csc

1.0.0

Generated by Doxygen 1.8.6

Sat Dec 29 2018 12:11:27

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

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

Chapter 3

Data Structure Documentation

3.1 cvector Struct Reference

Data Fields

- void ∗∗ data
- size_t size
- size_t capacity

3.1.1 Field Documentation

3.1.1.1 size_t cvector::capacity

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

3.1.1.2 void∗∗ cvector::data

The internal data store of the vector.

3.1.1.3 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

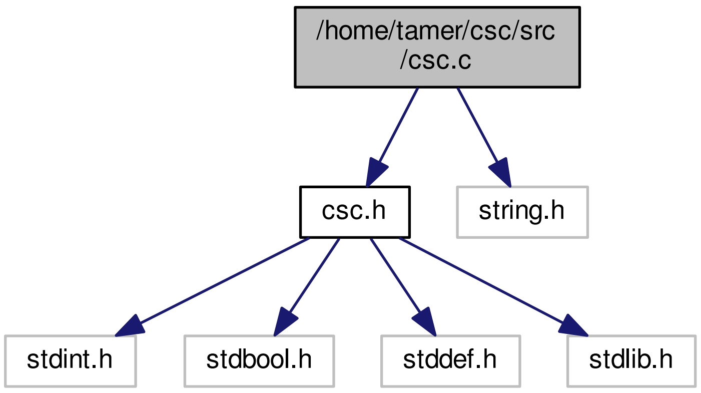
File Documentation

4.1 /home/tamer/csc/src/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.

4.1.1 Detailed Description

the implementation file for csc.h.

Author

Tamer Aly

Date

27 Dec 2018

See Also

csc.h

4.1.2 Macro Definition Documentation

4.1.2.1 #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

4.1.3 Function Documentation

4.1.3.1 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

e	the error.
buf	the buffer to fill. Must be **non-null**.
len	the length of the buffer. Recommended to be >= CSC_MAX_ERROR_MSG_LEN.

See Also

CSC_MAX_ERROR_MSG_LEN

4.1.3.2 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

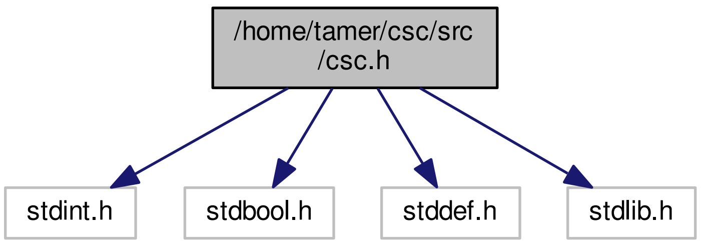
a	A is the first elem.
b	B is the second elem.

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

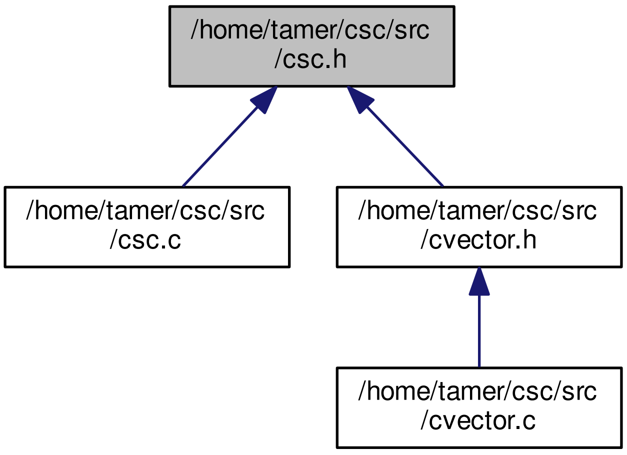
the main include file for the csc library.

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

Include dependency graph for csc.h:



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


/home/tamer/csc/src /csc.h
/home/tamer/csc/src /csc.c
/home/tamer/csc/src /cvector.h
/home/tamer/csc/src /cvector.c

Macros

- #define CSC_UNUSED(x) (void)x

    macro that silences compiler warnings about unused function parameters.
- #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)

4.2.1 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.

4.2.2 Macro Definition Documentation

4.2.2.1 #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

4.2.2.2 #define CSC_MAX_ERROR_MSG_LEN 128

the maximum message length a CSCError is guaranteed to generate.

See Also

csc_error_str

4.2.2.3 #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.

4.2.3 Typedef Documentation

4.2.3.1 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

4.2.3.2 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 condition.
}
```

4.2.4 Enumeration Type Documentation

4.2.4.1 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 condition.
}
```

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.

4.2.5 Function Documentation

4.2.5.1 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

e	the error.
buf	the buffer to fill. Must be **non-null**.
len	the length of the buffer. Recommended to be >= CSC_MAX_ERROR_MSG_LEN.

See Also

CSC_MAX_ERROR_MSG_LEN

4.2.5.2 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

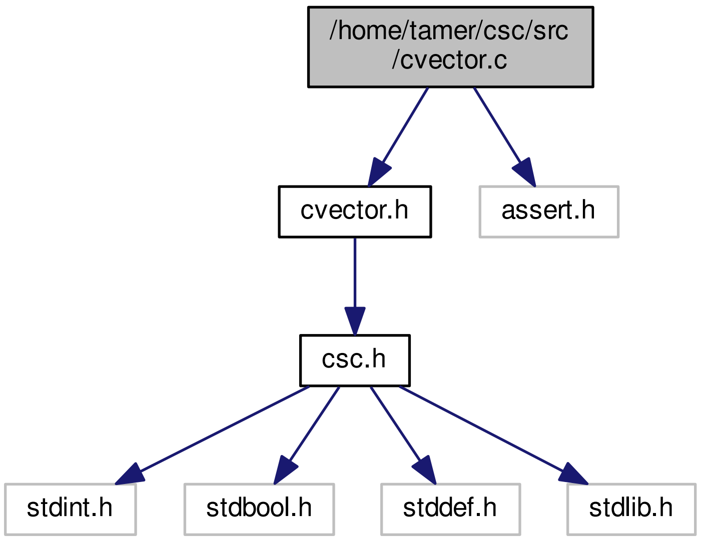
a	A is the first elem.
b	B is the second elem.

4.3 /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.

4.3.1 Detailed Description

contains the implementation of the cvector structure.

Author

Tamer Aly

Date

27 Dec 2018

4.3.2 Function Documentation

4.3.2.1 CSCError csc_cvector_add (cvector ∗ *v*, void ∗ *elem*)

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that `elem` **MUST** point to an element allocated on the heap.

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

v	the vector.
elem	the element to add.

Returns

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

4.3.2.2 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

v	the vector.
idx	the index.

Returns

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

4.3.2.3 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

v	the vector.

Returns

the capacity of the vector.

4.3.2.4 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

4.3.2.5 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

4.3.2.6 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

v	the vector.

Returns

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

4.3.2.7 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

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

Returns

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

4.3.2.8 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

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

See Also

cvector_foreach

4.3.2.9 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 milage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is < `csc_cvector_size(v)`, that is an error.

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

Time Complexity: OS-specific.

Parameters

v	the vector.

num_elems	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`.

4.3.2.10 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

v	the vector.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

See Also

csc_compare
csc_cvector_find

4.3.2.11 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

v	the vector.
idx	the index.

Returns

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

4.3.2.12 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

v	the vector.

Returns

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

4.3.2.13 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

v	the vector.

Returns

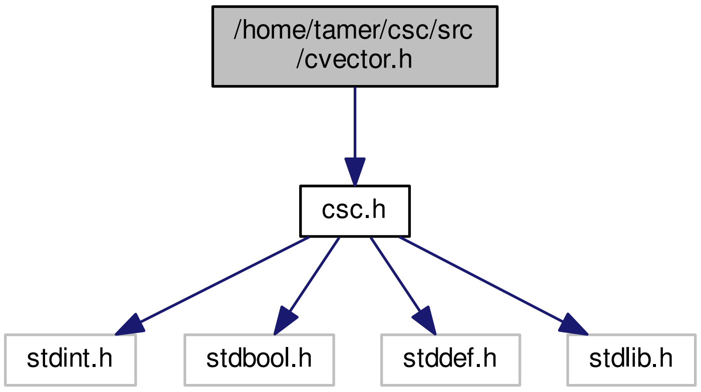
the size of the vector.

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

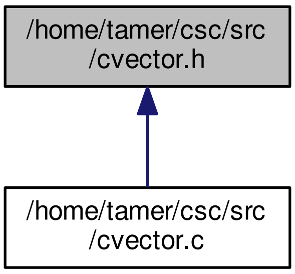
contains interface of the cvector structure.

`#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.

4.4.1 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. Note that they must be on the heap and that the vector "owns" the
       element.
for (int i = 0; i < 10; i++) {
     int* x = malloc(sizeof(*x));
     if (x == NULL) {
         // couldn't allocate memory.
     }
     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
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);
}
```

4.4.2 Typedef Documentation

4.4.2.1 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

4.4.2.2 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

elem	the element to process
context	user-defined data that can be passed into the function. Can be `NULL` if unused.

See Also

csc_cvector_foreach

4.4.3 Function Documentation

4.4.3.1 CSCError csc_cvector_add (cvector ∗ *v*, void ∗ *elem*)

adds an element into the vector.

This function adds `elem` into the supplied vector. Note that `elem` **MUST** point to an element allocated on the heap.

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

v	the vector.
elem	the element to add.

Returns

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

4.4.3.2 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

v	the vector.
idx	the index.

Returns

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

4.4.3.3 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

v	the vector.

Returns

the capacity of the vector.

4.4.3.4 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

4.4.3.5 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

4.4.3.6 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

v	the vector.

Returns

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

4.4.3.7 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

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

Returns

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

4.4.3.8 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

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

See Also

cvector_foreach

4.4.3.9 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 milage may vary.

Note that memory truncation is **not** allowed. That is, if `num_elems` is $<$ `csc_cvector_size(v)`, that is an error.

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

Time Complexity: OS-specific.

Parameters

v	the vector.

num_elems	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`.

4.4.3.10 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

v	the vector.
elem	the element to remove.
cmp	the comparison function to use. See csc_compare for more details.

See Also

csc_compare
csc_cvector_find

4.4.3.11 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

v	the vector.
idx	the index.

Returns

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

4.4.3.12 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`.

4.4.3.13 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