The Memory Management Library

0.2.1

Generated by Doxygen 1.5.8

Mon Jan 24 01:12:38 2011

Contents

Chapter 1

C Basic Library: Memory Management Library

Version:

0.2.1

Author:

Jun Woong (woong.jun at gmail.com)

Date:

last modified on 2010-01-24

1.1 Introduction

This document specifies the Memory Management Library which belongs to the C Basic Library. The basic structure is from David Hanson's book, "C Interfaces and Implementations." I modifies the original implementation to add missing but useful facilities, to make it conform to the C standard and to enhance its readibility; for example a prefix is used more strictly in order to avoid the user namespace pollution.

Since the book explains its design and implementation in a very comprehensive way, not to mention the copyright issues, it is nothing but waste to repeat it here, so I finish this document by giving introduction to the library, including explanation on its two versions, one for production code and the other for debugging code. How to use the facilities is deeply explained in files that define them.

The Memory Management Library reserves identifiers starting with `mem_` and `MEM_`, and imports the Assertion Library and the Exception Handling Library.

1.2 How to Use The Library

The Memory Management Library is intended to substitute calls to the memory allocation/deallcation functions like malloc() provided by <stdlib.h>. Its main purpose is to enhance their safety by making them:

- never return a null pointer in any case, which eliminates handling an exceptional case after memory allocation; failing allocation results in raising an exception (that can be handled by the Exception Handling Library) rather than in returning a null pointer, and
- set a freed pointer to null, which helps preventing the pointer from being used further.

The following example shows a typical case to allocate/deallocate the storage for a type that a pointer `p` points to:

```
{
    type_t *p;
    MEM_NEW(p);
    ...
    MEM_FREE(p);
}
```

The user code does not need to check the returned value from MEM_NEW(), because if the allocation fails, in which case the standard's malloc() returns the null pointer, an exception named `mem_exceptfail` raised. If you want to do something when the memory allocation fails, simply establish its handler in a proper place as follows.

```
EXCEPT_TRY
    ... code containing call to allocation functions ...
EXCEPT_EXCEPT(mem_exceptfail)
    ... code handling allocation failure ...
EXCEPT_END
```

MEM_NEW0() is also provided to do the same job as MEM_NEW() except that the allocated storage is initialized to zero-valued bytes.

MEM_FREE() requires that a given pointer be an lvalue, and assigns a null pointer to it after deallocation. This means that a user should use a temporary object when having only a pointer value as opposed to an object containing the value, but its benefit that the freed pointer is prevented from being misused seems overwhelming the inconvenience.

MEM_RESIZE() that is intended to be a wrapper for realloc() differs from realloc() in that its job is limited to adjusting the size of an allocated area; realloc() allocates as malloc() when a given pointer is a null pointer, and deallocates as free() when a given size is 0. Thus, a pointer given to MEM_RESIZE() has to be non-null and a size greater than 0. The justification for the limitation is given in the book from which this library comes.

MEM_ALLOC() and MEM_CALLOC() are simple wrappers for malloc() and calloc(), and their major difference from the original functions is, of course, that allocation failure results in raising an exception.

1.2.1 Two Versions

This library is provided as two versions, one for production code (memory.c) and the other for debugging code (memoryd.c). Two versions offers exactly the same interfaces and only their implementations differ. During debugging code, linking the debugging version is helpful when you want to figure out if there are invalid memory usages like a free-free problem (that is, trying to release an already-deallocated area) and a memory leakage. This does not cover the whole range of such problems as valgrind does, but if there are no other tools available for catching memory problems, the debugging version of this library would be useful. Unfortunately, the debugging version is not able to keep track of memory usage unless done through this library; for example, an invalid operation applied to the storage allocated via malloc() goes undetected.

1.2.2 Debugging Version

As explained, the debugging version catches certain invalid memory usage. The full list includes:

- freeing an unallocated area
- resizing an unallocated area and
- listing allocated areas at a given time.

The function implemented in the debugging version print out no diagnostics unless mem_log() is invoked properly. You can get the list of allocated areas by calling mem_leak() after properly invoking mem_log().

The diagnostic message printed when an assertion failed changed in C99 to include the name of the function in which it failed. This can be readily attained with a newly introduced predefined identifier `__func__`. To provide more useful information, if an implementation claims to support C99 by defining the macro `__STDC_VERSION__` properly, the library also includes the function name when making up the message output when an uncaught exception detected.

1.2.3 Product Version

Even if the product version does not track the memory problems that the debugging version does, mem_log() and mem_leak() are provided as dummy functions for convenience. See the functions for more details.

1.2.4 Some Caveats

In the implementation of the debugging version, `MEM_MAXALIGN` plays an important role; it is intended to specify the alignment factor of pointers malloc() returns; without that, a valid memory operation might be mistaken as an invalid one and stop a running program issuing a wrong diagnostic message. If `MEM_MAXALIGN` not defined, the library tries to guess a proper value, but it is not guaranteed for the guess to be always correct. Thus, when compiling the library, giving an explicit definition of `MEM_MAXALIGN` (via a compiler option like -D, if available) is recommended.

MEM_ALLOC() and MEM_CALLOC() have the same interfaces as malloc() and calloc() respectively, and thus their return values should be stored. On the other hand, MEM_NEW() and MEM_RESIZE(), even if they act as if returning a pointer value, modify a given pointer as the result. This means that a user codes like:

```
type_t *p;
p = MEM_NEW(p);
```

might unconsciously trigger undefined behavior since between two sequence points `p` is modified twice. So remember that any MEM_ functions taking a pointer (including MEM_FREE()) modify the pointer and a user code need not to store explicitly the result to the pointer.

1.3 Boilerplate Code

No boilerplate code is provided for this library.

1.4 Future Directions

1.4.1 Minor Changes

To mimic the behavior of calloc() specified by the standard, it is required for the debugging version of MEM_CALLOC() to successfully return a null pointer when it cannot allocate the storage of the requested size. Since this does not allow overflow, it has to carefully check overflow when calculating the total size.

1.5 Contact Me

Visit `http://project.woong.org` to get the lastest version of this library. Only a small portion of my homepage (`http://www.woong.org`) is maintained in English, thus one who is not good at Korean would have difficulty when navigating most of other pages served in Korean. If you think the information you are looking for is on pages written in Korean you cannot read, do not hesitate to send me an email asking for help.

Any comments about the library are welcomed. If you have a proposal or question on the library just email me, and then I will reply as soon as possible.

1.6 Copyright

I do not wholly hold the copyright of this library; it is mostly held by David Hanson as stated in his book, "C: Interfaces and Implementations:"

Copyright (c) 1994,1995,1996,1997 by David R. Hanson.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

For the parts I added or modified, the following applies:

Copyright (C) 2009-2011 by Jun Woong.

This package is a memory management implementation by Jun Woong. The implementation was written so as to conform with the Standard C published by ISO 9899:1990 and ISO 9899:1999.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Todo List

Global mem_calloc Improvements are possible and planned:

- the C standard requires calloc() return a null pointer if it can allocates no storage of the size `c * n` in bytes, which allows no overflow in computing the multiplication. So overflow checking is necessary to mimic the behavior of calloc().

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

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

Chapter 5

Data Structure Documentation

5.1 mem_loginfo_t Struct Reference

contains the information about invalid memory operations.

`#include <memory.h>`

Data Fields

- const void $*$ p
- size_t size
- const char $*$ ifile
- const char $*$ ifunc
- int iline
- const char $*$ afile
- const char $*$ afunc
- int aline
- size_t asize

5.1.1 Detailed Description

contains the information about invalid memory operations.

An object of the type `mem_loginfo_t` is used when the information about an invalid memory operation is delivered to a user-provided log function. As explained in mem_log(), such a function must be declared to accept a `mem_loginfo_t` arguments.

Its members contains three kinds of information:

- the information about an invalid memory operation. For example, if mem_free() is invoked for the storage that is already deallocated, the pointer given to mem_free() is passed through `p`. In the case of mem_resize(), the requested size is also available in `size`.

- the information to locate an invalid memory operation. The file name, function name and line number where a problem occurred are provided through `ifile`, `ifunc` and `iline`, respectively.
- the information about the memory block for which an invalid memory operation is invoked. For example, the "free-free" case where trying to deallocate already deallocated storage means that the pointer value delivered to mem_free() was allocated before. `afile`, `afunc`, `aline` and `asize` provide where it was allocated and what its size was. This information is useful in tracking how such an invalid operation is invoked.

If any of them is not available, they are set to a null pointer (for `ifile`, `ifunc`, `afile` and `afunc`) or 0 (for `size`, `iline`, `aline` and `asize`).

Warning:

Logging invalid memory operations is activated by mem_log() which is available only when the debugging version (not the production version) is used.

5.1.2 Field Documentation

5.1.2.1 const char∗ mem_loginfo_t::afile

file name in which storage in problem originally allocated

5.1.2.2 const char∗ mem_loginfo_t::afunc

function name in which storage in problem originally allocated

5.1.2.3 int mem_loginfo_t::aline

line number on which storage in problem originally allocated

5.1.2.4 size_t mem_loginfo_t::asize

size of storage in problem when allocated before

5.1.2.5 const char∗ mem_loginfo_t::ifile

file name in which invalid memory operation invoked

5.1.2.6 const char∗ mem_loginfo_t::ifunc

function name in which invalid memory operation invoked

5.1.2.7 int mem_loginfo_t::iline

line number on which invalid memory operation invoked

5.1.2.8 const void* mem_loginfo_t::p

pointer value used in invalid memory operation

5.1.2.9 size_t mem_loginfo_t::size

requested size; meaningful only when triggered by mem_resize()

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

- memory.h

Chapter 6

File Documentation

6.1 memory.c File Reference

Source for Memory Management Library - Production Version (CBL).

```
#include <stddef.h>
#include <stdlib.h>
#include "cbl/assert.h"
#include "cbl/except.h"
#include "memory.h"
```

Include dependency graph for memory.c:

Functions

- void ∗() mem_alloc (size_t n, const char ∗file, int line)
  allocates storage of the size `n` *in bytes.*
- void ∗() mem_calloc (size_t c, size_t n, const char ∗file, int line)
  allocates zero-filled storage of the size `c ∗ n` *in bytes.*
- void() mem_free (void ∗p, const char ∗file, int line)
  deallocates storage pointed to by `p`.

- void ∗() mem_resize (void ∗p, size_t n, const char ∗file, int line)

  adjust the size of storage pointed to by `p` *to* `n`*.*

- void() **mem_log** (FILE ∗fp, void freefunc(FILE ∗, const mem_loginfo_t ∗), void resizefunc(FILE ∗, const mem_loginfo_t ∗))
- void() **mem_leak** (void apply(const mem_loginfo_t ∗, void ∗), void ∗cl)

Variables

- const except_t mem_exceptfail = { "Allocation failed" }

  exception for memory allocation failure.

6.1.1 Detailed Description

Source for Memory Management Library - Production Version (CBL).

6.1.2 Function Documentation

6.1.2.1 void∗() mem_alloc (size_t *n*, const char ∗ *file*, int *line*)

allocates storage of the size `n` in bytes.

mem_alloc() does the same job as malloc() except:

- mem_alloc() raises an exception when fails the requested allocation;
- mem_alloc() does not take 0 as the byte length to preclude the possibility of returning a null pointer;
- mem_alloc() never returns a null pointer.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***n*** size in bytes for storage to be allocated

← ***file*** file name in which storage requested

← ***func*** function name in which strage requested (if C99 supported)

← ***line*** line number on which storage requested

Returns:

pointer to allocated storage

Warning:

> mam_alloc() returns no null pointer in any case. Allocation failure triggers an exception, so no need to handle an exceptional case with the return value.

Here is the caller graph for this function:

6.1.2.2 void∗() mem_calloc (size_t *c*, size_t *n*, const char ∗ *file*, int *line*)

allocates zero-filled storage of the size `c * n` in bytes.

mem_calloc() does the same job as mem_alloc() except that the storage it allocates are zero- filled. The similar explanation as for mem_alloc() applies to mem_calloc() too; see mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

- ← ***c*** number of items to be allocated
- ← ***n*** size in bytes for one item
- ← ***file*** file name in which storage requested
- ← ***func*** function name in which strage requested (if C99 supported)
- ← ***line*** line number on which storage requested

Returns:

> pointer to allocated (zero-filled) storage

6.1.2.3 void() mem_free (void ∗ *p*, const char ∗ *file*, int *line*)

deallocates storage pointed to by `p`.

mem_free() is a simple wrapper function for free().

The additional parameters, `file`, `func` (if C99 supported), `line` are for the consistent form in the calling sites; the debugging version of this library takes advantage of them to raise an exception when something goes wrong in mem_free(). When using the debugging version, some of the following unchecked errors are to be detected.

Possible exceptions: none

Unchecked errors: foreign value given for `p`

Warning:

A "foreign" value also includes a pointer value which points to storage already moved to a different address by, say, mem_resize().

Parameters:

- ← ***p*** pointer to storage to release
- ← ***file*** file name in which deallocation requested
- ← ***func*** function name in which deallocation requested (if C99 supported)
- ← ***line*** line number on which deallocation requested

Returns:

nothing

Here is the caller graph for this function:

6.1.2.4 void∗() mem_resize (void ∗ *p*, size_t *n*, const char ∗ *file*, int *line*)

adjust the size of storage pointed to by `p` to `n`.

mem_resize() does the main job of realloc(); adjusting the size of storage already allocated by mem_alloc() or mem_calloc(). While realloc() deallocates like free() when the given size is 0 and allocates like malloc() when the given pointer is a null pointer, mem_resize() accepts neither a null pointer nor zero as its arguments. The similar explanation as for mem_alloc() also applies to mem_resize(). See mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign value given for `p`

Parameters:

- ← ***p*** pointer to storage whose size to be adjusted
- ← ***n*** new size for storage
- ← ***file*** file name in which adjustment requested
- ← ***func*** function name in which adjustment requested (if C99 supported)
- ← ***line*** line number on which adjustment requested

Returns:

pointer to size-adjusted storage

Here is the caller graph for this function:

6.2 memory.h File Reference

Documentation for Memory Management Library (CBL).

`#include <stddef.h>`

`#include <stdio.h>`

`#include "cbl/except.h"`

Include dependency graph for memory.h:

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

Data Structures

- struct mem_loginfo_t

  contains the information about invalid memory operations.

Defines

- #define MEM_ALLOC(n) (mem_alloc((n), __FILE__, __LINE__))

  allocates storage of the size `n` *in bytes.*

- #define MEM_CALLOC(c, n) (mem_calloc((c), (n), __FILE__, __LINE__))

  allocation zero-filled storage of the size `c * n` *in bytes.*

- #define MEM_NEW(p) ((p) = MEM_ALLOC(sizeof ∗(p)))

  allocates to `p` *storage whose size is determined by the size of the pointed-to type by* `p`.

- #define MEM_NEW0(p) ((p) = MEM_CALLOC(1, sizeof ∗(p)))

  allocates to `p` *zero-filled storage whose size is determined by the size of the pointed-to type by* `p`.

- #define MEM_FREE(p) ((void)(mem_free((p), __FILE__, __LINE__), (p)=0))

  deallocates storage pointed to by `p` *and set it to a null pointer.*

- #define MEM_RESIZE(p, n) ((p) = mem_resize((p), (n), __FILE__, __LINE__))

  adjusts the size of storage pointed to by `p` *to* `n` *bytes.*

Functions

memory allocating functions:

- void ∗ mem_alloc (size_t, const char ∗, int)

  allocates storage of the size `n` *in bytes.*

- void ∗ mem_calloc (size_t, size_t, const char ∗, int)

  allocates zero-filled storage of the size `c` ∗ `n` *in bytes.*

memory deallocating functions:

- void mem_free (void ∗, const char ∗, int)

  deallocates storage pointed to by `p`.

memory resizing functions:

- void ∗ mem_resize (void ∗, size_t, const char ∗, int)

  adjust the size of storage pointed to by `p` *to* `n`.

memory debugging functions:

- void **mem_log** (FILE ∗, void(FILE ∗, const mem_loginfo_t ∗), void(FILE ∗, const mem_loginfo_t ∗))
- void **mem_leak** (void(const mem_loginfo_t ∗, void ∗), void ∗)

Variables

- const except_t mem_exceptfail

  exception for memory allocation failure.

6.2.1 Detailed Description

Documentation for Memory Management Library (CBL).

Header for Memory Management Library (CBL).

6.2.2 Define Documentation

6.2.2.1 #define MEM_FREE(p) ((void)(mem_free((p), __FILE__, __LINE__), (p)=0))

deallocates storage pointed to by `p` and set it to a null pointer.

See mem_free() for details.

Warning:

> `p` must be a modifiable lvalue; a rvalue expression or non-modifiable lvalue like one qualified by `const` is not allowed. Also, MEM_FREE() evaluates its argument twice, so an argument containing side effects results in an unpredictable result.

Possible exceptions: none

Unchecked errors: foreign value given for `p`

6.2.2.2 #define MEM_NEW(p) ((p) = MEM_ALLOC(sizeof *(p)))

allocates to `p` storage whose size is determined by the size of the pointed-to type by `p`.

A common way to allocate storage to a pointer `p` is as follows:

```
type *p;
p = malloc(sizeof(type));
```

However, this is error-prone; it might cause the memory corrupted if one forget to change every instance of `type` when the type of `p` changes to, say, `another_type`. To preclude problems like this a proposed way to allocate storage for a pointer `p` is:

```
p = malloc(sizeof(*p));
```

In this code, changing the type of `p` is automatically reflected to the allocation code above. Note that the expression given in the `sizeof` expression is not evaluated, so the validity of `p's` value does not matter here.

The macro MEM_NEW() is provided to facilitate such usage. It takes a pointer as an argument and allocates to it storage whose size is the size of the referrenced type. Therefore it makes an invalid call to invoke MEM_NEW() with a pointer to an imconplete type like a pointer to `void` and a pointer to a structure whose type definition is not visible.

Note that the `sizeof` operator does not evaluate its operand, which makes MEM_NEW() evaluate its argument exactly once as an actual function does. Embedding a side effect in the argument, however, is discouraged.

Possible exceptions: mem_exceptfail

Unchecked errors: none

6.2.2.3 #define MEM_NEW0(p) ((p) = MEM_CALLOC(1, sizeof *(p)))

allocates to `p` zero-filled storage whose size is determined by the size of the pointed-to type by `p`.

The same explanation for MEM_NEW() applies. See MEM_NEW() for details.

Possible exceptions: mem_exceptfail

Unchecked errors: none

6.2.2.4 #define MEM_RESIZE(p, n) ((p) = mem_resize((p), (n), __FILE__, __LINE__))

adjusts the size of storage pointed to by `p` to `n` bytes.

See mem_resize() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign value given for `p`

Warning:

> MEM_RESIZE() evaluates its argument twice. An argument containing side effects results in an unpredictable result.

6.2.3 Function Documentation

6.2.3.1 void* mem_alloc (size_t *n*, const char * *file*, int *line*)

allocates storage of the size `n` in bytes.

mem_alloc() does the same job as malloc() except:

- mem_alloc() raises an exception when fails the requested allocation;
- mem_alloc() does not take 0 as the byte length to preclude the possibility of returning a null pointer;
- mem_alloc() never returns a null pointer.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***n*** size in bytes for storage to be allocated

← ***file*** file name in which storage requested

← ***func*** function name in which strage requested (if C99 supported)

← ***line*** line number on which storage requested

Returns:

pointer to allocated storage

Warning:

mam_alloc() returns no null pointer in any case. Allocation failure triggers an exception, so no need to handle an exceptional case with the return value.

allocates storage of the size `n` in bytes.

Some general explanation on mem_alloc() can be found on the production version of the library.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***n*** size of memory block requested
← ***file*** file name in which allocation requested
← ***func*** function name in which allocation requested (if C99 supported)
← ***line*** linu number on which allocation requested

Returns:

memory block requested

Here is the caller graph for this function:

6.2.3.2 void∗ mem_calloc (size_t *c*, size_t *n*, const char ∗ *file*, int *line*)

allocates zero-filled storage of the size `c` ∗ `n` in bytes.

mem_calloc() does the same job as mem_alloc() except that the storage it allocates are zero- filled. The similar explanation as for mem_alloc() applies to mem_calloc() too; see mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***c*** number of items to be allocated
← ***n*** size in bytes for one item

← ***file*** file name in which storage requested

← ***func*** function name in which strage requested (if C99 supported)

← ***line*** line number on which storage requested

Returns:

pointer to allocated (zero-filled) storage

allocates zero-filled storage of the size `c` ∗ `n` in bytes.

mem_calloc() returns a zero-filled memory block whose size is at least `n`. mem_calloc() allocates a memory block by invoking mem_malloc() and set its every byte to zero by memset(). The similar explanation as for mem_alloc() applies to mem_calloc() too; see mem_alloc().

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: none

Parameters:

← ***c*** number of items to be allocated

← ***n*** size in bytes for one item

← ***file*** file name in which allocation requested

← ***func*** function name in which allocation requested (if C99 supported)

← ***line*** line number on which allocation requested

Returns:

pointer to allocated (zero-filled) memory block

Todo

Improvements are possible and planned:

- the C standard requires calloc() return a null pointer if it can allocates no storage of the size `c` ∗ `n` in bytes, which allows no overflow in computing the multiplication. So overflow checking is necessary to mimic the behavior of calloc().

Here is the call graph for this function:

6.2.3.3 void mem_free (void ∗ *p*, const char ∗ *file*, int *line*)

deallocates storage pointed to by `p`.

mem_free() is a simple wrapper function for free().

The additional parameters, `file`, `func` (if C99 supported), `line` are for the consistent form in the calling sites; the debugging version of this library takes advantage of them to raise an exception when something goes wrong in mem_free(). When using the debugging version, some of the following unchecked errors are to be detected.

Possible exceptions: none

Unchecked errors: foreign value given for `p`

Warning:

A "foreign" value also includes a pointer value which points to storage already moved to a different address by, say, mem_resize().

Parameters:

← ***p*** pointer to storage to release

← ***file*** file name in which deallocation requested

← ***func*** function name in which deallocation requested (if C99 supported)

← ***line*** line number on which deallocation requested

Returns:

nothing

deallocates storage pointed to by `p`.

mem_free() releases a given memory block.

Possible exceptions: assert_exceptfail

Unchecked errors: none

Parameters:

← ***p*** pointer to memory block to release (to mark as "freed")

← ***file*** file name in which deallocation requested

← ***func*** function name in which deallocation requested (if C99 supported)

← ***line*** line number on which deallocation is requested

Returns:

nothing

Here is the call graph for this function:

Here is the caller graph for this function:

6.2.3.4 void∗ mem_resize (void ∗ *p*, size_t *n*, const char ∗ *file*, int *line*)

adjust the size of storage pointed to by `p` to `n`.

mem_resize() does the main job of realloc(); adjusting the size of storage already allocated by mem_alloc() or mem_calloc(). While realloc() deallocates like free() when the given size is 0 and allocates like malloc() when the given pointer is a null pointer, mem_resize() accepts neither a null pointer nor zero as its arguments. The similar explanation as for mem_alloc() also applies to mem_resize(). See mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: foreign value given for `p`

Parameters:

← ***p*** pointer to storage whose size to be adjusted

← ***n*** new size for storage

← ***file*** file name in which adjustment requested

← ***func*** function name in which adjustment requested (if C99 supported)

← ***line*** line number on which adjustment requested

Returns:

pointer to size-adjusted storage

adjust the size of storage pointed to by `p` to `n`.

mem_resize() does the main job of realloc(); adjusting the size of the memory block already allocated by mem_alloc() or mem_calloc(). While realloc() deallocates like free() when the given size is 0 and allocates like malloc() when the given pointer is a null pointer, mem_resize() accepts neither a null pointer nor zero as its arguments. The similar explanation as for mem_alloc() also applies to mem_resize(). See mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***p*** pointer to memory block whose size to be adjusted

← ***n*** new size for memory block

← ***file*** file name in which adjustment requested

← ***func*** function name in which adjustment requested (if C99 supported)

← ***line*** line number on which adjustment requested

Returns:

pointer to size-adjusted memory block

Here is the call graph for this function:

Here is the caller graph for this function:

6.3 memoryd.c File Reference

Source for Memory Management Library - Debugging Version (CBL).

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "cbl/assert.h"
#include "cbl/except.h"
#include "memory.h"
```

Include dependency graph for memoryd.c:

Data Structures

- union **align**
- struct **descriptor**

Defines

- #define **NELEMENT**(array) (sizeof(array) / sizeof(∗(array)))
- #define **HASH**(p, t) (((uintptr_t)(p)>>3) % NELEMENT(t))
- #define **NDESCRIPTORS** 512
- #define **MULTIPLE**(x, y) ((((x)+(y)-1)/(y)) ∗ (y))
- #define **NALLOC** MULTIPLE(4096, sizeof(union align))
- #define **ALIGNED**(p) ((uintptr_t)(p) % sizeof(union align) == 0)
- #define **RAISE_EXCEPT_IF_INVALID**(p, n, type)

Typedefs

- typedef unsigned long **uintptr_t**

Functions

- void() mem_free (void ∗p, const char ∗file, int line)

  deallocates a memory block.

- void ∗() mem_resize (void ∗p, size_t n, const char ∗file, int line)

  adjusts the size of a memory block pointed to by `p` *to* `n`*.*

- void ∗() mem_calloc (size_t c, size_t n, const char ∗file, int line)

  allocates a zero-filled memory block of the size `c * n` *in bytes.*

- void ∗() mem_alloc (size_t n, const char ∗file, int line)

  allocates a new memory block of the size `n` *in bytes.*

- void() mem_log (FILE ∗fp, void freefunc(FILE ∗, const mem_loginfo_t ∗), void resizefunc(FILE ∗, const mem_loginfo_t ∗))

  starts to log invalid memory usage.

- void() mem_leak (void apply(const mem_loginfo_t ∗, void ∗), void ∗cl)

  calls a user-provided function for each memory block in use.

Variables

- const except_t mem_exceptfail = { "Allocation failed" }

  exception for memory allocation failure.

- void(∗ **logfuncFreefree**)(FILE ∗, const mem_loginfo_t ∗)
- void(∗ **logfuncResizefree**)(FILE ∗, const mem_loginfo_t ∗)

6.3.1 Detailed Description

Source for Memory Management Library - Debugging Version (CBL).

6.3.2 Define Documentation

6.3.2.1 #define RAISE_EXCEPT_IF_INVALID(p, n, type)

Value:

```
do {                                                                   \
                if (!ALIGNED(p) || (bp=descfind(p)) == NULL || bp->free) {         \
                    if (logfile)                                                   \
                        logprint((p), (n), bp, file, line, (int (*)())(type));     \
                    else                                                           \
                        except_raise(&assert_exceptfail, file, line);              \
                }                                                                  \
            } while(0)
```

6.3.3 Function Documentation

6.3.3.1 void∗() mem_alloc (size_t *n*, const char ∗ *file*, int *line*)

allocates a new memory block of the size `n` in bytes.

allocates storage of the size `n` in bytes.

Some general explanation on mem_alloc() can be found on the production version of the library.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***n*** size of memory block requested

← ***file*** file name in which allocation requested

← ***func*** function name in which allocation requested (if C99 supported)

← ***line*** linu number on which allocation requested

Returns:

memory block requested

6.3.3.2 void∗() mem_calloc (size_t *c*, size_t *n*, const char ∗ *file*, int *line*)

allocates a zero-filled memory block of the size `c * n` in bytes.

allocates zero-filled storage of the size `c * n` in bytes.

mem_calloc() returns a zero-filled memory block whose size is at least `n`. mem_calloc() allocates a memory block by invoking mem_malloc() and set its every byte to zero by memset(). The similar explanation as for mem_alloc() applies to mem_calloc() too; see mem_alloc().

Possible exceptions: assert_exceptfail, mem_exceptfail

Unchecked errors: none

Parameters:

← ***c*** number of items to be allocated

← ***n*** size in bytes for one item

← ***file*** file name in which allocation requested

← ***func*** function name in which allocation requested (if C99 supported)

← ***line*** line number on which allocation requested

Returns:

pointer to allocated (zero-filled) memory block

Todo

Improvements are possible and planned:

- the C standard requires calloc() return a null pointer if it can allocates no storage of the size `c * n` in bytes, which allows no overflow in computing the multiplication. So overflow checking is necessary to mimic the behavior of calloc().

Here is the call graph for this function:

6.3.3.3 void() mem_free (void $*$ *p*, const char $*$ *file*, int *line*)

deallocates a memory block.

deallocates storage pointed to by `p`.

mem_free() releases a given memory block.

Possible exceptions: assert_exceptfail

Unchecked errors: none

Parameters:

← ***p*** pointer to memory block to release (to mark as "freed")

← ***file*** file name in which deallocation requested

← ***func*** function name in which deallocation requested (if C99 supported)

← ***line*** line number on which deallocation is requested

Returns:

nothing

Here is the call graph for this function:

6.3.3.4 void() mem_leak (void *apply*const mem_loginfo_t $*$, void $*$, void $*$ *cl*)

calls a user-provided function for each memory block in use.

mem_leak() is useful when detecting memory leakage. Before terminating a program, calling it with a callback function which are passed to `apply` makes the callback function called with the information of every memory block still in use (or not deallocated).

Among the member of `mem_loginfo_t`, `p`, `size`, `afile`, `afunc` and `aline` are filled; if some of them are unavailable, they are set to a null pointer for pointer members or 0 for integer members. An information that a user needs to give to a callback function can be passed through `cl`. The following shows an example of a callback function:

```
void inuse(const mem_loginfo_t *loginfo, void *cl)
{
    FILE *logfile = cl;
    const char *file, func;

    file = (loginfo->afile)? loginfo->afile: "unknown file";
    func = (loginfo->afunc)? loginfo->afunc: "unknown function";

    fprintf(logfile, "** memory in use at %p\n", loginfo->p);
    fprintf(logfile, "this block is %ld bytes long and was allocated from %s() %s:%d\n",
            (unsigned long)loginfo->size, func, file, loginfo->aline);

    fflush(logfile);
}
```

If this callback function is invoked by calling mem_leak() as follows:

```
mem_leak(inuse, stderr);
```

it prints out a list of memory blocks still in use to `stderr` as follows:

```
** memory in use at 0xfff7
this block is 2048 bytes long and was allocated from table_init() table.c:235
```

If a null pointer is given to `apply`, the pre-defined internal callback function is invoked to print the information for memory leak to a file given through `cl` (after converted to a pointer to `FILE`). If `cl` is also a null pointer, a file possibly set by mem_log() is inspected to see if it is available, before the information printed finally goes to `stderr`.

Possible exceptions: none

Unchecked errors: invalid function pointer given for `apply`, invalid file pointer given for `cl` when `apply` is given a null pointer

Parameters:

← ***apply*** user-provided function to be called for each memory block in use

← ***cl*** passing-by argument to `apply`

Returns:

nothing

6.3.3.5 void() mem_log (FILE ∗ *fp*, void *freefunc*FILE ∗, const mem_loginfo_t ∗, void *resizefunc*FILE ∗, const mem_loginfo_t ∗)

starts to log invalid memory usage.

mem_log() starts to log invalid memory usage; deallocating an already released memory called "free-free" or "double free" and resizing a non-allocated memory called "resize-free" here. mem_log() provides two ways to log them. A user can register his/her own log function for the free-free or resize-free case by providing a function to `freefunc` or `resizefunc`. The necessary information is provided to the registered function via a `mem_loginfo_t` object. A user-provided log function must be defined as follows:

```
void user_freefree(FILE *fp, const mem_loginfo_t *info)
{
    ...
}
```

See the explanation of `mem_loginfo_t` for the information provided to a user-registered function. The file pointer given to mem_log()'s `fp` is passed to the first parameter of an user-registered log function.

If `freefunc` or `resizefunc` are given a null pointer, the default log messages are printed to the file specified by `fp`. The message looks like:

```
** freeing free memory
mem_free(0x6418) called from table_mgr() table.c:461
this block is 48 bytes long and was allocated table_init() table.c:233
** resizing unallocated memory
mem_resize(0xf7ff, 640) called from table_mgr() table.c:468
this block is 32 bytes long and was allocated table_init() table.c:230
```

Invoking mem_log() with a null pointer for `fp` stops logging.

Possible exceptions: none

Unchecked errors: invalid file pointer given for `fp`, invalid function pointer given for `freefunc` or `resizefunc`

Parameters:

← ***fp*** file to which log message printed out

← ***freefunc*** user-provided function to log free-free case; default message used when null pointer given

← ***resizefunc*** user-provided function to log resize-free case; default message used when null pointer given

Returns:

nothing

6.3.3.6 void∗() mem_resize (void ∗ *p*, size_t *n*, const char ∗ *file*, int *line*)

adjusts the size of a memory block pointed to by `p` to `n`.

adjust the size of storage pointed to by `p` to `n`.

mem_resize() does the main job of realloc(); adjusting the size of the memory block already allocated by mem_alloc() or mem_calloc(). While realloc() deallocates like free() when the given size is 0 and allocates like malloc() when the given pointer is a null pointer, mem_resize() accepts neither a null pointer nor zero as its arguments. The similar explanation as for mem_alloc() also applies to mem_resize(). See mem_alloc() for details.

Possible exceptions: mem_exceptfail, assert_exceptfail

Unchecked errors: none

Parameters:

← ***p*** pointer to memory block whose size to be adjusted

← ***n*** new size for memory block

← ***file*** file name in which adjustment requested

← ***func*** function name in which adjustment requested (if C99 supported)

← ***line*** line number on which adjustment requested

Returns:

pointer to size-adjusted memory block

Here is the call graph for this function:

Index