

CSC  
1.0.0

Generated by Doxygen 1.8.6

Wed Jan 2 2019 14:26:31



# 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	_node Struct Reference . . . . .	5
3.2	cbitset Struct Reference . . . . .	5
3.2.1	Field Documentation . . . . .	5
3.2.1.1	data . . . . .	5
3.2.1.2	nbits . . . . .	6
3.2.1.3	size . . . . .	6
3.3	cbst Struct Reference . . . . .	6
3.4	cvector Struct Reference . . . . .	6
3.4.1	Field Documentation . . . . .	7
3.4.1.1	capacity . . . . .	7
3.4.1.2	data . . . . .	7
3.4.1.3	size . . . . .	7
<b>4</b>	<b>File Documentation</b>	<b>9</b>
4.1	/home/tamer/csc/src/cbitset.c File Reference . . . . .	9
4.1.1	Detailed Description . . . . .	10
4.1.2	Function Documentation . . . . .	10
4.1.2.1	csc_cbitset_at . . . . .	10
4.1.2.2	csc_cbitset_clear . . . . .	11
4.1.2.3	csc_cbitset_clear_all . . . . .	11
4.1.2.4	csc_cbitset_create . . . . .	11
4.1.2.5	csc_cbitset_destroy . . . . .	12
4.1.2.6	csc_cbitset_flip . . . . .	12
4.1.2.7	csc_cbitset_set . . . . .	12
4.1.2.8	csc_cbitset_set_all . . . . .	12

4.1.2.9	csc_cbitset_size	13
4.2	/home/tamer/csc/src/cbitset.h File Reference	13
4.2.1	Detailed Description	14
4.2.2	Typedef Documentation	15
4.2.2.1	cbitset	15
4.2.3	Function Documentation	15
4.2.3.1	csc_cbitset_at	15
4.2.3.2	csc_cbitset_clear	16
4.2.3.3	csc_cbitset_clear_all	16
4.2.3.4	csc_cbitset_create	16
4.2.3.5	csc_cbitset_destroy	17
4.2.3.6	csc_cbitset_flip	17
4.2.3.7	csc_cbitset_set	17
4.2.3.8	csc_cbitset_set_all	17
4.2.3.9	csc_cbitset_size	18
4.3	/home/tamer/csc/src/cbst.c File Reference	18
4.3.1	Detailed Description	19
4.3.2	Function Documentation	19
4.3.2.1	csc_cbst_add	19
4.3.2.2	csc_cbst_create	20
4.3.2.3	csc_cbst_destroy	20
4.3.2.4	csc_cbst_empty	20
4.3.2.5	csc_cbst_find	21
4.3.2.6	csc_cbst_foreach	22
4.3.2.7	csc_cbst_rm	22
4.3.2.8	csc_cbst_size	22
4.4	/home/tamer/csc/src/cbst.h File Reference	23
4.4.1	Detailed Description	24
4.4.2	Typedef Documentation	25
4.4.2.1	cbst	25
4.4.3	Function Documentation	25
4.4.3.1	csc_cbst_add	25
4.4.3.2	csc_cbst_create	26
4.4.3.3	csc_cbst_destroy	26
4.4.3.4	csc_cbst_empty	26
4.4.3.5	csc_cbst_find	26
4.4.3.6	csc_cbst_foreach	27
4.4.3.7	csc_cbst_rm	27
4.4.3.8	csc_cbst_size	27
4.5	/home/tamer/csc/src/csc.c File Reference	28

4.5.1	Detailed Description	28
4.5.2	Macro Definition Documentation	29
4.5.2.1	CSC_DEFINE_BUILTIN_CMP	29
4.5.3	Function Documentation	29
4.5.3.1	csc_error_str	29
4.5.3.2	csc_swap	30
4.6	/home/tamer/csc/src/csc.h File Reference	31
4.6.1	Detailed Description	32
4.6.2	Macro Definition Documentation	32
4.6.2.1	CSC_DECLARE_BUILTIN_CMP	32
4.6.2.2	CSC_MAX_ERROR_MSG_LEN	33
4.6.2.3	CSC_UNUSED	33
4.6.3	Typedef Documentation	33
4.6.3.1	csc_compare	33
4.6.3.2	csc_foreach	33
4.6.3.3	CSCError	34
4.6.4	Enumeration Type Documentation	34
4.6.4.1	CSCError	34
4.6.5	Function Documentation	35
4.6.5.1	csc_error_str	35
4.6.5.2	csc_swap	35
4.7	/home/tamer/csc/src/cvector.c File Reference	35
4.7.1	Detailed Description	37
4.7.2	Function Documentation	37
4.7.2.1	csc_cvector_add	37
4.7.2.2	csc_cvector_at	37
4.7.2.3	csc_cvector_capacity	38
4.7.2.4	csc_cvector_create	38
4.7.2.5	csc_cvector_destroy	38
4.7.2.6	csc_cvector_empty	38
4.7.2.7	csc_cvector_find	39
4.7.2.8	csc_cvector_foreach	39
4.7.2.9	csc_cvector_reserve	39
4.7.2.10	csc_cvector_rm	40
4.7.2.11	csc_cvector_rm_at	40
4.7.2.12	csc_cvector_shrink_to_fit	40
4.7.2.13	csc_cvector_size	41
4.8	/home/tamer/csc/src/cvector.h File Reference	41
4.8.1	Detailed Description	43
4.8.2	Typedef Documentation	43

---

4.8.2.1	<a href="#">cvector</a>	44
4.8.2.2	<a href="#">cvector_foreach</a>	44
4.8.3	<a href="#">Function Documentation</a>	44
4.8.3.1	<a href="#">csc_cvector_add</a>	44
4.8.3.2	<a href="#">csc_cvector_at</a>	44
4.8.3.3	<a href="#">csc_cvector_capacity</a>	45
4.8.3.4	<a href="#">csc_cvector_create</a>	45
4.8.3.5	<a href="#">csc_cvector_destroy</a>	45
4.8.3.6	<a href="#">csc_cvector_empty</a>	45
4.8.3.7	<a href="#">csc_cvector_find</a>	46
4.8.3.8	<a href="#">csc_cvector_foreach</a>	46
4.8.3.9	<a href="#">csc_cvector_reserve</a>	46
4.8.3.10	<a href="#">csc_cvector_rm</a>	47
4.8.3.11	<a href="#">csc_cvector_rm_at</a>	47
4.8.3.12	<a href="#">csc_cvector_shrink_to_fit</a>	47
4.8.3.13	<a href="#">csc_cvector_size</a>	49
	<a href="#">Index</a>	50

# Chapter 1

## Data Structure Index

### 1.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">_node</a>	5
<a href="#">cbitset</a>	5
<a href="#">cbst</a>	6
<a href="#">cvector</a>	6





## Chapter 2

# File Index

### 2.1 File List

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

/home/tamer/csc/src/ <a href="#">cbitset.c</a>	
Implementation of the <a href="#">cbitset</a> data structure . . . . .	9
/home/tamer/csc/src/ <a href="#">cbitset.h</a>	
Defines the interface to the <a href="#">cbitset</a> data structure . . . . .	13
/home/tamer/csc/src/ <a href="#">cbst.c</a>	
This file defines the implements the <a href="#">cbst</a> data structure and interface functions . . . . .	18
/home/tamer/csc/src/ <a href="#">cbst.h</a>	
This file defines the interface to the <a href="#">cbst</a> data structure . . . . .	23
/home/tamer/csc/src/ <a href="#">csc.c</a>	
Implementation file for <a href="#">csc.h</a> . . . . .	28
/home/tamer/csc/src/ <a href="#">csc.h</a>	
Main include file for the csc library . . . . .	31
/home/tamer/csc/src/ <a href="#">cvector.c</a>	
Implementation of the cvector structure . . . . .	35
/home/tamer/csc/src/ <a href="#">cvector.h</a>	
Interface of the cvector structure . . . . .	41



## Chapter 3

# Data Structure Documentation

### 3.1 `_node` Struct Reference

Collaboration diagram for `_node`:



#### Data Fields

- `void * data`
- `struct _node * left`
- `struct _node * right`

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

- `/home/tamer/csc/src/cbst.c`

### 3.2 `cbitset` Struct Reference

#### Data Fields

- `bitset_type * data`
- `size_t nbits`
- `size_t size`

#### 3.2.1 Field Documentation

##### 3.2.1.1 `bitset_type* cbiset::data`

The internal data of the `bitset`.

### 3.2.1.2 `size_t cbiset::nbits`

The number of bits the bitset can hold.

### 3.2.1.3 `size_t cbiset::size`

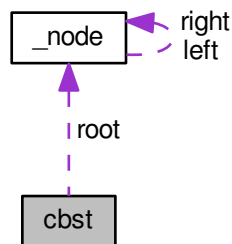
The number of elements stored in `cbiset::data`.

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

- `/home/tamer/csc/src/cbiset.c`

## 3.3 `cbst` Struct Reference

Collaboration diagram for `cbst`:



### Data Fields

- `_node * root`
- `size_t size`

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

- `/home/tamer/csc/src/cbst.c`

## 3.4 `cvector` Struct Reference

### Data Fields

- `void ** data`
- `size_t size`
- `size_t capacity`

### 3.4.1 Field Documentation

#### 3.4.1.1 `size_t cvector::capacity`

The number of elements the vector is capable of storing before needing to resize.

#### 3.4.1.2 `void** cvector::data`

The internal data store of the vector.

#### 3.4.1.3 `size_t cvector::size`

The number of elements currently in the vector.

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

- `/home/tamer/csc/src/cvector.c`



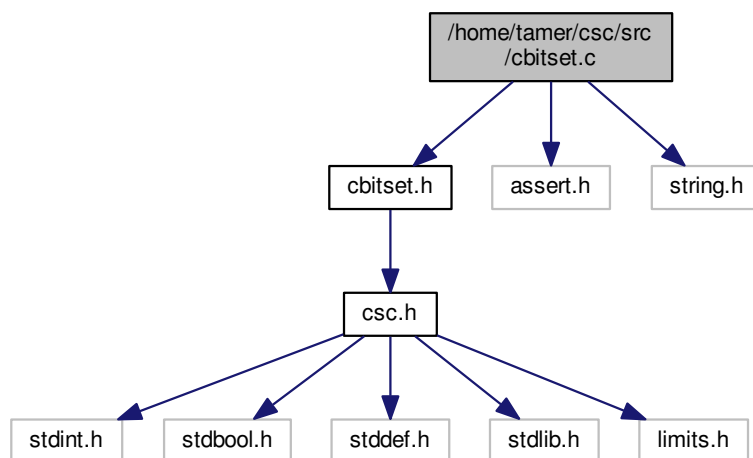
## Chapter 4

# File Documentation

### 4.1 /home/tamer/csc/src/cbitset.c File Reference

implementation of the [cbitset](#) data structure.

```
#include "cbitset.h"  
#include <assert.h>  
#include <string.h>  
Include dependency graph for cbitset.c:
```



### Data Structures

- struct [cbitset](#)

### Macros

- `#define CSC_BITSIZE ((sizeof(bitset_type)) * (8))`

## Typedefs

- typedef uint\_fast64\_t **bitset\_type**

## Functions

- **cbitsset \* csc\_cbitsset\_create** (size\_t nbits)  
*creates a [cbitsset](#).*
- void **csc\_cbitsset\_destroy** (cbitsset \*b)  
*destroys a [cbitsset](#).*
- size\_t **csc\_cbitsset\_size** (const cbitsset \*b)  
*return the number of bits the bitsset is capable of holding.*
- **CSCError csc\_cbitsset\_set** (cbitsset \*b, size\_t bit)  
*sets the 0-indexed bit supplied in the bitsset.*
- **CSCError csc\_cbitsset\_clear** (cbitsset \*b, size\_t bit)  
*clears the 0-indexed bit supplied in the bitsset.*
- **CSCError csc\_cbitsset\_flip** (cbitsset \*b, size\_t bit)  
*flips the 0-indexed bit supplied in the bitsset.*
- bool **csc\_cbitsset\_at** (const cbitsset \*b, size\_t bit, **CSCError \*e**)  
*retrieves the state of the bit at the specified 0-indexed position.*
- void **csc\_cbitsset\_set\_all** (cbitsset \*b)  
*sets all the bits in the bitsset.*
- void **csc\_cbitsset\_clear\_all** (cbitsset \*b)  
*clears all the bits in the bitsset.*

### 4.1.1 Detailed Description

implementation of the [cbitsset](#) data structure.

#### Author

Tamer Aly

#### Date

27 Dec 2018

#### See Also

[cbitsset.h](#)

### 4.1.2 Function Documentation

#### 4.1.2.1 bool csc\_cbitsset\_at ( const cbitsset \* b, size\_t bit, CSCError \* e )

retrieves the state of the bit at the specified 0-indexed position.

b is expected to be **non-null**.

**Time Complexity:**  $O(1)$



## Parameters

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to check.
<i>e</i>	<b>optional</b> parameter to retrieve any errors. Can be NULL.

## Returns

On success, *e* is `CSCError::E_NOERR`. If *bit* is out of range, *e* is `CSCError::E_OUTOFRANGE`. If the bit is set, `true` is returned and `false` otherwise.

4.1.2.2 `CSCError csc_cbitset_clear ( cbitset * b, size_t bit )`

clears the 0-indexed bit supplied in the bitset.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to clear.

## Returns

On success, `CSCError::E_NOERR`. If *bit* is out of range, `CSCError::E_OUTOFRANGE`.

4.1.2.3 `void csc_cbitset_clear_all ( cbitset * b )`

clears all the bits in the bitset.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(n)$

## Parameters

<i>b</i>	the bitset.
----------	-------------

4.1.2.4 `cbitset* csc_cbitset_create ( size_t nbits )`

creates a [cbitset](#).

This function creates a [cbitset](#) capable of holding *nbits* of data. Note that *nbits* must be greater than 0. The bitset is initialized with all of the bits cleared.

## Parameters

<i>nbits</i>	the number of bits the bitset should manage.
--------------	--

## Returns

a pointer to a [cbitset](#) if successful. On failure or if *nbits* is 0, NULL is returned.

## See Also

[csc\\_cbitset\\_destroy](#)

#### 4.1.2.5 void `csc_cbitset_destroy` ( `cbitset * b` )

destroys a `cbitset`.

This function destroys a `cbitset` by cleaning up any resources it holds. After a call to this function `b` should no longer be used.

`b` is expected to be **non-null**.

##### Parameters

<i>b</i>	the <code>bitset</code> .
----------	---------------------------

##### See Also

[csc\\_cbitset\\_create](#)

#### 4.1.2.6 `CSCError` `csc_cbitset_flip` ( `cbitset * b`, `size_t bit` )

flips the 0-indexed bit supplied in the `bitset`.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(1)$

##### Parameters

<i>b</i>	the <code>bitset</code> .
<i>bit</i>	the 0-indexed bit to flip.

##### Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

#### 4.1.2.7 `CSCError` `csc_cbitset_set` ( `cbitset * b`, `size_t bit` )

sets the 0-indexed bit supplied in the `bitset`.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(1)$

##### Parameters

<i>b</i>	the <code>bitset</code> .
<i>bit</i>	the 0-indexed bit to set.

##### Returns

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

#### 4.1.2.8 void `csc_cbitset_set_all` ( `cbitset * b` )

sets all the bits in the `bitset`.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(n)$

## Parameters

<i>b</i>	the bitset.
----------	-------------

4.1.2.9 `size_t csc_cbitset_size ( const cbitset * b )`

return the number of bits the bitset is capable of holding.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the bitset.
----------	-------------

## Returns

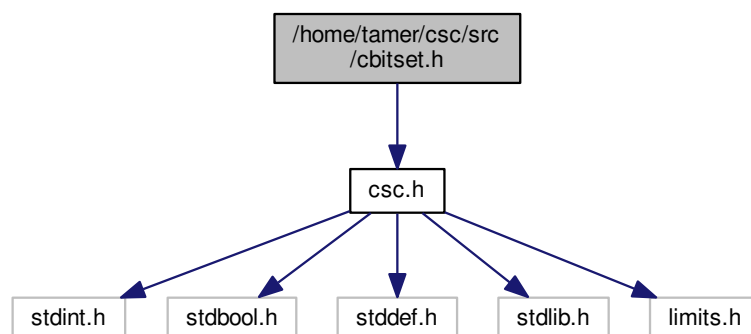
the number of bits the bitset is capable of holding.

## 4.2 /home/tamer/csc/src/cbitset.h File Reference

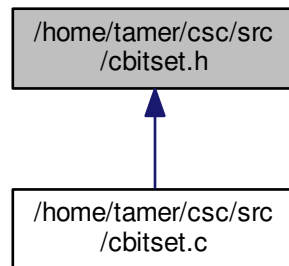
defines the interface to the [cbitset](#) data structure.

```
#include "csc.h"
```

Include dependency graph for cbitset.h:



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



## Typedefs

- typedef struct [cbitset](#) [cbitset](#)  
the *cbitset* data structure.

## Functions

- [cbitset \\*](#) [csc\\_cbitset\\_create](#) (size\_t nbits)  
creates a *cbitset*.
- void [csc\\_cbitset\\_destroy](#) (cbitset \*b)  
destroys a *cbitset*.
- size\_t [csc\\_cbitset\\_size](#) (const cbitset \*b)  
return the number of bits the bitset is capable of holding.
- [CSCError](#) [csc\\_cbitset\\_set](#) (cbitset \*b, size\_t bit)  
sets the 0-indexed bit supplied in the bitset.
- [CSCError](#) [csc\\_cbitset\\_clear](#) (cbitset \*b, size\_t bit)  
clears the 0-indexed bit supplied in the bitset.
- [CSCError](#) [csc\\_cbitset\\_flip](#) (cbitset \*b, size\_t bit)  
flips the 0-indexed bit supplied in the bitset.
- bool [csc\\_cbitset\\_at](#) (const cbitset \*b, size\_t bit, [CSCError](#) \*e)  
retrieves the state of the bit at the specified 0-indexed position.
- void [csc\\_cbitset\\_set\\_all](#) (cbitset \*b)  
sets all the bits in the bitset.
- void [csc\\_cbitset\\_clear\\_all](#) (cbitset \*b)  
clears all the bits in the bitset.

### 4.2.1 Detailed Description

defines the interface to the [cbitset](#) data structure.

#### Author

Tamer Aly

## Date

27 Dec 2018 Here is some code to get you started with basic usage of a [cbitset](#):

```
// create a bitset capable of holding 10 bits
cbitset* b = csc_cbitset_create(10);

// set bit 2nd bit
CSCError e = csc_cbitset_set(b, 1);
if (e != E_NOERR) {
    // handle the error
}

// check if the 3rd bit is set
if (csc_cbitset_at(b, 2, &e)) {
    // the bit is set
} else {
    // the bit isn't set
}

// flip the 4th bit
e = csc_cbitset_flip(b, 3);
if (e != E_NOERR) {
    // handle the error
}

// access the 11th bit (an error)
if (csc_cbitset_at(b, 10, &e)) {
    // can't happen
} else {
    if (e != E_NOERR) {
        // out of range
    } else {
        // can't happen
    }
}

// clean up
csc_cbitset_destroy(b);
```

## See Also

[cbitset.c](#)

## 4.2.2 Typedef Documentation

### 4.2.2.1 typedef struct cbitset cbitset

the cbitset data structure.

A bitset is a data structure that allows a user to manipulate state that can be represented in a single bit. Normally, this is used to store the state of several boolean conditions in a space efficient manner. You can think of a bitset as a space-optimized version of a vector of `bool` types.

## 4.2.3 Function Documentation

### 4.2.3.1 `bool csc_cbitset_at ( const cbitset * b, size_t bit, CSCError * e )`

retrieves the state of the bit at the specified 0-indexed position.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(1)$

#### Parameters

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to check.
<i>e</i>	<b>optional</b> parameter to retrieve any errors. Can be <code>NULL</code> .

**Returns**

On success, `e` is `CSCError::E_NOERR`. If `bit` is out of range, `e` is `CSCError::E_OUTOFRANGE`. If the bit is set, `true` is returned and `false` otherwise.

**4.2.3.2 `CSCError csc_cbitset_clear ( cbitset * b, size_t bit )`**

clears the 0-indexed bit supplied in the bitset.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(1)$

**Parameters**

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to clear.

**Returns**

On success, `CSCError::E_NOERR`. If `bit` is out of range, `CSCError::E_OUTOFRANGE`.

**4.2.3.3 `void csc_cbitset_clear_all ( cbitset * b )`**

clears all the bits in the bitset.

`b` is expected to be **non-null**.

**Time Complexity:**  $O(n)$

**Parameters**

<i>b</i>	the bitset.
----------	-------------

**4.2.3.4 `cbitset* csc_cbitset_create ( size_t nbits )`**

creates a [cbitset](#).

This function creates a [cbitset](#) capable of holding `nbits` of data. Note that `nbits` must be greater than 0. The bitset is initialized with all of the bits cleared.

**Parameters**

<i>nbits</i>	the number of bits the bitset should manage.
--------------	--

**Returns**

a pointer to a [cbitset](#) if successful. On failure or if `nbits` is 0, `NULL` is returned.

**See Also**

[csc\\_cbitset\\_destroy](#)

**4.2.3.5 void csc\_cbitset\_destroy ( cbitset \* *b* )**

destroys a [cbitset](#).

This function destroys a [cbitset](#) by cleaning up any resources it holds. After a call to this function *b* should no longer be used.

*b* is expected to be **non-null**.

**Parameters**

<i>b</i>	the bitset.
----------	-------------

**See Also**

[csc\\_cbitset\\_create](#)

**4.2.3.6 CSCErrors csc\_cbitset\_flip ( cbitset \* *b*, size\_t *bit* )**

flips the 0-indexed bit supplied in the bitset.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(1)$

**Parameters**

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to flip.

**Returns**

On success, `CSCErrors::E_NOERR`. If *bit* is out of range, `CSCErrors::E_OUTOFRANGE`.

**4.2.3.7 CSCErrors csc\_cbitset\_set ( cbitset \* *b*, size\_t *bit* )**

sets the 0-indexed bit supplied in the bitset.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(1)$

**Parameters**

<i>b</i>	the bitset.
<i>bit</i>	the 0-indexed bit to set.

**Returns**

On success, `CSCErrors::E_NOERR`. If *bit* is out of range, `CSCErrors::E_OUTOFRANGE`.

**4.2.3.8 void csc\_cbitset\_set\_all ( cbitset \* *b* )**

sets all the bits in the bitset.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(n)$

## Parameters

<i>b</i>	the bitset.
----------	-------------

4.2.3.9 `size_t csc_cbitset_size ( const cbitset * b )`

return the number of bits the bitset is capable of holding.

*b* is expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the bitset.
----------	-------------

## Returns

the number of bits the bitset is capable of holding.

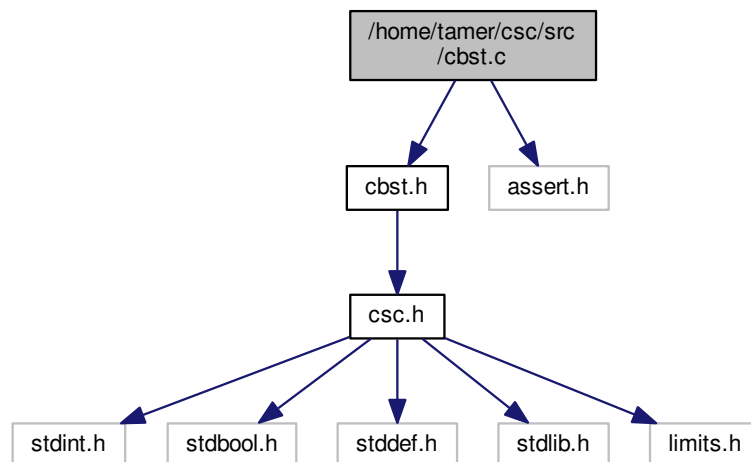
4.3 `/home/tamer/csc/src/cbst.c` File Reference

This file defines the implements the `cbst` data structure and interface functions.

```
#include "cbst.h"
```

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

Include dependency graph for `cbst.c`:



## Data Structures

- struct `_node`
- struct `cbst`



## Typedefs

- typedef struct `_node` `_node`

## Functions

- `cbst * csc_cbst_create ()`  
*cbst "constructor" function*
- `void csc_cbst_destroy (cbst *b)`  
*cbst "destructor" function*
- `CSError csc_cbst_add (cbst *b, void *elem, csc_compare cmp)`  
*adds an element into the BST.*
- `void * csc_cbst_rm (cbst *b, const void *elem, csc_compare cmp)`  
*removes an element from the BST.*
- `void * csc_cbst_find (const cbst *b, const void *elem, csc_compare cmp)`  
*finds the element in the specified BST.*
- `size_t csc_cbst_size (const cbst *b)`  
*returns the size of the BST.*
- `bool csc_cbst_empty (const cbst *b)`  
*checks if the BST is empty.*
- `void csc_cbst_foreach (cbst *b, csc_foreach fn, void *context)`  
*applies the callback function to each element of the BST in an **in-order** traversal. The user may pass in additional context using the `context` param or pass in `NULL` if not required.*

### 4.3.1 Detailed Description

This file defines the implements the `cbst` data structure and interface functions.

#### Author

Tamer Aly

#### Date

27 Dec 2018

### 4.3.2 Function Documentation

#### 4.3.2.1 `CSError csc_cbst_add ( cbst * b, void * elem, csc_compare cmp )`

adds an element into the BST.

This function adds `elem` into the supplied BST. Note that adding the element into the BST does **not** make the BST own the element. The user is still responsible for cleaning up that memory. Moreover, duplicate elements are not allowed.

Both `elem` and `b` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

## Parameters

<i>b</i>	the BST.
<i>elem</i>	the element to add.

## Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`. If a duplicate element is attempted to be added, `CSCError::E_INVALIDOPERATION`.

4.3.2.2 `cbst* csc_cbst_create ( )`

cbst "constructor" function

This function is used to create a `cbst`. If the function is successful, the function returns a pointer to a `cbst` created on the heap. If unsuccessful, `NULL` is returned.

## Returns

a pointer to a constructed `cbst`.

## See Also

[csc\\_cbst\\_destroy](#)

4.3.2.3 `void csc_cbst_destroy ( cbst * b )`

cbst "destructor" function

This function is used to clean up resources used by a `cbst` created via the [csc\\_cbst\\_create](#) function. This function must be called whenever a `cbst` is no longer used.

## See Also

[csc\\_cbst\\_create](#)

4.3.2.4 `bool csc_cbst_empty ( const cbst * b )`

checks if the BST is empty.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the BST.
----------	----------

## Returns

`true` if the BST is empty. In other words, `true` if `csc_cbst_size(v) == 0`. Otherwise, `false`.

4.3.2.5 void\* `csc_cbst_find` ( const `cbst` \* *b*, const void \* *elem*, `csc_compare` *cmp* )

finds the element in the specified BST.

This function attempts to find `elem` using comparator `cmp`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

## Parameters

<i>b</i>	the BST.
<i>elem</i>	the element to find.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

## Returns

the element or `NULL` if the element couldn't be found.

#### 4.3.2.6 void csc\_cbst\_foreach ( cbst \* b, csc\_foreach fn, void \* context )

applies the callback function to each element of the BST in an **in-order** traversal. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

**Time Complexity:**  $O(n)$

## Parameters

<i>b</i>	the BST.
<i>fn</i>	the callback function to apply to each element.
<i>context</i>	user-defined data that will be applied to the callback. Can be <code>NULL</code> if unused.

#### 4.3.2.7 void\* csc\_cbst\_rm ( cbst \* b, const void \* elem, csc\_compare cmp )

removes an element from the BST.

This function removes `elem` from the supplied BST if it exists. In order to remove the element, the function must search for the element in the BST using the supplied `cmp` function. The removed element is returned.

All three parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

## Parameters

<i>b</i>	the BST.
<i>elem</i>	the element to remove.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

## Returns

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

## See Also

[csc\\_compare](#)  
[csc\\_cbst\\_find](#)

#### 4.3.2.8 size\_t csc\_cbst\_size ( const cbst \* b )

returns the size of the BST.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the BST.
----------	----------

## Returns

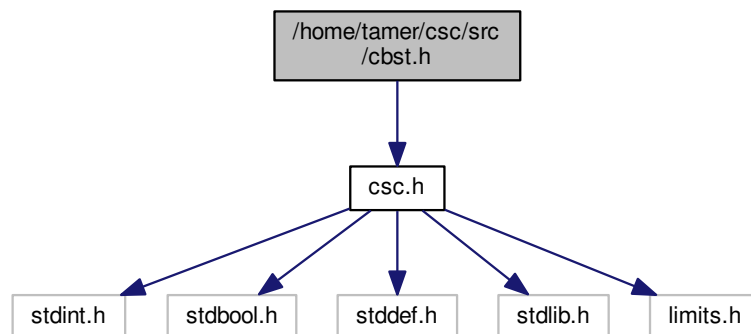
the size of the BST.

## 4.4 /home/tamer/csc/src/cbst.h File Reference

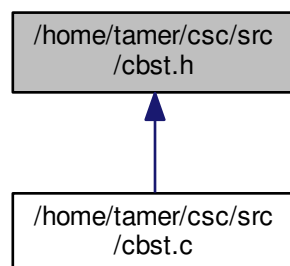
This file defines the interface to the [cbst](#) data structure.

```
#include "csc.h"
```

Include dependency graph for cbst.h:



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



## Typedefs

- typedef struct [cbst](#) [cbst](#)  
*implementation of a binary search tree.*

## Functions

- `cbst * csc_cbst_create ()`  
*cbst "constructor" function*
- `void csc_cbst_destroy (cbst *b)`  
*cbst "destructor" function*
- `CSCError csc_cbst_add (cbst *b, void *elem, csc_compare cmp)`  
*adds an element into the BST.*
- `void * csc_cbst_rm (cbst *b, const void *elem, csc_compare cmp)`  
*removes an element from the BST.*
- `void * csc_cbst_find (const cbst *b, const void *elem, csc_compare cmp)`  
*finds the element in the specified BST.*
- `size_t csc_cbst_size (const cbst *b)`  
*returns the size of the BST.*
- `bool csc_cbst_empty (const cbst *b)`  
*checks if the BST is empty.*
- `void csc_cbst_foreach (cbst *b, csc_foreach fn, void *context)`  
*applies the callback function to each element of the BST in an **in-order** traversal. The user may pass in additional context using the *context* param or pass in *NULL* if not required.*

### 4.4.1 Detailed Description

This file defines the interface to the `cbst` data structure.

#### Author

Tamer Aly

#### Date

27 Dec 2018 `cbst` implements a standard binary search tree (BST). In this implementation, attempting to add duplicate or `NULL` keys is not allowed. See `#csc_bst_add` for more details.

Here is a brief code sample to get you started with using `cbst`:

```
// a callback function to call on each element of the tree
void print_and_free_elem(void* elem, void* context);

//
// somewhere in main
//

// create a tree
cbst* tree = csc_cbst_create();
if (tree == NULL) {
    // couldn't create the tree
}

// add some elements
int elems[] = {5, 3, 7, 2, 4, 6, 8};
for (int i = 0; i < sizeof(elems) / sizeof(int); ++i) {
    int* x = malloc(sizeof(*x));
    if (x == NULL) {
        // couldn't allocate memory
    }
    *x = elems[i];
    CSCError e = csc_cbst_add(tree, x, csc_cmp_int);
    if (e != E_NOERR) {
        // handle the error
    }
}

// remove an element
int* e = (int*) csc_cbst_rm(b, &elems[0], csc_cmp_int);
if (e == NULL) {
```

```

    // the element didn't exist
} else {
    // e is the element which was removed.
    // here is an opportunity to do any resource cleanup or operations on e.
    free(e);
}

// find an element
int* elem = (int*) csc_cbst_find(b, &elems[2], csc_cmp_int);
if (elem == NULL) {
    // element wasn't found.
}

// get the number of elements in the tree
size_t n_elems = csc_cbst_size(b);

// iterate over all of the elements in the tree in an in-order traversal
csc_cbst_foreach(b, print_elem, NULL);

// clean up
csc_cbst_destroy(b);

// implement the callback
void print_and_free_elem(void* elem, void* context)
{
    CSC_UNUSED(context);
    int* e = (int*) elem;
    printf("%d\n", *e);
    free(e);
}

```

## 4.4.2 Typedef Documentation

### 4.4.2.1 typedef struct cbst cbst

implementation of a binary search tree.

See Also

[csc\\_cbst\\_create](#)

## 4.4.3 Function Documentation

### 4.4.3.1 CSCErrror csc\_cbst\_add ( cbst \* b, void \* elem, csc\_compare cmp )

adds an element into the BST.

This function adds `elem` into the supplied BST. Note that adding the element into the BST does **not** make the BST own the element. The user is still responsible for cleaning up that memory. Moreover, duplicate elements are not allowed.

Both `elem` and `b` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

Parameters

<i>b</i>	the BST.
<i>elem</i>	the element to add.

Returns

On success, `CSCErrror::E_NOERR`. On memory allocation failure `CSCErrror::E_OUTOFMEM`. If a duplicate element is attempted to be added, `CSCErrror::E_INVALIDOPERATION`.

#### 4.4.3.2 `cbst* csc_cbst_create ( )`

cbst "constructor" function

This function is used to create a `cbst`. If the function is successful, the function returns a pointer to a `cbst` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed `cbst`.

See Also

[csc\\_cbst\\_destroy](#)

#### 4.4.3.3 `void csc_cbst_destroy ( cbst * b )`

cbst "destructor" function

This function is used to clean up resources used by a `cbst` created via the [csc\\_cbst\\_create](#) function. This function must be called whenever a `cbst` is no longer used.

See Also

[csc\\_cbst\\_create](#)

#### 4.4.3.4 `bool csc_cbst_empty ( const cbst * b )`

checks if the BST is empty.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

Parameters

<i>b</i>	the BST.
----------	----------

Returns

`true` if the BST is empty. In other words, true if `csc_cbst_size(v) == 0`. Otherwise, `false`.

#### 4.4.3.5 `void* csc_cbst_find ( const cbst * b, const void * elem, csc_compare cmp )`

finds the element in the specified BST.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

Parameters



<i>b</i>	the BST.
<i>elem</i>	the element to find.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

**Returns**

the element or `NULL` if the element couldn't be found.

**4.4.3.6 void csc\_cbst\_foreach ( cbst \* b, csc\_foreach fn, void \* context )**

applies the callback function to each element of the BST in an **in-order** traversal. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

**Time Complexity:**  $O(n)$

**Parameters**

<i>b</i>	the BST.
<i>fn</i>	the callback function to apply to each element.
<i>context</i>	user-defined data that will be applied to the callback. Can be <code>NULL</code> if unused.

**4.4.3.7 void\* csc\_cbst\_rm ( cbst \* b, const void \* elem, csc\_compare cmp )**

removes an element from the BST.

This function removes `elem` from the supplied BST if it exists. In order to remove the element, the function must search for the element in the BST using the supplied `cmp` function. The removed element is returned.

All three parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(h)$  worst case,  $O(\log(n))$  average case where  $h$  is the height of the tree and  $n$  is the number of elements the tree holds.

**Parameters**

<i>b</i>	the BST.
<i>elem</i>	the element to remove.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

**Returns**

If the element is successfully removed, the element is returned. Otherwise, `NULL`.

**See Also**

[csc\\_compare](#)  
[csc\\_cbst\\_find](#)

**4.4.3.8 size\_t csc\_cbst\_size ( const cbst \* b )**

returns the size of the BST.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>b</i>	the BST.
----------	----------

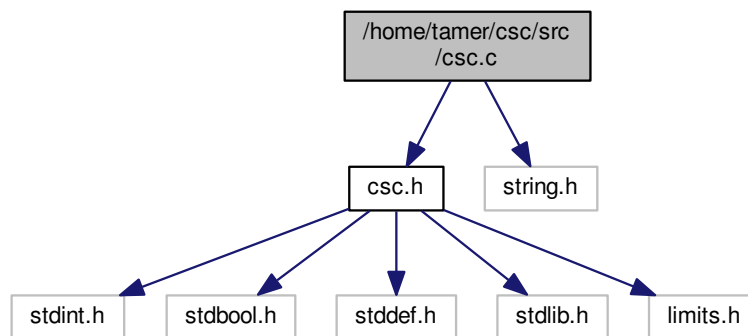
## Returns

the size of the BST.

## 4.5 /home/tamer/csc/src/csc.c File Reference

the implementation file for [csc.h](#).

```
#include "csc.h"
#include <string.h>
Include dependency graph for csc.c:
```



### Macros

- `#define CSC_DEFINE_BUILTIN_CMP(type)`  
*implements a builtin type comparison function*

### Functions

- `void csc_swap (void **a, void **b)`  
*generic swap function to swap two void\**
- `void csc_error_str (CSCError e, char *buf, size_t len)`  
*returns a library-defined error string depending on the error.*

#### 4.5.1 Detailed Description

the implementation file for [csc.h](#).

#### Author

Tamer Aly

## Date

27 Dec 2018

## See Also

[csc.h](#)

## 4.5.2 Macro Definition Documentation

## 4.5.2.1 #define CSC\_DEFINE\_BUILTIN\_CMP( type )

## Value:

```
int csc_cmp_##type(const void* a, const void* b) \
{ \
    type vA = *(type*)a; \
    type vB = *(type*)b; \
    if (vA == vB) { return 0; } \
    else if (vA < vB) { return -1; } \
    else { return 1; } \
}
```

implements a builtin type comparison function

When defined with a type, this macro will implement the function signature that the `CSC_DECLARE_BUILTIN_CMP` defines. Note that the function signature **must** be declared first using `CSC_DECLARE_BUILTIN_CMP` in the header file.

## See Also

[csc.h](#)

## 4.5.3 Function Documentation

## 4.5.3.1 void csc\_error\_str ( CSCError e, char \* buf, size\_t len )

returns a library-defined error string depending on the error.

This is a convenience function that populates `buf` of length `len` with a library-defined error message that depends on the value of `e`. It is recommended that `len` is *at least* [CSC\\_MAX\\_ERROR\\_MSG\\_LEN](#).

This function should ideally be used after a library call returning a [CSCError](#) for a simple diagnostic error handling mechanism. For example:

```
CSCError e = csc_some_func(args...);
if (e != E_NOERR) { // uh oh. An error.
    char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
    csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
    puts(buf);
}
```

## Parameters

<i>e</i>	the error.
<i>buf</i>	the buffer to fill. Must be <b>non-null</b> .
<i>len</i>	the length of the buffer. Recommended to be $\geq$ <a href="#">CSC_MAX_ERROR_MSG_LEN</a> .

## See Also

[CSC\\_MAX\\_ERROR\\_MSG\\_LEN](#)

#### 4.5.3.2 void csc\_swap ( void \*\* *a*, void \*\* *b* )

generic swap function to swap two void\*

This is a generic swap function to swap the provided elements.

## Parameters

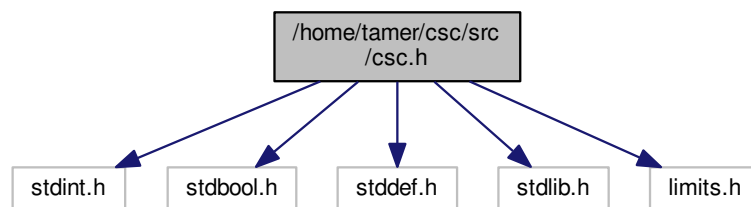
<i>a</i>	A is the first elem.
<i>b</i>	B is the second elem.

## 4.6 /home/tamer/csc/src/csc.h File Reference

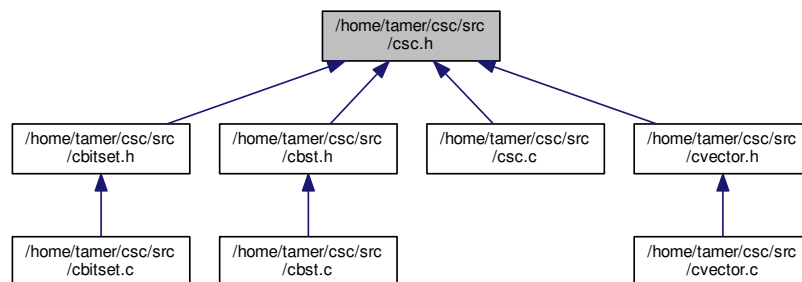
the main include file for the csc library.

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <limits.h>
```

Include dependency graph for csc.h:



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



## Macros

- `#define CSC_UNUSED(x) (void)x`  
macro that silences compiler warnings about unused function parameters.
- `#define CSC_64`  
This macro is only defined if compiling on a 64 bit architecture.
- `#define CSC_32`  
This macro is only defined if compiling on a 32 bit architecture.

- `#define CSC_DECLARE_BUILTIN_CMP(type) int csc_cmp_##type(const void* a, const void* b)`  
*convenience macro defining comparison functions for built in types.*
- `#define CSC_MAX_ERROR_MSG_LEN 128`  
*the maximum message length a `CSCError` is guaranteed to generate.*

## Typedefs

- `typedef enum CSCError CSCError`  
*the list of errors that can be returned by the library.*
- `typedef int(* csc_compare)(const void *a, const void *b)`  
*comparison function callback for comparing two elements*
- `typedef void(* csc_foreach)(void *elem, void *context)`  
*callback function for iterating the elements of a container.*

## Enumerations

- `enum CSCError {`  
`E_NOERR = 0, E_OUTOFMEM, E_OUTOFRANGE, E_INVALIDOPERATION,`  
`E_ERR_N }`  
*the list of errors that can be returned by the library.*

## Functions

- `void csc_swap (void **a, void **b)`  
*generic swap function to swap two `void*`*
- `void csc_error_str (CSCError e, char *buf, size_t len)`  
*returns a library-defined error string depending on the error.*
- `CSC_DECLARE_BUILTIN_CMP (int)`

### 4.6.1 Detailed Description

the main include file for the csc library.

#### Author

Tamer Aly

#### Date

27 Dec 2018 This is the main include file that must be included alongside any other source and header combination for a particular data structure in the library. This file defines several helper functions that are used throughout the library.

### 4.6.2 Macro Definition Documentation

#### 4.6.2.1 `#define CSC_DECLARE_BUILTIN_CMP( type ) int csc_cmp_##type(const void* a, const void* b)`

convenience macro defining comparison functions for built in types.

This macro defines a comparison function for built-in C types.

For example, defining `CSC_DECLARE_BUILTIN_CMP(int)` would create the signature:

```
int csc_cmp_int(const void* a, const void* b);
```

Note that this macro only creates the signature of the function. See [csc.c](#) for how to implement the signature.

See Also

[CSC.C](#)

#### 4.6.2.2 #define CSC\_MAX\_ERROR\_MSG\_LEN 128

the maximum message length a [CSCError](#) is guaranteed to generate.

See Also

[csc\\_error\\_str](#)

#### 4.6.2.3 #define CSC\_UNUSED( x )(void)x

macro that silences compiler warnings about unused function parameters.

This macro is used to silence compiler warnings about unused function parameters. Mostly, this is for unused context parameters in generic callback functions used internally by the library.

### 4.6.3 Typedef Documentation

#### 4.6.3.1 typedef int(\* csc\_compare)(const void \*a, const void \*b)

comparison function callback for comparing two elements

This is a comparison function callback for comparing two elements that is used for routine functions like sorting or searching a generic container. When creating a custom comparison function for your type, the following protocol **must** be adhered to: a return value  $< 0$  means  $a$  is less than  $b$ . a return value of  $> 0$  means  $a$  is greater than  $b$ . a return value of  $0$  means  $a$  is equal to  $b$ .

As a convenience, the library provides comparison functions for all the C built in types.

Parameters

$a$	the first element
$b$	the second element

See Also

[CSC\\_DECLARE\\_BUILTIN\\_CMP](#)

#### 4.6.3.2 typedef void(\* csc\_foreach)(void \*elem, void \*context)

callback function for iterating the elements of a container.

This callback function defines an operation that will be applied to each element of a container.

Parameters

<i>elem</i>	the element to process
<i>context</i>	user-defined data that can be passed into the function. Can be <code>NULL</code> if unused.

#### 4.6.3.3 typedef enum `CSCError` `CSCError`

the list of errors that can be returned by the library.

This enumeration defines all of the errors that can be returned by certain library calls. These errors can provide more diagnostic information than a simple true/false return. Whenever this error type is returned, a type of `CSCError` : : `E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCError` : : `E_ERR_N`, indicates an error condition.

It is recommended that you check for this error code whenever possible:

```
CSCError e = csc_function(args...);
if (e != E_NOERR) {
    // handle the error by printing a simple diagnostic message.
    char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
    csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
    puts(buf);
}
```

See Also

[csc\\_error\\_str](#)

### 4.6.4 Enumeration Type Documentation

#### 4.6.4.1 enum `CSCError`

the list of errors that can be returned by the library.

This enumeration defines all of the errors that can be returned by certain library calls. These errors can provide more diagnostic information than a simple true/false return. Whenever this error type is returned, a type of `CSCError` : : `E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCError` : : `E_ERR_N`, indicates an error condition.

It is recommended that you check for this error code whenever possible:

```
CSCError e = csc_function(args...);
if (e != E_NOERR) {
    // handle the error by printing a simple diagnostic message.
    char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
    csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
    puts(buf);
}
```

See Also

[csc\\_error\\_str](#)

Enumerator

**`E_NOERR`** This indicates no errors occurred in the operation.

**`E_OUTOFMEM`** This error indicates the operation failed since memory could not be allocated.

**`E_OUTOFRANGE`** This error indicates that the operation failed due to accessing an out of range element. i.e. array index of -1.

**`E_INVALIDOPERATION`** This error indicates that the operation failed because an invalid operation was attempted.

**`E_ERR_N`** This is never returned by any function calls and can be ignored.



### 4.6.5 Function Documentation

#### 4.6.5.1 void csc\_error\_str ( CSCError e, char \* buf, size\_t len )

returns a library-defined error string depending on the error.

This is a convenience function that populates `buf` of length `len` with a library-defined error message that depends on the value of `e`. It is recommended that `len` is *at least* [CSC\\_MAX\\_ERROR\\_MSG\\_LEN](#).

This function should ideally be used after a library call returning a [CSCError](#) for a simple diagnostic error handling mechanism. For example:

```
CSCError e = csc_some_func(args...);
if (e != E_NOERR) { // uh oh. An error.
    char buf[CSC_MAX_ERROR_MSG_LEN] = {0};
    csc_error_str(e, buf, CSC_MAX_ERROR_MSG_LEN);
    puts(buf);
}
```

#### Parameters

<code>e</code>	the error.
<code>buf</code>	the buffer to fill. Must be <b>non-null</b> .
<code>len</code>	the length of the buffer. Recommended to be $\geq$ <a href="#">CSC_MAX_ERROR_MSG_LEN</a> .

#### See Also

[CSC\\_MAX\\_ERROR\\_MSG\\_LEN](#)

#### 4.6.5.2 void csc\_swap ( void \*\* a, void \*\* b )

generic swap function to swap two `void*`

This is a generic swap function to swap the provided elements.

#### Parameters

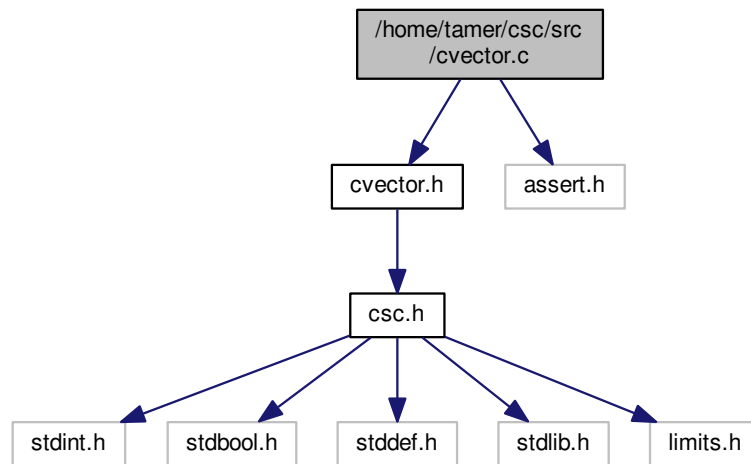
<code>a</code>	A is the first elem.
<code>b</code>	B is the second elem.

## 4.7 /home/tamer/csc/src/cvector.c File Reference

contains the implementation of the `cvector` structure.

```
#include "cvector.h"
#include <assert.h>
```

Include dependency graph for `cvector.c`:



## Data Structures

- struct `cvector`

## Functions

- `cvector * csc_cvector_create ()`  
*cvector "constructor" function*
- `size_t csc_cvector_size (const cvector *v)`  
*returns the size of the vector.*
- `size_t csc_cvector_capacity (const cvector *v)`  
*returns the capacity of the vector.*
- `void csc_cvector_destroy (cvector *v)`  
*cvector "destructor" function*
- `void csc_cvector_foreach (cvector *v, cvector_foreach fn, void *context)`  
*applies the callback function to each element of the vector.*
- `CSCError csc_cvector_add (cvector *v, void *elem)`  
*adds an element into the vector.*
- `void * csc_cvector_at (const cvector *v, size_t idx)`  
*returns the element at the specified index.*
- `void csc_cvector_rm (cvector *v, const void *elem, csc_compare cmp)`  
*removes an element from the vector.*
- `CSCError csc_cvector_rm_at (cvector *v, size_t idx)`  
*removes the element at the specified 0-indexed index from the vector.*
- `void * csc_cvector_find (const cvector *v, const void *elem, csc_compare cmp)`  
*finds the element in the specified vector.*
- `bool csc_cvector_empty (const cvector *v)`  
*checks if the vector is empty.*
- `CSCError csc_cvector_reserve (cvector *v, size_t num_elems)`

*reserves memory for the specified number of elements in the vector.*

- `CSCError csc_cvector_shrink_to_fit (cvector *v)`

*shrinks the capacity to match the size of the vector.*

### 4.7.1 Detailed Description

contains the implementation of the cvector structure.

#### Author

Tamer Aly

#### Date

27 Dec 2018

### 4.7.2 Function Documentation

#### 4.7.2.1 `CSCError csc_cvector_add ( cvector * v, void * elem )`

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that adding the element into the vector does **not** make the vector own the element. The user is still responsible for cleaning up that memory.

Both `elem` and `v` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  worst case,  $O(1)$  amortized.

#### Parameters

<code>v</code>	the vector.
<code>elem</code>	the element to add.

#### Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`.

#### 4.7.2.2 `void* csc_cvector_at ( const cvector * v, size_t idx )`

returns the element at the specified index.

This function performs a range check to ensure that `idx` is less than the size of the vector.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

#### Parameters

<code>v</code>	the vector.
<code>idx</code>	the index.

#### Returns

the element at that index in the vector or `NULL` if the index is out of range.

#### 4.7.2.3 `size_t csc_cvector_capacity ( const cvector * v )`

returns the capacity of the vector.

This function returns the number of elements the vector can hold before it needs to be resized.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

Parameters

<code>v</code>	the vector.
----------------	-------------

Returns

the capacity of the vector.

#### 4.7.2.4 `cvector* csc_cvector_create ( )`

cvector "constructor" function

This function is used to create a `cvector`. If the function is successful, the function returns a pointer to a `cvector` created on the heap. If unsuccessful, `NULL` is returned.

Returns

a pointer to a constructed `cvector`.

See Also

[csc\\_cvector\\_destroy](#)

#### 4.7.2.5 `void csc_cvector_destroy ( cvector * v )`

cvector "destructor" function

This function is used to clean up resources used by a `cvector` created via the [csc\\_cvector\\_create](#) function. This function must be called whenever a `cvector` is no longer used.

See Also

[csc\\_cvector\\_create](#)

#### 4.7.2.6 `bool csc_cvector_empty ( const cvector * v )`

checks if the vector is empty.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

Parameters

<code>v</code>	the vector.
----------------	-------------

Returns

`true` if the vector is empty. In other words, `true` if `csc_cvector_size(v) == 0`. Otherwise, `false`.

**4.7.2.7** void\* `csc_cvector_find` ( const `cvector` \* *v*, const void \* *elem*, `csc_compare` *cmp* )

finds the element in the specified vector.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  average and worst case.

**Parameters**

<i>v</i>	the vector.
<i>elem</i>	the element to find.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

**Returns**

the element or `NULL` if the element couldn't be found.

**4.7.2.8** void `csc_cvector_foreach` ( `cvector` \* *v*, `cvector_foreach` *fn*, void \* *context* )

applies the callback function to each element of the vector.

This callback function defines an operation that will be applied to each element of the `cvector`. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

**Time Complexity:**  $O(n)$

**Parameters**

<i>v</i>	the vector.
<i>fn</i>	the callback function to apply to each element. See <a href="#">cvector_foreach</a> .
<i>context</i>	user-defined data that will be applied to the callback. Can be <code>NULL</code> if unused.

**See Also**

[cvector\\_foreach](#)

**4.7.2.9** `CSCError` `csc_cvector_reserve` ( `cvector` \* *v*, `size_t` *num\_elems* )

reserves memory for the specified number of elements in the vector.

This functions reserves enough memory in the vector such that it is able to hold *at least* `num_elems` without needing to expand. If the number of elements that will be contained in the vector is known or can be estimated, you may be able to improve the performance of your application by allocating the memory for the elements up front using this function. As always with performance, your mileage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is  $< \text{csc\_cvector\_size}(v)$ , that is an error.

All parameters are expected to be **non-null**.

**Time Complexity:** OS-specific.

**Parameters**

<i>v</i>	the vector.
----------	-------------

<i>num_elems</i>	the number of elements to allocate memory for.
------------------	--

#### Returns

On success `CSCError::E_NOERR`. If the requested size is less than the current size, `CSCError::E_INVALIDOPERATION`. If there is a memory error, `CSCError::E_OUTOFMEM`.

#### 4.7.2.10 `void csc_cvector_rm ( cvector * v, const void * elem, csc_compare cmp )`

removes an element from the vector.

This function removes `elem` from the supplied vector if it exists. In order to remove the element, the function must search for the element in the vector using the supplied `cmp` function.

All three parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  average and worst case.

#### Parameters

<i>v</i>	the vector.
<i>elem</i>	the element to remove.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

#### See Also

[csc\\_compare](#)  
[csc\\_cvector\\_find](#)

#### 4.7.2.11 `CSCError csc_cvector_rm_at ( cvector * v, size_t idx )`

removes the element at the specified 0-indexed index from the vector.

This function the element at index `idx` from `v`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

#### Parameters

<i>v</i>	the vector.
<i>idx</i>	the index.

#### Returns

`CSCError::E_NOERR` or `CSCError::E_OUTOFRANGE` if the supplied index is out of range.

#### 4.7.2.12 `CSCError csc_cvector_shrink_to_fit ( cvector * v )`

shrinks the capacity to match the size of the vector.

After a call to this function, the following will be true:

```
csc_cvector_size(v) == csc_cvector_capacity(v);
```

This function may be useful in low-memory settings where the vector's capacity greatly exceeds the size and the extra memory won't be required.

All parameters are expected to be **non-null**.

**Time Complexity:** OS-specific.

## Parameters

<i>v</i>	the vector.
----------	-------------

## Returns

On success `CSCError::E_NOERR`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.7.2.13 `size_t csc_cvector_size ( const cvector * v )`

returns the size of the vector.

This function returns the number of elements currently in the vector.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>v</i>	the vector.
----------	-------------

## Returns

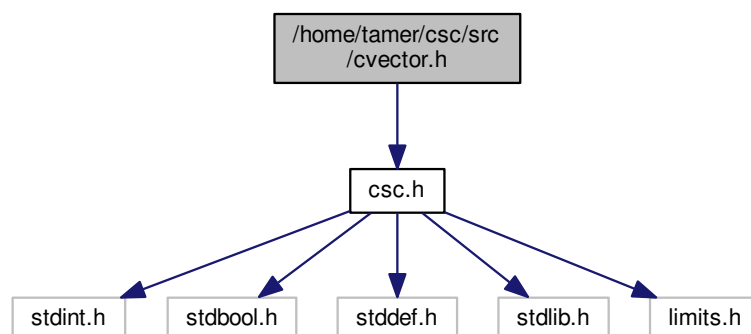
the size of the vector.

## 4.8 /home/tamer/csc/src/cvector.h File Reference

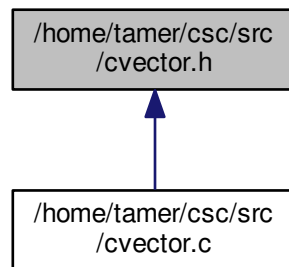
contains interface of the cvector structure.

```
#include "csc.h"
```

Include dependency graph for cvector.h:



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



## Typedefs

- typedef struct `cvector` `cvector`  
*implementation of a generic dynamic array.*
- typedef void(\* `cvector_foreach`)(void \*elem, void \*context)  
*callback function for iterating the elements of a `cvector`.*

## Functions

- `cvector * csc_cvector_create ()`  
*cvector "constructor" function*
- void `csc_cvector_destroy (cvector *v)`  
*cvector "destructor" function*
- `CSCError csc_cvector_add (cvector *v, void *elem)`  
*adds an element into the vector.*
- void `csc_cvector_rm (cvector *v, const void *elem, csc_compare cmp)`  
*removes an element from the vector.*
- `CSCError csc_cvector_rm_at (cvector *v, size_t idx)`  
*removes the element at the specified 0-indexed index from the vector.*
- void \* `csc_cvector_find (const cvector *v, const void *elem, csc_compare cmp)`  
*finds the element in the specified vector.*
- size\_t `csc_cvector_size (const cvector *v)`  
*returns the size of the vector.*
- size\_t `csc_cvector_capacity (const cvector *v)`  
*returns the capacity of the vector.*
- void `csc_cvector_foreach (cvector *v, cvector_foreach fn, void *context)`  
*applies the callback function to each element of the vector.*
- void \* `csc_cvector_at (const cvector *v, size_t idx)`  
*returns the element at the specified index.*
- bool `csc_cvector_empty (const cvector *v)`  
*checks if the vector is empty.*
- `CSCError csc_cvector_reserve (cvector *v, size_t num_elems)`  
*reserves memory for the specified number of elements in the vector.*
- `CSCError csc_cvector_shrink_to_fit (cvector *v)`  
*shrinks the capacity to match the size of the vector.*



### 4.8.1 Detailed Description

contains interface of the cvector structure.

#### Author

Tamer Aly

#### Date

27 Dec 2018 Here is example code to get you started on using the `cvector`:

```
// for-each callback function signature
void print_elem(void* elem, void* context);

//
// somewhere in main....
//

// create a vector
cvector* v = csc_cvector_create();
if (v == NULL) {
    // couldn't create the vector.
}

// add some elements into the vector.
for (int i = 0; i < 10; i++) {
    int* x = malloc(sizeof(*x));
    if (x == NULL) {
        // couldn't allocate memory.
    }
    *x = i;

    CSCErr e = csc_cvector_add(v, x);
    if (e != E_NOERR) {
        // couldn't add the element.
    }
}

// get the size
size_t size = csc_cvector_size(v);

// print the elements "manually"
for (size_t i = 0; i < size; i++) {
    int* x = (int*) csc_cvector_at(v, i);
    printf("%d\n", *x);
}

// remove the 2nd element
CSCErr e = csc_cvector_rm_at(v, 1);
if (e != E_NOERR) {
    // couldn't remove the element
}

// print the elements "functionally"
csc_cvector_foreach(v, print_elem, NULL);

// clean up resources
for (size_t i = 0; i < csc_cvector_size(v); ++i) {
    void* x = csc_cvector_at(v, i);
    free(x);
}
csc_cvector_destroy(v);

//
// somewhere outside of main...
//

// implement the callback
void print_elem(void* elem, void* context)
{
    CSC_UNUSED(context); // no need for context
    printf("%d\n", *(int*)elem);
}
```

### 4.8.2 Typedef Documentation

#### 4.8.2.1 typedef struct `cvector` `cvector`

implementation of a generic dynamic array.

`cvector` implements a dynamic array that mimics `std::vector` from C++.

See Also

[csc\\_cvector\\_create](#)

#### 4.8.2.2 typedef void(\* `cvector_foreach`)(void \*`elem`, void \*`context`)

callback function for iterating the elements of a `cvector`.

This callback function defines an operation that will be applied to each element of the `cvector` by the `csc_cvector_foreach` function.

Parameters

<i>elem</i>	the element to process
<i>context</i>	user-defined data that can be passed into the function. Can be <code>NULL</code> if unused.

See Also

[csc\\_cvector\\_foreach](#)

### 4.8.3 Function Documentation

#### 4.8.3.1 `CSCError csc_cvector_add ( cvector * v, void * elem )`

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that adding the element into the vector does **not** make the vector own the element. The user is still responsible for cleaning up that memory.

Both `elem` and `v` are expected to be **non-null**. This means that `NULL` elements are **not** allowed.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  worst case,  $O(1)$  amortized.

Parameters

<i>v</i>	the vector.
<i>elem</i>	the element to add.

Returns

On success, `CSCError::E_NOERR`. On memory allocation failure `CSCError::E_OUTOFMEM`.

#### 4.8.3.2 `void* csc_cvector_at ( const cvector * v, size_t idx )`

returns the element at the specified index.

This function performs a range check to ensure that `idx` is less than the size of the vector.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>v</i>	the vector.
<i>idx</i>	the index.

## Returns

the element at that index in the vector or `NULL` if the index is out of range.

4.8.3.3 `size_t csc_cvector_capacity ( const cvector * v )`

returns the capacity of the vector.

This function returns the number of elements the vector can hold before it needs to be resized.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>v</i>	the vector.
----------	-------------

## Returns

the capacity of the vector.

4.8.3.4 `cvector* csc_cvector_create ( )`

cvector "constructor" function

This function is used to create a `cvector`. If the function is successful, the function returns a pointer to a `cvector` created on the heap. If unsuccessful, `NULL` is returned.

## Returns

a pointer to a constructed `cvector`.

## See Also

[csc\\_cvector\\_destroy](#)

4.8.3.5 `void csc_cvector_destroy ( cvector * v )`

cvector "destructor" function

This function is used to clean up resources used by a `cvector` created via the [csc\\_cvector\\_create](#) function. This function must be called whenever a `cvector` is no longer used.

## See Also

[csc\\_cvector\\_create](#)

4.8.3.6 `bool csc_cvector_empty ( const cvector * v )`

checks if the vector is empty.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>v</i>	the vector.
----------	-------------

## Returns

`true` if the vector is empty. In other words, true if `csc_cvector_size(v) == 0`. Otherwise, false.

#### 4.8.3.7 void\* csc\_cvector\_find ( const cvector \* v, const void \* elem, csc\_compare cmp )

finds the element in the specified vector.

This function attempts to find `elem` using comparator `comp`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  average and worst case.

## Parameters

<i>v</i>	the vector.
<i>elem</i>	the element to find.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

## Returns

the element or `NULL` if the element couldn't be found.

#### 4.8.3.8 void csc\_cvector\_foreach ( cvector \* v, cvector\_foreach fn, void \* context )

applies the callback function to each element of the vector.

This callback function defines an operation that will be applied to each element of the `cvector`. The user may pass in additional context using the `context` param or pass in `NULL` if not required.

**Time Complexity:**  $O(n)$

## Parameters

<i>v</i>	the vector.
<i>fn</i>	the callback function to apply to each element. See <a href="#">cvector_foreach</a> .
<i>context</i>	user-defined data that will be applied to the callback. Can be <code>NULL</code> if unused.

## See Also

[cvector\\_foreach](#)

#### 4.8.3.9 CSCErr csc\_cvector\_reserve ( cvector \* v, size\_t num\_elems )

reserves memory for the specified number of elements in the vector.

This functions reserves enough memory in the vector such that it is able to hold *at least* `num_elems` without needing to expand. If the number of elements that will be contained in the vector is known or can be estimated, you may be able to improve the performance of your application by allocating the memory for the elements up front using this function. As always with performance, your mileage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is  $< \text{csc\_cvector\_size}(v)$ , that is an error.

All parameters are expected to be **non-null**.

**Time Complexity:** OS-specific.

## Parameters

<i>v</i>	the vector.
<i>num_elems</i>	the number of elements to allocate memory for.

## Returns

On success `CSCError::E_NOERR`. If the requested size is less than the current size, `CSCError::E_INVALIDOPERATION`. If there is a memory error, `CSCError::E_OUTOFMEM`.

4.8.3.10 `void csc_cvector_rm ( cvector * v, const void * elem, csc_compare cmp )`

removes an element from the vector.

This function removes `elem` from the supplied vector if it exists. In order to remove the element, the function must search for the element in the vector using the supplied `cmp` function.

All three parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$  best case,  $O(n)$  average and worst case.

## Parameters

<i>v</i>	the vector.
<i>elem</i>	the element to remove.
<i>cmp</i>	the comparison function to use. See <a href="#">csc_compare</a> for more details.

## See Also

[csc\\_compare](#)  
[csc\\_cvector\\_find](#)

4.8.3.11 `CSCError csc_cvector_rm_at ( cvector * v, size_t idx )`

removes the element at the specified 0-indexed index from the vector.

This function the element at index `idx` from `v`.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

<i>v</i>	the vector.
<i>idx</i>	the index.

## Returns

`CSCError::E_NOERR` or `CSCError::E_OUTOFRANGE` if the supplied index is out of range.

4.8.3.12 `CSCError csc_cvector_shrink_to_fit ( cvector * v )`

shrinks the capacity to match the size of the vector.

After a call to this function, the following will be true:

```
csc_cvector_size(v) == csc_cvector_capacity(v);
```

This function may be useful in low-memory settings where the vector's capacity greatly exceeds the size and the extra memory won't be required.

All parameters are expected to be **non-null**.

**Time Complexity:** OS-specific.

## Parameters

$v$	the vector.
-----	-------------

## Returns

On success `CSCError : : E_NOERR`. If there is a memory error, `CSCError : : E_OUTOFMEM`.

**4.8.3.13** `size_t csc_cvector_size ( const cvector * v )`

returns the size of the vector.

This function returns the number of elements currently in the vector.

All parameters are expected to be **non-null**.

**Time Complexity:**  $O(1)$

## Parameters

$v$	the vector.
-----	-------------

## Returns

the size of the vector.

# Index

/home/tamer/csc/src/cbitset.c, [9](#)  
/home/tamer/csc/src/cbitset.h, [13](#)  
/home/tamer/csc/src/cbst.c, [18](#)  
/home/tamer/csc/src/cbst.h, [23](#)  
/home/tamer/csc/src/csc.c, [28](#)  
/home/tamer/csc/src/csc.h, [31](#)  
/home/tamer/csc/src/cvector.c, [35](#)  
/home/tamer/csc/src/cvector.h, [41](#)  
\_node, [5](#)

## CSC\_UNUSED

csc.h, [33](#)

## CSCError

csc.h, [34](#)

## capacity

cvector, [7](#)

## cbitset, [5](#)

cbitset.h, [15](#)

data, [5](#)

nbits, [5](#)

size, [6](#)

## cbitset.c

csc\_cbitset\_at, [10](#)

csc\_cbitset\_clear, [11](#)

csc\_cbitset\_clear\_all, [11](#)

csc\_cbitset\_create, [11](#)

csc\_cbitset\_destroy, [11](#)

csc\_cbitset\_flip, [12](#)

csc\_cbitset\_set, [12](#)

csc\_cbitset\_set\_all, [12](#)

csc\_cbitset\_size, [13](#)

## cbitset.h

cbitset, [15](#)

csc\_cbitset\_at, [15](#)

csc\_cbitset\_clear, [16](#)

csc\_cbitset\_clear\_all, [16](#)

csc\_cbitset\_create, [16](#)

csc\_cbitset\_destroy, [16](#)

csc\_cbitset\_flip, [17](#)

csc\_cbitset\_set, [17](#)

csc\_cbitset\_set\_all, [17](#)

csc\_cbitset\_size, [18](#)

## cbst, [6](#)

cbst.h, [25](#)

## cbst.c

csc\_cbst\_add, [19](#)

csc\_cbst\_create, [20](#)

csc\_cbst\_destroy, [20](#)

csc\_cbst\_empty, [20](#)

csc\_cbst\_find, [20](#)

csc\_cbst\_foreach, [22](#)

csc\_cbst\_rm, [22](#)

csc\_cbst\_size, [22](#)

## cbst.h

cbst, [25](#)

csc\_cbst\_add, [25](#)

csc\_cbst\_create, [25](#)

csc\_cbst\_destroy, [26](#)

csc\_cbst\_empty, [26](#)

csc\_cbst\_find, [26](#)

csc\_cbst\_foreach, [27](#)

csc\_cbst\_rm, [27](#)

csc\_cbst\_size, [27](#)

## csc.h

E\_ERR\_N, [34](#)

E\_INVALIDOPERATION, [34](#)

E\_NOERR, [34](#)

E\_OUTOFMEM, [34](#)

E\_OUTOFRANGE, [34](#)

## csc.c

csc\_error\_str, [29](#)

csc\_swap, [29](#)

## csc.h

CSC\_UNUSED, [33](#)

CSCError, [34](#)

csc\_compare, [33](#)

csc\_error\_str, [35](#)

csc\_foreach, [33](#)

csc\_swap, [35](#)

## csc\_cbitset\_at

cbitset.c, [10](#)

cbitset.h, [15](#)

## csc\_cbitset\_clear

cbitset.c, [11](#)

cbitset.h, [16](#)

## csc\_cbitset\_clear\_all

cbitset.c, [11](#)

cbitset.h, [16](#)

## csc\_cbitset\_create

cbitset.c, [11](#)

cbitset.h, [16](#)

## csc\_cbitset\_destroy

cbitset.c, [11](#)

cbitset.h, [16](#)

## csc\_cbitset\_flip

cbitset.c, [12](#)

cbitset.h, [17](#)

## csc\_cbitset\_set

cbitset.c, [12](#)



- cbitset.h, [17](#)
- csc\_cbitset\_set\_all
  - cbitset.c, [12](#)
  - cbitset.h, [17](#)
- csc\_cbitset\_size
  - cbitset.c, [13](#)
  - cbitset.h, [18](#)
- csc\_cbst\_add
  - cbst.c, [19](#)
  - cbst.h, [25](#)
- csc\_cbst\_create
  - cbst.c, [20](#)
  - cbst.h, [25](#)
- csc\_cbst\_destroy
  - cbst.c, [20](#)
  - cbst.h, [26](#)
- csc\_cbst\_empty
  - cbst.c, [20](#)
  - cbst.h, [26](#)
- csc\_cbst\_find
  - cbst.c, [20](#)
  - cbst.h, [26](#)
- csc\_cbst\_foreach
  - cbst.c, [22](#)
  - cbst.h, [27](#)
- csc\_cbst\_rm
  - cbst.c, [22](#)
  - cbst.h, [27](#)
- csc\_cbst\_size
  - cbst.c, [22](#)
  - cbst.h, [27](#)
- csc\_compare
  - csc.h, [33](#)
- csc\_cvector\_add
  - cvector.c, [37](#)
  - cvector.h, [44](#)
- csc\_cvector\_at
  - cvector.c, [37](#)
  - cvector.h, [44](#)
- csc\_cvector\_capacity
  - cvector.c, [37](#)
  - cvector.h, [45](#)
- csc\_cvector\_create
  - cvector.c, [38](#)
  - cvector.h, [45](#)
- csc\_cvector\_destroy
  - cvector.c, [38](#)
  - cvector.h, [45](#)
- csc\_cvector\_empty
  - cvector.c, [38](#)
  - cvector.h, [45](#)
- csc\_cvector\_find
  - cvector.c, [38](#)
  - cvector.h, [46](#)
- csc\_cvector\_foreach
  - cvector.c, [39](#)
  - cvector.h, [46](#)
- csc\_cvector\_reserve
  - cvector.c, [39](#)
  - cvector.h, [46](#)
- cvector.c, [39](#)
- cvector.h, [46](#)
- csc\_cvector\_rm
  - cvector.c, [40](#)
  - cvector.h, [47](#)
- csc\_cvector\_rm\_at
  - cvector.c, [40](#)
  - cvector.h, [47](#)
- csc\_cvector\_shrink\_to\_fit
  - cvector.c, [40](#)
  - cvector.h, [47](#)
- csc\_cvector\_size
  - cvector.c, [41](#)
  - cvector.h, [49](#)
- csc\_error\_str
  - csc.c, [29](#)
  - csc.h, [35](#)
- csc\_foreach
  - csc.h, [33](#)
- csc\_swap
  - csc.c, [29](#)
  - csc.h, [35](#)
- cvector, [6](#)
  - capacity, [7](#)
  - cvector.h, [43](#)
  - data, [7](#)
  - size, [7](#)
- cvector.c
  - csc\_cvector\_add, [37](#)
  - csc\_cvector\_at, [37](#)
  - csc\_cvector\_capacity, [37](#)
  - csc\_cvector\_create, [38](#)
  - csc\_cvector\_destroy, [38](#)
  - csc\_cvector\_empty, [38](#)
  - csc\_cvector\_find, [38](#)
  - csc\_cvector\_foreach, [39](#)
  - csc\_cvector\_reserve, [39](#)
  - csc\_cvector\_rm, [40](#)
  - csc\_cvector\_rm\_at, [40](#)
  - csc\_cvector\_shrink\_to\_fit, [40](#)
  - csc\_cvector\_size, [41](#)
- cvector.h
  - csc\_cvector\_add, [44](#)
  - csc\_cvector\_at, [44](#)
  - csc\_cvector\_capacity, [45](#)
  - csc\_cvector\_create, [45](#)
  - csc\_cvector\_destroy, [45](#)
  - csc\_cvector\_empty, [45](#)
  - csc\_cvector\_find, [46](#)
  - csc\_cvector\_foreach, [46](#)
  - csc\_cvector\_reserve, [46](#)
  - csc\_cvector\_rm, [47](#)
  - csc\_cvector\_rm\_at, [47](#)
  - csc\_cvector\_shrink\_to\_fit, [47](#)
  - csc\_cvector\_size, [49](#)
  - cvector, [43](#)
  - cvector\_foreach, [44](#)
- cvector\_foreach

cvector.h, [44](#)

#### data

cbitsset, [5](#)

cvector, [7](#)

#### E\_ERR\_N

csc.h, [34](#)

#### E\_INVALIDOPERATION

csc.h, [34](#)

#### E\_NOERR

csc.h, [34](#)

#### E\_OUTOFMEM

csc.h, [34](#)

#### E\_OUTOFRANGE

csc.h, [34](#)

#### nbits

cbitsset, [5](#)

#### size

cbitsset, [6](#)

cvector, [7](#)