 10

CDSERR\_ERROR

cds\_common.h, 10

CDSERR\_NOTFOUND

cds\_common.h, 10

CDSERR\_OUTOFRANGE

cds\_common.h, 10

cdatastruct.h, 9

cds\_common.h

CDSERR\_BADITERATOR, 10

CDSERR\_ERROR, 10

CDSERR\_NOTFOUND, 10

CDSERR\_OUTOFRANGE, 10

cds\_common.h, 9

cds\_error, 10

cds\_compare\_int

cds\_general.h, 18

general.c, 43

cds\_compare\_long

cds\_general.h, 18

general.c, 43

cds\_compare\_string

cds\_general.h, 18

general.c, 43

cds\_compare\_uint

cds\_general.h, 18

general.c, 43

cds\_compare\_ulong

cds\_general.h, 18

general.c, 43

cds\_dl\_list.h, 10

dl\_list\_append, 13

dl\_list\_data, 13

dl\_list\_delete\_at, 13

dl\_list\_find\_index, 13

dl\_list\_find\_itr, 13

dl\_list\_first, 13

dl\_list\_free, 14

dl\_list\_init, 14

dl\_list\_insert\_after, 14

dl\_list\_insert\_at, 14

dl\_list\_insert\_before, 15

dl\_list\_isempty, 15

dl\_list\_itr\_from\_index, 15

dl\_list\_last, 15

dl\_list\_length, 16

dl\_list\_next, 16

dl\_list\_prepend, 16

dl\_list\_prev, 16

cds\_error

cds\_common.h, 10

cds\_general.h, 16

cds\_compare\_int, 18

cds\_compare\_long, 18

cds\_compare\_string, 18

cds\_compare\_uint, 18

cds\_compare\_ulong, 18

cds\_new\_int, 19

cds\_new\_long, 19

cds\_new\_string, 19

cds\_new\_uint, 19

cds\_new\_ulong, 19

cds\_new\_int

cds\_general.h, 19

general.c, 44

cds\_new\_long

cds\_general.h, 19

general.c, 44

cds\_new\_string

cds\_general.h, 19

general.c, 44

cds\_new\_uint

cds\_general.h, 19

general.c, 44

cds\_new\_ulong

cds\_general.h, 19

general.c, 44

cds\_queue.h, 20

queue\_free, 21

queue\_init, 21

queue\_isempty, 21

queue\_length, 21

queue\_pop, 22

queue\_pushback, 22

cds\_sl\_list.h, 22

sl\_list\_data, 24

sl\_list\_delete\_at, 24

sl\_list\_find\_index, 24

sl\_list\_find\_itr, 25

sl\_list\_first, 25

sl\_list\_free, 25

sl\_list\_init, 25

sl\_list\_insert\_after, 25

sl\_list\_insert\_at, 26

sl\_list\_isempty, 26

- sl\_list\_itr\_from\_index, 26
- sl\_list\_length, 26
- sl\_list\_next, 26
- sl\_list\_prepend, 27
- cds\_stack.h, 27
  - stack\_free, 28
  - stack\_init, 29
  - stack\_isempty, 29
  - stack\_length, 29
  - stack\_pop, 29
  - stack\_push, 29
- cfunc
  - dl\_list\_t, 6
  - sl\_list\_t, 8
- data
  - dl\_list\_node\_t, 5
  - sl\_list\_node\_t, 7
- dl\_list.c, 30
  - dl\_list\_append, 31
  - dl\_list\_data, 32
  - dl\_list\_delete\_at, 32
  - dl\_list\_find, 32
  - dl\_list\_find\_index, 32
  - dl\_list\_find\_itr, 32
  - dl\_list\_first, 32
  - dl\_list\_free, 33
  - dl\_list\_free\_node, 33
  - dl\_list\_init, 33
  - dl\_list\_insert\_after, 33
  - dl\_list\_insert\_at, 34
  - dl\_list\_insert\_before, 34
  - dl\_list\_insert\_node\_after\_mid, 34
  - dl\_list\_insert\_node\_back, 34
  - dl\_list\_insert\_node\_before\_mid, 34
  - dl\_list\_insert\_node\_front, 35
  - dl\_list\_isempty, 35
  - dl\_list\_itr\_from\_index, 35
  - dl\_list\_last, 35
  - dl\_list\_length, 35
  - dl\_list\_new\_node, 35
  - dl\_list\_next, 36
  - dl\_list\_prepend, 36
  - dl\_list\_prev, 36
  - dl\_list\_remove\_at, 36
  - dl\_list\_remove\_node\_back, 36
  - dl\_list\_remove\_node\_front, 37
  - dl\_list\_remove\_node\_mid, 37
- dl\_list.h, 37
  - dl\_list\_find, 39
  - dl\_list\_free\_node, 39
  - dl\_list\_insert\_node\_after\_mid, 39
  - dl\_list\_insert\_node\_back, 40
  - dl\_list\_insert\_node\_before\_mid, 40
  - dl\_list\_insert\_node\_front, 40
  - dl\_list\_new\_node, 40
  - dl\_list\_remove\_at, 40
  - dl\_list\_remove\_node\_back, 41
  - dl\_list\_remove\_node\_front, 41
  - dl\_list\_remove\_node\_mid, 41
  - dl\_list\_remove\_node\_mid, 41
- dl\_list\_append
  - cds\_dl\_list.h, 13
  - dl\_list.c, 31
- dl\_list\_data
  - cds\_dl\_list.h, 13
  - dl\_list.c, 32
- dl\_list\_delete\_at
  - cds\_dl\_list.h, 13
  - dl\_list.c, 32
- dl\_list\_find
  - dl\_list.c, 32
  - dl\_list.h, 39
- dl\_list\_find\_index
  - cds\_dl\_list.h, 13
  - dl\_list.c, 32
- dl\_list\_find\_itr
  - cds\_dl\_list.h, 13
  - dl\_list.c, 32
- dl\_list\_first
  - cds\_dl\_list.h, 13
  - dl\_list.c, 32
- dl\_list\_free
  - cds\_dl\_list.h, 14
  - dl\_list.c, 33
- dl\_list\_free\_node
  - dl\_list.c, 33
  - dl\_list.h, 39
- dl\_list\_init
  - cds\_dl\_list.h, 14
  - dl\_list.c, 33
- dl\_list\_insert\_after
  - cds\_dl\_list.h, 14
  - dl\_list.c, 33
- dl\_list\_insert\_at
  - cds\_dl\_list.h, 14
  - dl\_list.c, 34
- dl\_list\_insert\_before
  - cds\_dl\_list.h, 15
  - dl\_list.c, 34
- dl\_list\_insert\_node\_after\_mid
  - dl\_list.c, 34
  - dl\_list.h, 39
- dl\_list\_insert\_node\_back
  - dl\_list.c, 34
  - dl\_list.h, 40
- dl\_list\_insert\_node\_before\_mid
  - dl\_list.c, 34
  - dl\_list.h, 40
- dl\_list\_insert\_node\_front
  - dl\_list.c, 35
  - dl\_list.h, 40
- dl\_list\_isempty
  - cds\_dl\_list.h, 15
  - dl\_list.c, 35
- dl\_list\_itr\_from\_index
  - cds\_dl\_list.h, 15
  - dl\_list.c, 35

- dl\_list\_last
  - cds\_dl\_list.h, [15](#)
  - dl\_list.c, [35](#)
- dl\_list\_length
  - cds\_dl\_list.h, [16](#)
  - dl\_list.c, [35](#)
- dl\_list\_new\_node
  - dl\_list.c, [35](#)
  - dl\_list.h, [40](#)
- dl\_list\_next
  - cds\_dl\_list.h, [16](#)
  - dl\_list.c, [36](#)
- dl\_list\_node\_t, [5](#)
  - data, [5](#)
  - next, [5](#)
  - prev, [5](#)
- dl\_list\_prepend
  - cds\_dl\_list.h, [16](#)
  - dl\_list.c, [36](#)
- dl\_list\_prev
  - cds\_dl\_list.h, [16](#)
  - dl\_list.c, [36](#)
- dl\_list\_remove\_at
  - dl\_list.c, [36](#)
  - dl\_list.h, [40](#)
- dl\_list\_remove\_node\_back
  - dl\_list.c, [36](#)
  - dl\_list.h, [41](#)
- dl\_list\_remove\_node\_front
  - dl\_list.c, [37](#)
  - dl\_list.h, [41](#)
- dl\_list\_remove\_node\_mid
  - dl\_list.c, [37](#)
  - dl\_list.h, [41](#)
- dl\_list\_t, [6](#)
  - back, [6](#)
  - cfunc, [6](#)
  - front, [6](#)
  - length, [6](#)
- front
  - dl\_list\_t, [6](#)
  - sl\_list\_t, [8](#)
- general.c, [41](#)
  - cds\_compare\_int, [43](#)
  - cds\_compare\_long, [43](#)
  - cds\_compare\_string, [43](#)
  - cds\_compare\_uint, [43](#)
  - cds\_compare\_ulong, [43](#)
  - cds\_new\_int, [44](#)
  - cds\_new\_long, [44](#)
  - cds\_new\_string, [44](#)
  - cds\_new\_uint, [44](#)
  - cds\_new\_ulong, [44](#)
- length
  - dl\_list\_t, [6](#)
  - sl\_list\_t, [8](#)
- next
  - dl\_list\_node\_t, [5](#)
  - sl\_list\_node\_t, [7](#)
- prev
  - dl\_list\_node\_t, [5](#)
- queue.c, [45](#)
  - queue\_free, [46](#)
  - queue\_init, [46](#)
  - queue\_isempty, [46](#)
  - queue\_length, [46](#)
  - queue\_pop, [46](#)
  - queue\_pushback, [47](#)
- queue\_free
  - cds\_queue.h, [21](#)
  - queue.c, [46](#)
- queue\_init
  - cds\_queue.h, [21](#)
  - queue.c, [46](#)
- queue\_isempty
  - cds\_queue.h, [21](#)
  - queue.c, [46](#)
- queue\_length
  - cds\_queue.h, [21](#)
  - queue.c, [46](#)
- queue\_pop
  - cds\_queue.h, [22](#)
  - queue.c, [46](#)
- queue\_pushback
  - cds\_queue.h, [22](#)
  - queue.c, [47](#)
- sl\_list.c, [47](#)
  - sl\_list\_data, [48](#)
  - sl\_list\_delete\_at, [49](#)
  - sl\_list\_find, [49](#)
  - sl\_list\_find\_index, [49](#)
  - sl\_list\_find\_itr, [49](#)
  - sl\_list\_first, [49](#)
  - sl\_list\_free, [50](#)
  - sl\_list\_free\_node, [50](#)
  - sl\_list\_init, [50](#)
  - sl\_list\_insert\_after, [50](#)
  - sl\_list\_insert\_at, [50](#)
  - sl\_list\_isempty, [51](#)
  - sl\_list\_itr\_from\_index, [51](#)
  - sl\_list\_length, [51](#)
  - sl\_list\_new\_node, [51](#)
  - sl\_list\_next, [51](#)
  - sl\_list\_prepend, [51](#)
  - sl\_list\_remove\_at, [52](#)
- sl\_list.h, [52](#)
  - sl\_list\_find, [54](#)
  - sl\_list\_free\_node, [54](#)
  - sl\_list\_new\_node, [54](#)
  - sl\_list\_remove\_at, [54](#)
- sl\_list\_data
  - cds\_sl\_list.h, [24](#)

- sl\_list.c, 48
- sl\_list\_delete\_at
  - cds\_sl\_list.h, 24
  - sl\_list.c, 49
- sl\_list\_find
  - sl\_list.c, 49
  - sl\_list.h, 54
- sl\_list\_find\_index
  - cds\_sl\_list.h, 24
  - sl\_list.c, 49
- sl\_list\_find\_itr
  - cds\_sl\_list.h, 25
  - sl\_list.c, 49
- sl\_list\_first
  - cds\_sl\_list.h, 25
  - sl\_list.c, 49
- sl\_list\_free
  - cds\_sl\_list.h, 25
  - sl\_list.c, 50
- sl\_list\_free\_node
  - sl\_list.c, 50
  - sl\_list.h, 54
- sl\_list\_init
  - cds\_sl\_list.h, 25
  - sl\_list.c, 50
- sl\_list\_insert\_after
  - cds\_sl\_list.h, 25
  - sl\_list.c, 50
- sl\_list\_insert\_at
  - cds\_sl\_list.h, 26
  - sl\_list.c, 50
- sl\_list\_isempty
  - cds\_sl\_list.h, 26
  - sl\_list.c, 51
- sl\_list\_itr\_from\_index
  - cds\_sl\_list.h, 26
  - sl\_list.c, 51
- sl\_list\_length
  - cds\_sl\_list.h, 26
  - sl\_list.c, 51
- sl\_list\_new\_node
  - sl\_list.c, 51
  - sl\_list.h, 54
- sl\_list\_next
  - cds\_sl\_list.h, 26
  - sl\_list.c, 51
- sl\_list\_node\_t, 7
  - data, 7
  - next, 7
- sl\_list\_prepend
  - cds\_sl\_list.h, 27
  - sl\_list.c, 51
- sl\_list\_remove\_at
  - sl\_list.c, 52
  - sl\_list.h, 54
- sl\_list\_t, 7
  - cfunc, 8
  - front, 8
  - length, 8
- stack.c, 54
  - stack\_free, 55
  - stack\_init, 56
  - stack\_isempty, 56
  - stack\_length, 56
  - stack\_pop, 56
  - stack\_push, 56
- stack\_free
  - cds\_stack.h, 28
  - stack.c, 55
- stack\_init
  - cds\_stack.h, 29
  - stack.c, 56
- stack\_isempty
  - cds\_stack.h, 29
  - stack.c, 56
- stack\_length
  - cds\_stack.h, 29
  - stack.c, 56
- stack\_pop
  - cds\_stack.h, 29
  - stack.c, 56
- stack\_push
  - cds\_stack.h, 29
  - stack.c, 56