c-stdaux

C-Util Community

Mar 03, 2023

LIBRARY DOCUMENTATION

The **c-stdaux** project contains support-macros and auxiliary functions around the functionality of common C standard libraries. This includes helpers for the ISO-C Standard Library, but also other common specifications like POSIX or common extended features of wide-spread compilers like gcc and clang.

CHAPTER

ONE

API

The `c-stdaux.h` header contains a collection of auxiliary macros and helper functions around the functionality provided by the different C standard library implementations, as well as other specifications implemented by them.

Most of the helpers provided here provide aliases for common library and compiler features. Furthermore, several helpers simply provide other calling conventions than their standard counterparts (e.g., they allow for NULL to be passed with an object length of 0 where it makes sense to accept empty input).

The namespace used by this project is:

- `c_*` for all common C symbols or definitions that behave like proper C entities (e.g., macros that protect against double-evaluation would use lower-case names).
- `C_*` for all constants, as well as macros that may not be safe against double evaluation.
- `c_internal_*` and `C_INTERNAL_*` for all internal symbols that should not be invoked by the caller and are not part of the API guarantees.

1.1 Target Properties

Since multiple target compilers and systems are supported, c-stdaux exports a set of symbols that identify the target of the current compilation. The following pre-processor constants are defined (and evaluate to 1) if the current compilation targets the specific system. Note that multiple constants might be defined at the same time if compatibility to multiple targets is available.

- `C_COMPILER_CLANG`: The compiling software is compatible to the CLang LLVM Compiler.
- `C_COMPILER_DOCS`: The compilation is part of generating documentation.
- `C_COMPILER_GNUC`: The compiling software is compatible to the GNU C Compiler.
- `C_COMPILER_MSVC`: The compiling software is compatible to Microsoft Visual Studio (use `_MSC_VER` to check for specific version support).
- `C_OS_LINUX`: The target system is compatible to Linux.
- `C_OS_MACOS`: The target system is compatible to Apple MacOS.
- `C_OS_WINDOWS`: The target system is compatible to Microsoft Windows.
- `C_MODULE_GENERIC`: The **-generic.h* module was included.
- `C_MODULE_GNUC`: The **-gnuc.h* module was included.
- `C_MODULE_UNIX`: The **-unix.h* module was included.

Note that other exported symbols might depend on one of these constants to be set in order to be exposed. See the documentation of each symbol for details. Furthermore, if stub implementations do not violate the guarantees of a symbol, they will be provided for targets that do not provide the necessary infrastructure (e.g., `_c_likely_()` is a no-op on MSVC).

1.2 Guaranteed STD-C Includes

c-stdaux includes a set of C Standard Library headers. All those includes are guaranteed and part of the API. See the actual header for a comprehensive list.

1.3 Generic Compiler Intrinsics

This section provides access to compiler extensions and intrinsics which are either portable or have generic fallbacks.

`_c_always_inline_`

Always-inline attribute

Annotate a symbol to be inlined more aggressively. On GNUC targets this is an alias for `__attribute__((__always_inline__))`. On MSVC targets this is and alias for `__forceinline`. On other systems, this is a no-op.

`_c_boolean_expr_`(_x)

Evaluate a boolean expression

Parameters

- **`_x`** – Expression to evaluate

Evaluate the given expression and convert the result to 1 or 0. In most cases this is equivalent to `(!!(_x))`. However, for given compilers this avoids the parentheses to improve diagnostics with `-Wparentheses`.

Outside of macros, this has no added value.

Returns
Evaluates to the value of `!!_x`.

`_c_likely_`(_x)

Likely attribute

Parameters

- **`_x`** – Expression to evaluate

Alias for `__builtin_expect(!!(_x), 1)`.

Returns
The expression `!!_x` is evaluated and returned.

`_c_public_`

Public attribute

Mark a symbol definition as public, to be exported by the linker. On GNUC-compatible systems, this is an alias for `__attribute__((__visibility__("default")))`. On all other systems, this is a no-op.

Note that this explicitly does not resolve to `__declspec(dllexport)` on MSVC targets, since that would require knowing whether to compile for export or inport and whether to compile for static or dynamic linking. Instead, the `_c_public_` attribute is meant to be used unconditionally on definition only. For MSVC exports, we recommend module definition files.

_c_unlikely_(_x)

Unlikely attribute

Parameters

- **_x** – Expression to evaluate

Alias for `__builtin_expect(!!(_x), 0)`.

Returns

The expression `!!_x` is evaluated and returned.

1.4 Generic Utility Macros

A set of utility macros which is portable to all supported platforms or has generic fallback variants.

C_STRINGIFY(_x)

Stringify a token, but evaluate it first

Parameters

- **_x** – Token to evaluate and stringify

Returns

Evaluates to a constant string literal

C_CONCATENATE(_x, _y)

Concatenate two tokens, but evaluate them first

Parameters

- **_x** – First token
- **_y** – Second token

Returns

Evaluates to a constant identifier

C_EXPAND(_x)

Expand a tuple to a series of its values

Parameters

- **_x** – Tuple to expand

Returns

Evaluates to the expanded tuple

C_VAR(...)

Generate unique variable name

Parameters

- **_x** – Name of variable, optional
- **_uniq** – Unique prefix, usually provided by `__COUNTER__`, optional

This macro shall be used to generate unique variable names, that will not be shadowed by recursive macro invocations. It is effectively a *C_CONCATENATE* of both arguments, but also provides a globally separated prefix and makes the code better readable.

The second argument is optional. If not given, `__LINE__` is implied, and as such the macro will generate the same identifier if used multiple times on the same code-line (or within a macro). This should be used if recursive

calls into the macro are not expected. In fact, no argument is necessary in this case, as a mere `C_VAR` will evaluate to a valid variable name.

This helper may be used by macro implementations that might reasonable well be called in a stacked fasion, like:

```
c_max(foo, c_max(bar, baz))
```

Such a stacked call of *c_max()* might cause compiler warnings of shadowed variables in the definition of *c_max()*. By using `C_VAR()`, such warnings can be silenced as each evaluation of *c_max()* uses unique variable names.

Returns
This evaluates to a constant identifier.

1.5 Generic Standard Library Utilities

The C Standard Library lacks some crucial and basic support functions. This section describes the set of helpers provided as extension to the standard library.

c_assume_aligned(_ptr, _alignment, _offset)

Hint alignment to compiler

Parameters

- **_ptr** – Pointer to provide alignment hint for
- **_alignment** – Alignment in bytes
- **_offset** – Misalignment offset

This hints to the compiler that *_ptr* - *_offset* is aligned to the alignment specified in *_alignment*.

On platforms without support for *__builtin_assume_aligned()* this is a no-op.

Returns
_ptr is returned.

c_assert(_x)

Runtime assertions

Parameters

- **_x** – Result of an expression

This function behaves like the standard `assert(3)` macro. That is, if `NDEBUG` is defined, it is a no-op. In all other cases it will assert that the result of the passed expression is true.

Unlike the standard `assert(3)` macro, this function always evaluates its argument. This means side-effects will always be evaluated! However, if the macro is used with constant expressions, the compiler will be able to optimize it away.

int **c_errno**(void)

Return valid errno

This helper should be used to silence warnings if you know `errno` is valid (ie., `errno` is greater than 0). Instead of `return -errno;`, use `return -c_errno();` It will suppress bogus warnings in case the compiler assumes `errno` might be 0 (or smaller than 0) and thus the caller's error-handling might not be triggered.

This helper should be avoided whenever possible. However, occasionally we really want to silence warnings (especially with static/inline functions). In those cases, the compiler usually cannot deduce that some error paths are guaranteed to be taken. Hence, making the return value explicit allows it to better optimize the code.

Note that you really should never use this helper to work around broken libc calls or syscalls, not setting 'errno' correctly.

Returns
Positive error code is returned.

void *`c_memset`(void *p, int c, size_t n)

Fill memory region with constant byte

Parameters

- `p` – Pointer to memory region, if non-empty
- `c` – Value to fill with
- `n` – Size of the memory region in bytes

This function works like `memset(3)` if `n` is non-zero. If `n` is zero, this function is a no-op. Therefore, unlike `memset(3)` it is safe to call this function with `NULL` as `p` if `n` is 0.

Returns
`p` is returned.

void *`c_memzero`(void *p, size_t n)

Clear memory area

Parameters

- `p` – Pointer to memory region, if non-empty
- `n` – Size of the memory region in bytes

Clear a memory area to 0. If the memory area is empty, this is a no-op. Similar to `c_memset()`, this function allows `p` to be `NULL` if the area is empty.

Returns
`p` is returned.

void *`c_memcpy`(void *dst, const void *src, size_t n)

Copy memory area

Parameters

- `dst` – Pointer to target area
- `src` – Pointer to source area
- `n` – Length of area to copy

Copy the memory of size `n` from `src` to `dst`, just as `memcpy(3)` does, except this function allows either to be `NULL` if `n` is zero. In the latter case, the operation is a no-op.

Returns
`p` is returned.

int `c_memcmp`(const void *s1, const void *s2, size_t n)

Compare memory areas

Parameters

- `s1` – Pointer to one area
- `s2` – Pointer to other area
- `n` – Length of area to compare

Compare the memory of size `n` of `s1` and `s2`, just as `memcmp(3)` does, except this function allows either to be `NULL` if `n` is zero.

Returns
Comparison result for ordering is returned.

1.6 Memory Access

This section provides helpers to read and write arbitrary memory locations. They are carefully designed to follow all language restrictions and thus work with strict-aliasing and alignment rules.

The C language does not allow aliasing an object with a pointer of an incompatible type (with few exceptions). Furthermore, memory access must be aligned. This function uses exceptions in the language to circumvent both restrictions.

Note that pointer-offset calculations should avoid exceeding the extents of the object, even if the object is surrounded by other objects. That is, *ptr+offset* should point to the same object as *ptr*. Otherwise, pointer provenance will have to be considered.

uint8_t **c_load_8**(const void *memory, size_t offset)

Read a u8 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unsigned 8-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint16_t **c_load_16be_unaligned**(const void *memory, size_t offset)

Read an unaligned big-endian u16 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned big-endian unsigned 16-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint16_t **c_load_16be_aligned**(const void *memory, size_t offset)

Read an aligned big-endian u16 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned big-endian unsigned 16-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint16_t **c_load_16le_unaligned**(const void *memory, size_t offset)

Read an unaligned little-endian u16 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned little-endian unsigned 16-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint16_t **c_load_16le_aligned**(const void *memory, size_t offset)

Read an aligned little-endian u16 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned little-endian unsigned 16-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint32_t **c_load_32be_unaligned**(const void *memory, size_t offset)

Read an unaligned big-endian u32 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned big-endian unsigned 32-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint32_t **c_load_32be_aligned**(const void *memory, size_t offset)

Read an aligned big-endian u32 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned big-endian unsigned 32-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint32_t **c_load_32le_unaligned**(const void *memory, size_t offset)

Read an unaligned little-endian u32 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned little-endian unsigned 32-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint32_t **c_load_32le_aligned**(const void *memory, size_t offset)

Read an aligned little-endian u32 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned little-endian unsigned 32-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint64_t **c_load_64be_unaligned**(const void *memory, size_t offset)

Read an unaligned big-endian u64 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned big-endian unsigned 64-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint64_t **c_load_64be_aligned**(const void *memory, size_t offset)

Read an aligned big-endian u64 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned big-endian unsigned 64-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint64_t **c_load_64le_unaligned**(const void *memory, size_t offset)

Read an unaligned little-endian u64 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an unaligned little-endian unsigned 64-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

uint64_t **c_load_64le_aligned**(const void *memory, size_t offset)

Read an aligned little-endian u64 from memory

Parameters

- **memory** – Memory location to operate on
- **offset** – Offset in bytes from the pointed memory location

This reads an aligned little-endian unsigned 64-bit integer at the offset of the specified memory location.

Returns
The read value is returned.

c_load(_type, _endian, _aligned, _memory, _offset)

Read from memory

Parameters

- **_type** – Datatype to read
- **_endian** – Endianness
- **_aligned** – Aligned or unaligned access
- **_memory** – Memory location to operate on
- **_offset** – Offset in bytes from the pointed memory location

This reads a value of the same size as *_type* at the offset of the specified memory location. *_endian* must be either *be* or *le*, *_aligned* must be either *aligned* or *unaligned*.

This is a generic macro that maps to the respective *c_load_*()* function.

Returns
The read value is returned.

1.7 Generic Destructors

A set of destructors is provided which extends standard library destructors to adhere to some adjuvant rules. In particular, they return an invalid value of the particular object, rather than void. This allows direct assignment to any member-field and/or variable they are defined in, like:

```
foo = c_free(foo);
foo->bar = c_fclose(foo->bar);
```

Furthermore, all those destructors can be safely called with the "INVALID" value as argument, and they will be a no-op.

void *__c_free__(void *p)

Destructor-wrapper for free()

Parameters

- **p** – Value to pass to destructor, or NULL

Wrapper around `free()`, but always returns `NULL`.

Returns
NULL is returned.

FILE *__c_fclose__(FILE *f)

Destructor-wrapper for fclose()

Parameters

- **f** – File handle to pass to destructor, or NULL

Wrapper around `fclose()`, but a no-op if `NULL` is passed. Always returns `NULL`.

Returns
NULL is returned.

1.8 Generic Cleanup Helpers

A set of helpers that aid in creating functions suitable for use with *_c_cleanup_()*. Furthermore, a collection of predefined cleanup functions of a set of standard library objects ready for use with *_c_cleanup_()*. Those cleanup helpers are always suffixed with a `p`.

The helpers that are provided are:

- `c_freep()`: Wrapper around *c_free()*.
- `c_fclosep()`: Wrapper around *c_fclose()*.

C_DEFINE_CLEANUP(_type, _func)

Define cleanup helper

Parameters

- **_type** – Type of object to cleanup
- **_func** – Destructor of the respective type

Define a C static inline function that takes a single argument of type *_type* and calls *_func* on it, if its dereferenced value of its argument evaluates to true. Otherwise, it is a no-op.

This macro allows for very simple and fast creation of cleanup helpers for use with `_c_cleanup_()`, based on any destructor and type you provide to it.

C_DEFINE_DIRECT_CLEANUP(_type, _func)

Define direct cleanup helper

Parameters

- **_type** – Type of object to cleanup
- **_func** – Destructor of the respective type

This works like *C_DEFINE_CLEANUP()* but does not check the dereferenced value of its argument. It always unconditionally invokes the destructor.

1.9 GNUC Compiler Attributes

The GCC compiler uses the `__attribute__((__xyz__()))` syntax to annotate language entities with special attributes. Aliases are provided by this header which map one-to-one to the respective compiler attributes.

These attributes are not supported by all compilers, but are always provided by this header. They are pre-processor macros and do not affect the compilation, unless used. Note that most compilers support these, not just GCC.

_c_cleanup_(_x)

Cleanup attribute

Parameters

- **_x** – Cleanup function to use

Alias for `__attribute__((__cleanup__(_x)))`.

_c_const_

Const attribute

Alias for `__attribute__((__const__))`.

_c_deprecated_

Deprecated attribute

Alias for `__attribute__((__deprecated__))`.

_c_hidden_

Hidden attribute

Alias for `__attribute__((__visibility__("hidden")))`.

_c_packed_

Packed attribute

Alias for `__attribute__((__packed__))`.

_c_printf_(_a, _b)

Printf attribute

Parameters

- **_a** – Format expression argument index
- **_b** – First format-parameter argument index

Alias for `__attribute__((__format__(printf, _a, _b)))`.

_c_pure_

Pure attribute

Alias for `__attribute__((__pure__))`.

_c_sentinel_

Sentinel attribute

Alias for `__attribute__((__sentinel__))`.

_c_unused_

Unused attribute

Alias for `__attribute__((__unused__))`.

1.10 GNUC-Specific Utility Macros

A set of utility macros is provided which aids in creating safe macros suitable for use in other pre-processor statements as well as in C expressions.

C_EXPR_ASSERT(_expr, _assertion, _message)

Create expression with assertion

Parameters

- **_expr** – Expression to evaluate to
- **_assertion** – Arbitrary assertion
- **_message** – Message associated with the assertion

This macro simply evaluates to `_expr`. That is, it can be used in any context that expects an expression like `_expr`. Additionally, it takes an assertion as `_assertion` and evaluates it through `_Static_assert()`, using `_message` as debug message.

The `_Static_assert()` builtin of C11 is defined as statement and thus cannot be used in expressions. This macro circumvents this restriction.

Returns
Evaluates to `_expr`.

`C_CC_MACRO1`(_call, _x1, ...)

Provide safe environment to a macro

Parameters

- **`_call`** – Macro to call
- **`_x1`** – First argument
- **`...`** – Further arguments to forward unmodified to `_call`

This function simplifies the implementation of macros. Whenever you implement a macro, provide the internal macro name as `_call` and its argument as `_x1`. Inside of your internal macro, you…

- are safe against multiple evaluation errors, since `C_CC_MACRO1` will store the initial parameters in temporary variables.
- support constant folding, as `C_CC_MACRO1` takes care to invoke your macro with the original values, if they are compile-time constant.
- have unique variable names for recursive callers and will not run into variable-shadowing-warnings accidentally.
- have properly typed arguments as `C_CC_MACRO1` stores the original arguments in an `__auto_type` temporary variable.

Returns
Result of `_call` is returned.

`C_CC_MACRO2`(_call, _x1, _x2, ...)

Provide safe environment to a macro

Parameters

- **`_call`** – Macro to call
- **`_x1`** – First argument
- **`_x2`** – Second argument
- **`...`** – Further arguments to forward unmodified to `_call`

This is the 2-argument equivalent of *C_CC_MACRO1()*.

Returns
Result of `_call` is returned.

`C_CC_MACRO3`(_call, _x1, _x2, _x3, ...)

Provide safe environment to a macro

Parameters

- **`_call`** – Macro to call
- **`_x1`** – First argument

- **_x2** – Second argument
- **_x3** – Third argument
- **...** – Further arguments to forward unmodified to `_call`

This is the 3-argument equivalent of *C_CC_MACRO1()*.

Returns
Result of `_call` is returned.

1.11 Standard Library Utilities

The C Standard Library lacks some crucial and basic support functions. This section describes the set of helpers provided as extension to the standard library.

C_ARRAY_SIZE(_x)

Calculate number of array elements at compile time

Parameters

- **_x** – Array to calculate size of

Returns
Evaluates to a constant integer expression.

C_DECIMAL_MAX(_arg)

Calculate maximum length of a decimal representation

Parameters

- **_type** – Integer variable/type

This calculates the bytes required for the decimal representation of an integer of the given type. It accounts for a possible +/- prefix, but it does *NOT* include the trailing terminating zero byte.

Returns
Evaluates to a constant integer expression

c_container_of(_ptr, _type, _member)

Cast a member of a structure out to the containing type

Parameters

- **_ptr** – Pointer to the member or NULL
- **_type** – Type of the container struct this is embedded in
- **_member** – Name of the member within the struct

This uses `offsetof(3)` to turn a pointer to a structure-member into a pointer to the surrounding structure.

Returns
Pointer to the surrounding object.

c_max(_a, _b)

Compute maximum of two values

Parameters

- **_a** – Value A
- **_b** – Value B

Calculate the maximum of both passed values. Both arguments are evaluated exactly once, under all circumstances. Furthermore, if both values are constant expressions, the result will be constant as well.

The comparison of their values is performed with the types given by the caller. It is the caller's responsibility to convert them to suitable types if necessary.

Returns
Maximum of both values is returned.

`c_min`(_a, _b)

Compute minimum of two values

Parameters

- **_a** – Value A
- **_b** – Value B

Calculate the minimum of both passed values. Both arguments are evaluated exactly once, under all circumstances. Furthermore, if both values are constant expressions, the result will be constant as well.

The comparison of their values is performed with the types given by the caller. It is the caller's responsibility to convert them to suitable types if necessary.

Returns
Minimum of both values is returned.

`c_less_by`(_a, _b)

Calculate clamped difference of two values

Parameters

- **_a** – Minuend
- **_b** – Subtrahend

Calculate `_a - _b`, but clamp the result to 0. Both arguments are evaluated exactly once, under all circumstances. Furthermore, if both values are constant expressions, the result will be constant as well.

The comparison of their values is performed with the types given by the caller. It is the caller's responsibility to convert them to suitable types if necessary.

Returns
This computes `_a - _b`, if `_a > _b`. Otherwise, 0 is returned.

`c_clamp`(_x, _low, _high)

Clamp value to lower and upper boundary

Parameters

- **_x** – Value to clamp
- **_low** – Lower boundary
- **_high** – Higher boundary

This clamps `_x` to the lower and higher bounds given as `_low` and `_high`. All arguments are evaluated exactly once, and yield a constant expression if all arguments are constant as well.

The comparison of their values is performed with the types given by the caller. It is the caller's responsibility to convert them to suitable types if necessary.

Returns
Clamped integer value.

`c_div_round_up`(_x, _y)

Calculate integer quotient but round up

Parameters

- **_x** – Dividend
- **_y** – Divisor

Calculates `x / y` but rounds up the result to the next integer. All arguments are evaluated exactly once, and yield a constant expression if all arguments are constant.

Note: `(x + y - 1) / y` suffers from an integer overflow, even though the computation should be possible in the given type. Therefore, we use `x / y + !!(x % y)`. Note that on most CPUs a division returns both the quotient and the remainder, so both should be equally fast. Furthermore, if the divisor is a power of two, the compiler will optimize it, anyway.

The operationsare performed with the types given by the caller. It is the caller's responsibility to convert the arguments to suitable types if necessary.

Returns
The quotient is returned.

`c_align_to`(_val, _to)

Align value to a multiple

Parameters

- **_val** – Value to align
- **_to** – Align to multiple of this

This aligns `_val` to a multiple of `_to`. If `_val` is already a multiple of `_to`, `_val` is returned unchanged. This function operates within the boundaries of the type of `_val` and `_to`. Make sure to cast them if needed.

The arguments of this macro are evaluated exactly once. If both arguments are a constant expression, this also yields a constant return value.

Note that `_to` must be a power of 2, otherwise the behavior will not match expectations.

Returns
`_val` aligned to a multiple of `_to`.

1.12 Guaranteed Unix Includes

c-stdaux-unix includes a set of Unix headers. All those includes are guaranteed and part of the API. See the actual header for a comprehensive list.

1.13 Common Unix Destructors

A set of destructors is provided which extends standard library destructors to adhere to some adjuvant rules. In particular, they return an invalid value of the particular object, rather than void. This allows direct assignment to any member-field and/or variable they are defined in, like:

```
foo->bar = c_close(foo->bar);
```

Furthermore, all those destructors can be safely called with the "INVALID" value as argument, and they will be a no-op.

int **c_close**(int fd)

Destructor-wrapper for close()

Parameters

- **fd** – File-descriptor to pass to destructor, or negative value

Wrapper around `close()`, but a no-op if a negative value is provided. Always returns `-1`.

Returns
-1 is returned.

DIR ***c_closedir**(DIR *d)

Destructor-wrapper for closedir()

Parameters

- **d** – Directory handle to pass to destructor, or NULL

Wrapper around `closedir()`, but a no-op if `NULL` is passed. Always returns `NULL`.

Returns
NULL is returned.

1.14 Common Cleanup Helpers

A set of helpers that aid in creating functions suitable for use with *_c_cleanup_()*. Furthermore, a collection of predefined cleanup functions of a set of standard library objects ready for use with *_c_cleanup_()*. Those cleanup helpers are always suffixed with a `p`.

The helpers that are provided are:

- `c_closep()`: Wrapper around *c_close()*.
- `c_closedirp()`: Wrapper around *c_closedir()*.

INDEX

Symbols

C