

CSC

1.0.0

Generated by Doxygen 1.8.6

Sun Dec 30 2018 21:57:59

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	3
2.1	File List	3
3	Data Structure Documentation	5
3.1	cbitset Struct Reference	5
3.1.1	Field Documentation	5
3.1.1.1	data	5
3.1.1.2	nbits	5
3.1.1.3	size	5
3.2	cvector Struct Reference	5
3.2.1	Field Documentation	6
3.2.1.1	capacity	6
3.2.1.2	data	6
3.2.1.3	size	6
4	File Documentation	7
4.1	/home/tamer/csc/src/cbitset.c File Reference	7
4.1.1	Detailed Description	8
4.1.2	Function Documentation	8
4.1.2.1	csc_cbitset_at	8
4.1.2.2	csc_cbitset_clear	9
4.1.2.3	csc_cbitset_clear_all	9
4.1.2.4	csc_cbitset_create	9
4.1.2.5	csc_cbitset_destroy	10
4.1.2.6	csc_cbitset_flip	10
4.1.2.7	csc_cbitset_set	10
4.1.2.8	csc_cbitset_set_all	10
4.1.2.9	csc_cbitset_size	11
4.2	/home/tamer/csc/src/cbitset.h File Reference	11

4.2.1	Detailed Description	12
4.2.2	Typedef Documentation	13
4.2.2.1	cbitset	13
4.2.3	Function Documentation	13
4.2.3.1	csc_cbitset_at	13
4.2.3.2	csc_cbitset_clear	14
4.2.3.3	csc_cbitset_clear_all	14
4.2.3.4	csc_cbitset_create	14
4.2.3.5	csc_cbitset_destroy	15
4.2.3.6	csc_cbitset_flip	15
4.2.3.7	csc_cbitset_set	15
4.2.3.8	csc_cbitset_set_all	15
4.2.3.9	csc_cbitset_size	16
4.3	/home/tamer/csc/src/csc.c File Reference	16
4.3.1	Detailed Description	17
4.3.2	Macro Definition Documentation	17
4.3.2.1	CSC_DEFINE_BUILTIN_CMP	17
4.3.3	Function Documentation	17
4.3.3.1	csc_error_str	17
4.3.3.2	csc_swap	18
4.4	/home/tamer/csc/src/csc.h File Reference	18
4.4.1	Detailed Description	20
4.4.2	Macro Definition Documentation	20
4.4.2.1	CSC_DECLARE_BUILTIN_CMP	20
4.4.2.2	CSC_MAX_ERROR_MSG_LEN	20
4.4.2.3	CSC_UNUSED	20
4.4.3	Typedef Documentation	20
4.4.3.1	csc_compare	20
4.4.3.2	CSCError	21
4.4.4	Enumeration Type Documentation	21
4.4.4.1	CSCError	21
4.4.5	Function Documentation	22
4.4.5.1	csc_error_str	22
4.4.5.2	csc_swap	22
4.5	/home/tamer/csc/src/cvector.c File Reference	22
4.5.1	Detailed Description	24
4.5.2	Function Documentation	24
4.5.2.1	csc_cvector_add	24
4.5.2.2	csc_cvector_at	24
4.5.2.3	csc_cvector_capacity	25

4.5.2.4	csc_cvector_create	25
4.5.2.5	csc_cvector_destroy	25
4.5.2.6	csc_cvector_empty	25
4.5.2.7	csc_cvector_find	26
4.5.2.8	csc_cvector_foreach	26
4.5.2.9	csc_cvector_reserve	26
4.5.2.10	csc_cvector_rm	27
4.5.2.11	csc_cvector_rm_at	27
4.5.2.12	csc_cvector_shrink_to_fit	27
4.5.2.13	csc_cvector_size	28
4.6	/home/tamer/csc/src/cvector.h File Reference	28
4.6.1	Detailed Description	30
4.6.2	Typedef Documentation	30
4.6.2.1	cvector	31
4.6.2.2	cvector_foreach	31
4.6.3	Function Documentation	31
4.6.3.1	csc_cvector_add	31
4.6.3.2	csc_cvector_at	31
4.6.3.3	csc_cvector_capacity	32
4.6.3.4	csc_cvector_create	32
4.6.3.5	csc_cvector_destroy	32
4.6.3.6	csc_cvector_empty	32
4.6.3.7	csc_cvector_find	33
4.6.3.8	csc_cvector_foreach	33
4.6.3.9	csc_cvector_reserve	33
4.6.3.10	csc_cvector_rm	34
4.6.3.11	csc_cvector_rm_at	34
4.6.3.12	csc_cvector_shrink_to_fit	34
4.6.3.13	csc_cvector_size	36
	Index	37

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

cbitsset	5
cvector	5

Chapter 2

File Index

2.1 File List

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

/home/tamer/csc/src/ cbitset.c	
Implementation of the cbitset data structure	7
/home/tamer/csc/src/ cbitset.h	
Defines the interface to the cbitset data structure	11
/home/tamer/csc/src/ csc.c	
Implementation file for csc.h	16
/home/tamer/csc/src/ csc.h	
Main include file for the csc library	18
/home/tamer/csc/src/ cvector.c	
Implementation of the cvector structure	22
/home/tamer/csc/src/ cvector.h	
Interface of the cvector structure	28

Chapter 3

Data Structure Documentation

3.1 cbitset Struct Reference

Data Fields

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

3.1.1 Field Documentation

3.1.1.1 `bitset_type* cbitset::data`

The internal data of the bitset.

3.1.1.2 `size_t cbitset::nbits`

The number of bits the bitset can hold.

3.1.1.3 `size_t cbitset::size`

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

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

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

3.2 cvector Struct Reference

Data Fields

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

3.2.1 Field Documentation

3.2.1.1 `size_t cvector::capacity`

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

3.2.1.2 `void** cvector::data`

The internal data store of the vector.

3.2.1.3 `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`

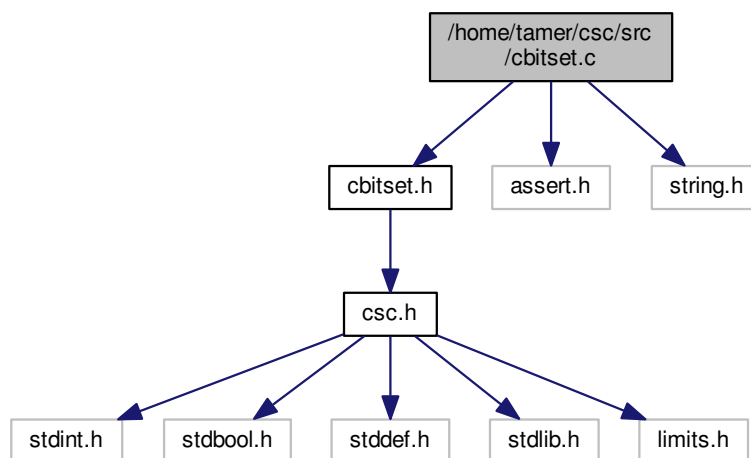
Chapter 4

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>	optional 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.

4.1.2.2 `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

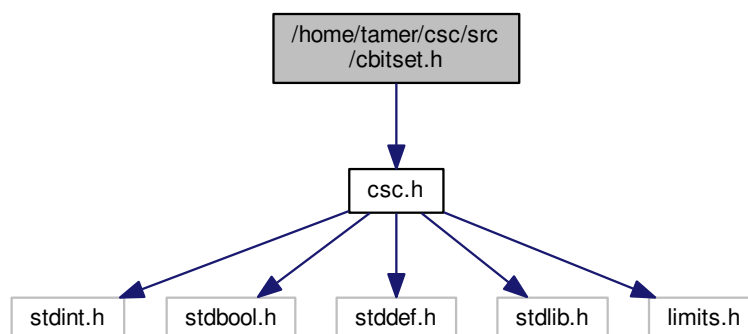
the number of bits the bitset is capable of holding.

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

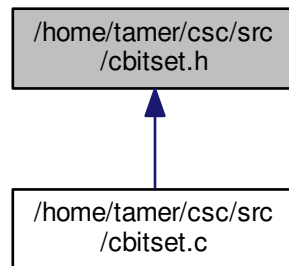
defines the interface to the [cbitset](#) data structure.

```
#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>	optional 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.

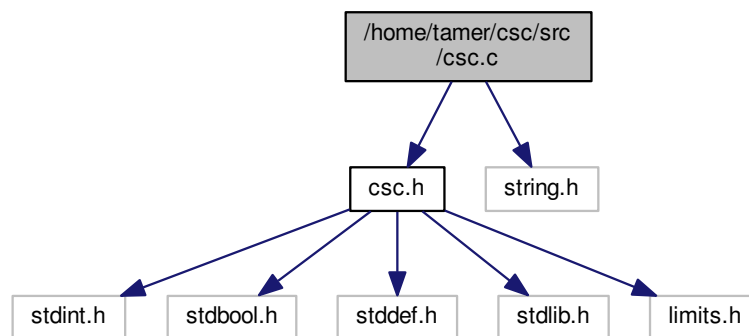
4.3 `/home/tamer/csc/src/csc.c` File Reference

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.

4.3.1 Detailed Description

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

Author

Tamer Aly

Date

27 Dec 2018

See Also

[csc.h](#)

4.3.2 Macro Definition Documentation

4.3.2.1 #define CSC_DECLARE_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](#)

4.3.3 Function Documentation

4.3.3.1 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 non-null .
<i>len</i>	the length of the buffer. Recommended to be \geq CSC_MAX_ERROR_MSG_LEN .

See Also

[CSC_MAX_ERROR_MSG_LEN](#)

4.3.3.2 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

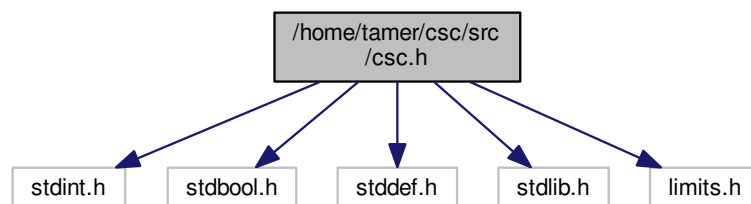
<i>a</i>	A is the first elem.
<i>b</i>	B is the second elem.

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

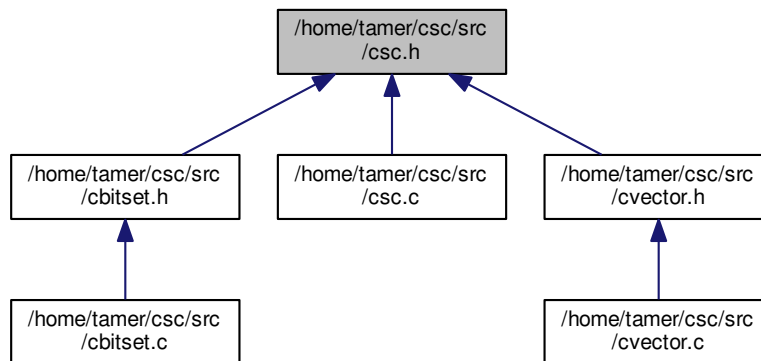
the main include file for the csc library.

```
#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

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)

4.4.1 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.

4.4.2 Macro Definition Documentation

4.4.2.1 `#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](#)

4.4.2.2 `#define CSC_MAX_ERROR_MSG_LEN 128`

the maximum message length a `CSCError` is guaranteed to generate.

See Also

[csc_error_str](#)

4.4.2.3 `#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.

4.4.3 Typedef Documentation

4.4.3.1 `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 < 1 means a is less than b . a return value of > 1 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](#)

4.4.3.2 typedef enum CSCErrors CSCErrors

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 `CSCErrors : : E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCErrors : : E_ERR_N`, indicates an error condition.

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

```
CSCErrors 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](#)

4.4.4 Enumeration Type Documentation

4.4.4.1 enum CSCErrors

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 `CSCErrors : : E_NOERR` indicates a successful operation. Any other error, with the exception of `CSCErrors : : E_ERR_N`, indicates an error condition.

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

```
CSCErrors 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.

4.4.5 Function Documentation

4.4.5.1 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

<i>e</i>	the error.
<i>buf</i>	the buffer to fill. Must be non-null .
<i>len</i>	the length of the buffer. Recommended to be \geq CSC_MAX_ERROR_MSG_LEN .

See Also

[CSC_MAX_ERROR_MSG_LEN](#)

4.4.5.2 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.

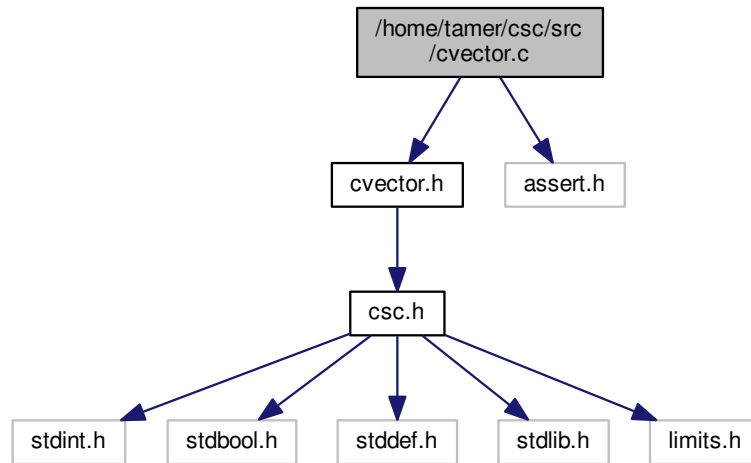
Parameters

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

4.5 /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.

4.5.1 Detailed Description

contains the implementation of the cvector structure.

Author

Tamer Aly

Date

27 Dec 2018

4.5.2 Function Documentation

4.5.2.1 `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`.

4.5.2.2 `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.

4.5.2.3 `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.

4.5.2.4 `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](#)

4.5.2.5 `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](#)

4.5.2.6 `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`.

4.5.2.7 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

<code>v</code>	the vector.
<code>elem</code>	the element to find.
<code>cmp</code>	the comparison function to use. See csc_compare for more details.

Returns

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

4.5.2.8 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

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

See Also

[cvector_foreach](#)

4.5.2.9 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

<code>v</code>	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`.

4.5.2.10 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 csc_compare for more details.

See Also

[csc_compare](#)
[csc_cvector_find](#)

4.5.2.11 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.

4.5.2.12 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`.

4.5.2.13 `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

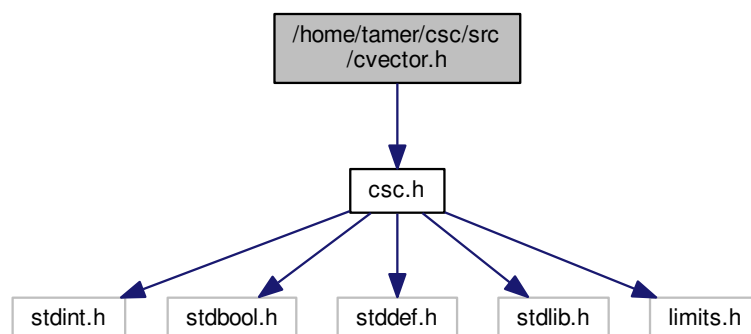
the size of the vector.

4.6 `/home/tamer/csc/src/cvector.h` File Reference

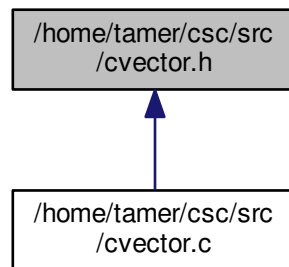
contains interface of the cvector structure.

```
#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.

4.6.1 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;

    CSCError 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
CSCError 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);
}
```

4.6.2 Typedef Documentation

4.6.2.1 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](#)

4.6.2.2 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](#)

4.6.3 Function Documentation

4.6.3.1 CSCErrors 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, `CSCErrors::E_NOERR`. On memory allocation failure `CSCErrors::E_OUTOFMEM`.

4.6.3.2 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.

4.6.3.3 `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.

4.6.3.4 `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](#)

4.6.3.5 `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](#)

4.6.3.6 `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.

4.6.3.7 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 csc_compare for more details.

Returns

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

4.6.3.8 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 cvector_foreach .
<i>context</i>	user-defined data that will be applied to the callback. Can be NULL if unused.

See Also

[cvector_foreach](#)

4.6.3.9 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`.

4.6.3.10 `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 csc_compare for more details.

See Also

[csc_compare](#)
[csc_cvector_find](#)

4.6.3.11 `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.

4.6.3.12 `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`.

4.6.3.13 `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, [7](#)
/home/tamer/csc/src/cbitset.h, [11](#)
/home/tamer/csc/src/csc.c, [16](#)
/home/tamer/csc/src/csc.h, [18](#)
/home/tamer/csc/src/cvector.c, [22](#)
/home/tamer/csc/src/cvector.h, [28](#)

CSC_UNUSED

csc.h, [20](#)

CSCError

csc.h, [21](#)

capacity

cvector, [6](#)

cbitset, [5](#)

cbitset.h, [13](#)

data, [5](#)

nbits, [5](#)

size, [5](#)

cbitset.c

csc_cbitset_at, [8](#)

csc_cbitset_clear, [9](#)

csc_cbitset_clear_all, [9](#)

csc_cbitset_create, [9](#)

csc_cbitset_destroy, [9](#)

csc_cbitset_flip, [10](#)

csc_cbitset_set, [10](#)

csc_cbitset_set_all, [10](#)

csc_cbitset_size, [11](#)

cbitset.h

cbitset, [13](#)

csc_cbitset_at, [13](#)

csc_cbitset_clear, [14](#)

csc_cbitset_clear_all, [14](#)

csc_cbitset_create, [14](#)

csc_cbitset_destroy, [14](#)

csc_cbitset_flip, [15](#)

csc_cbitset_set, [15](#)

csc_cbitset_set_all, [15](#)

csc_cbitset_size, [16](#)

csc.h

E_ERR_N, [22](#)

E_INVALIDOPERATION, [22](#)

E_NOERR, [22](#)

E_OUTOFMEM, [22](#)

E_OUTOFRANGE, [22](#)

csc.c

csc_error_str, [17](#)

csc_swap, [18](#)

csc.h

CSC_UNUSED, [20](#)

CSCError, [21](#)

csc_compare, [20](#)

csc_error_str, [22](#)

csc_swap, [22](#)

csc_cbitset_at

cbitset.c, [8](#)

cbitset.h, [13](#)

csc_cbitset_clear

cbitset.c, [9](#)

cbitset.h, [14](#)

csc_cbitset_clear_all

cbitset.c, [9](#)

cbitset.h, [14](#)

csc_cbitset_create

cbitset.c, [9](#)

cbitset.h, [14](#)

csc_cbitset_destroy

cbitset.c, [9](#)

cbitset.h, [14](#)

csc_cbitset_flip

cbitset.c, [10](#)

cbitset.h, [15](#)

csc_cbitset_set

cbitset.c, [10](#)

cbitset.h, [15](#)

csc_cbitset_set_all

cbitset.c, [10](#)

cbitset.h, [15](#)

csc_cbitset_size

cbitset.c, [11](#)

cbitset.h, [16](#)

csc_compare

csc.h, [20](#)

csc_cvector_add

cvector.c, [24](#)

cvector.h, [31](#)

csc_cvector_at

cvector.c, [24](#)

cvector.h, [31](#)

csc_cvector_capacity

cvector.c, [24](#)

cvector.h, [32](#)

csc_cvector_create

cvector.c, [25](#)

cvector.h, [32](#)

csc_cvector_destroy

cvector.c, [25](#)

cvector.h, [32](#)

csc_cvector_empty

- cvector.c, 25
 - cvector.h, 32
- csc_cvector_find
 - cvector.c, 25
 - cvector.h, 33
- csc_cvector_foreach
 - cvector.c, 26
 - cvector.h, 33
- csc_cvector_reserve
 - cvector.c, 26
 - cvector.h, 33
- csc_cvector_rm
 - cvector.c, 27
 - cvector.h, 34
- csc_cvector_rm_at
 - cvector.c, 27
 - cvector.h, 34
- csc_cvector_shrink_to_fit
 - cvector.c, 27
 - cvector.h, 34
- csc_cvector_size
 - cvector.c, 28
 - cvector.h, 36
- csc_error_str
 - csc.c, 17
 - csc.h, 22
- csc_swap
 - csc.c, 18
 - csc.h, 22
- cvector, 5
 - capacity, 6
 - cvector.h, 30
 - data, 6
 - size, 6
- cvector.c
 - csc_cvector_add, 24
 - csc_cvector_at, 24
 - csc_cvector_capacity, 24
 - csc_cvector_create, 25
 - csc_cvector_destroy, 25
 - csc_cvector_empty, 25
 - csc_cvector_find, 25
 - csc_cvector_foreach, 26
 - csc_cvector_reserve, 26
 - csc_cvector_rm, 27
 - csc_cvector_rm_at, 27
 - csc_cvector_shrink_to_fit, 27
 - csc_cvector_size, 28
- cvector.h
 - csc_cvector_add, 31
 - csc_cvector_at, 31
 - csc_cvector_capacity, 32
 - csc_cvector_create, 32
 - csc_cvector_destroy, 32
 - csc_cvector_empty, 32
 - csc_cvector_find, 33
 - csc_cvector_foreach, 33
 - csc_cvector_reserve, 33
 - csc_cvector_rm, 34
 - csc_cvector_rm_at, 34
 - csc_cvector_shrink_to_fit, 34
 - csc_cvector_size, 36
 - cvector, 30
 - cvector_foreach, 31
- cvector_foreach
 - cvector.h, 31
- data
 - cbitset, 5
 - cvector, 6
- E_ERR_N
 - csc.h, 22
- E_INVALIDOPERATION
 - csc.h, 22
- E_NOERR
 - csc.h, 22
- E_OUTOFMEM
 - csc.h, 22
- E_OUTOFRANGE
 - csc.h, 22
- nbits
 - cbitset, 5
- size
 - cbitset, 5
 - cvector, 6