

# Package: Rtinycc (via r-universe)

June 3, 2026

**Title** Builds the 'TinyCC' Command-Line Interface and Library for 'C' Scripting in 'R'

**Version** 0.1.11.9000

**Description** Builds the 'TinyCC' (Tiny 'C' Compiler) command-line interface and library for package use in 'R'. The package compiles 'TinyCC' from source and provides R functions to interact with the compiler. 'TinyCC' can be used for header preprocessing, just-in-time compilation of 'C' code in 'R', and lightweight 'C' scripting workflows.

**License** GPL (>= 3)

**Copyright** See inst/LICENSE.note for bundled TinyCC copyright and licensing details.

**Depends** R (>= 4.4.0)

**Imports** lambda.r

**Suggests** bench, callme, knitr, rmarkdown, tinytest, treesitter.c (>= 0.0.4)

**VignetteBuilder** knitr

**SystemRequirements** GNU make

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.3

**URL** <https://github.com/soukoku-bioinfo/Rtinycc>,  
<https://soukoku-bioinfo.github.io/Rtinycc/>

**BugReports** <https://github.com/soukoku-bioinfo/Rtinycc/issues>

**Config/pak/sysreqs** make

**Repository** <https://soukoku-bioinfo.r-universe.dev>

**Date/Publication** 2026-06-03 20:20:38 UTC

**RemoteUrl** <https://github.com/soukoku-bioinfo/Rtinycc>

**RemoteRef** HEAD

**RemoteSha** b79e789a60d3b31c7a68d3f1884025ee8618206a

## Contents

\$.tcc_compiled . . . . .	4
as.character.tcc_cstring . . . . .	5
blas_lapack_info . . . . .	5
generate_trampoline . . . . .	6
get_external_ptr_addr . . . . .	6
is_callback_type . . . . .	7
parse_callback_type . . . . .	7
print.tcc_callback . . . . .	8
print.tcc_compiled . . . . .	8
print.tcc_cstring . . . . .	9
print.tcc_ffi . . . . .	9
tcc_add_file . . . . .	10
tcc_add_include_path . . . . .	10
tcc_add_library . . . . .	11
tcc_add_library_path . . . . .	11
tcc_add_symbol . . . . .	12
tcc_add_sysinclude_path . . . . .	12
tcc_bind . . . . .	13
tcc_call_symbol . . . . .	14
tcc_callback . . . . .	15
tcc_callback_async_drain . . . . .	16
tcc_callback_async_schedule . . . . .	16
tcc_callback_close . . . . .	17
tcc_callback_ptr . . . . .	17
tcc_callback_valid . . . . .	18
tcc_compile . . . . .	18
tcc_compile_string . . . . .	19
tcc_container_of . . . . .	19
tcc_cstring . . . . .	20
tcc_cstring_object . . . . .	20
tcc_data_ptr . . . . .	21
tcc_enum . . . . .	21
tcc_ffi . . . . .	22
tcc_field_addr . . . . .	22
tcc_free . . . . .	23
tcc_generate_bindings . . . . .	24
tcc_get_symbol . . . . .	25
tcc_global . . . . .	25
tcc_header . . . . .	26
tcc_include . . . . .	27
tcc_include_paths . . . . .	27
tcc_introspect . . . . .	28
tcc_library . . . . .	28
tcc_library_path . . . . .	29
tcc_link . . . . .	29
tcc_list_symbols . . . . .	31

tcc_malloc . . . . .	31
tcc_map_c_type_to_ffi . . . . .	32
tcc_null_ptr . . . . .	33
tcc_options . . . . .	33
tcc_output . . . . .	34
tcc_path . . . . .	34
tcc_prefix . . . . .	35
tcc_ptr_addr . . . . .	35
tcc_ptr_free_set_null . . . . .	36
tcc_ptr_is_null . . . . .	36
tcc_ptr_is_owned . . . . .	37
tcc_ptr_set . . . . .	37
tcc_ptr_utils . . . . .	38
tcc_read_bytes . . . . .	38
tcc_read_cstring . . . . .	39
tcc_read_f32 . . . . .	39
tcc_read_f64 . . . . .	40
tcc_read_i16 . . . . .	40
tcc_read_i32 . . . . .	41
tcc_read_i64 . . . . .	41
tcc_read_i8 . . . . .	42
tcc_read_ptr . . . . .	42
tcc_read_u16 . . . . .	43
tcc_read_u32 . . . . .	43
tcc_read_u64 . . . . .	44
tcc_read_u8 . . . . .	44
tcc_recompile . . . . .	45
tcc_relocate . . . . .	45
tcc_run_cli . . . . .	46
tcc_set_options . . . . .	46
tcc_source . . . . .	47
tcc_state . . . . .	47
tcc_struct . . . . .	48
tcc_struct_raw_access . . . . .	48
tcc_symbol_is_valid . . . . .	49
tcc_treesitter_bindings . . . . .	49
tcc_treesitter_defines . . . . .	50
tcc_treesitter_enum_bindings . . . . .	51
tcc_treesitter_enum_members . . . . .	52
tcc_treesitter_enums . . . . .	52
tcc_treesitter_functions . . . . .	53
tcc_treesitter_global_types . . . . .	54
tcc_treesitter_globals . . . . .	54
tcc_treesitter_struct_accessors . . . . .	55
tcc_treesitter_struct_bindings . . . . .	56
tcc_treesitter_struct_members . . . . .	56
tcc_treesitter_structs . . . . .	57
tcc_treesitter_union_accessors . . . . .	58

tcc_treesitter_union_bindings . . . . .	59
tcc_treesitter_union_members . . . . .	59
tcc_treesitter_unions . . . . .	60
tcc_union . . . . .	61
tcc_write_bytes . . . . .	61
tcc_write_f32 . . . . .	62
tcc_write_f64 . . . . .	62
tcc_write_i16 . . . . .	63
tcc_write_i32 . . . . .	63
tcc_write_i64 . . . . .	64
tcc_write_i8 . . . . .	64
tcc_write_ptr . . . . .	65
tcc_write_u16 . . . . .	65
tcc_write_u32 . . . . .	66
tcc_write_u64 . . . . .	66
tcc_write_u8 . . . . .	67
<b>Index</b>	<b>68</b>

---

\$.tcc_compiled	<i>Access a compiled FFI symbol</i>
-----------------	-------------------------------------

---

### Description

Overrides \$ to detect dead pointers after deserialization and recompile transparently from the stored recipe.

### Usage

```
## S3 method for class 'tcc_compiled'
x$name
```

### Arguments

x	A tcc_compiled object
name	Symbol name to access

### Value

The callable function or metadata field

---

as.character.tcc\_cstring

*Convert a tcc\_cstring object to an R string*


---

**Description**

Convert a tcc\_cstring object to an R string

**Usage**

```
## S3 method for class 'tcc_cstring'
as.character(x, ...)
```

**Arguments**

x	A tcc_cstring object.
...	Ignored.

**Value**

A character scalar containing the string value. Returns the cached R copy when available; otherwise reads the current NUL-terminated C string from x\$ptr.

---

blas\_lapack\_info

*Report active BLAS/LAPACK runtime information from R*


---

**Description**

Returns the BLAS/LAPACK runtime details as reported by R itself, plus convenience flags indicating whether Rblas and Rlapack appear available in loaded DLLs/shared objects.

**Usage**

```
blas_lapack_info()
```

**Value**

A named list with fields: blas\_path, lapack\_path, has\_rblas, has\_rlapack, loaded\_dlls.

---

generate\_trampoline    *Generate trampoline code for a callback argument*

---

**Description**

Generate trampoline code for a callback argument

**Usage**

```
generate_trampoline(trampoline_name, sig)
```

**Arguments**

trampoline_name	Name of the trampoline function
sig	Parsed signature

**Value**

C code string for the trampoline

---

get\_external\_ptr\_addr    *Get the address of an external pointer*

---

**Description**

Extract the memory address from an external pointer as a numeric value. This is primarily useful for debugging and inspection purposes.

**Usage**

```
get_external_ptr_addr(ptr)
```

**Arguments**

ptr	An external pointer object (e.g., from tcc_get_symbol()).
-----	---

**Value**

The memory address as a numeric value.

---

is\_callback\_type      *Check if a type represents a callback*

---

**Description**

Check if a type represents a callback

**Usage**

is\_callback\_type(type)

**Arguments**

type                  Type string to check

**Value**

Logical

---

parse\_callback\_type      *Parse callback type specification*

---

**Description**

Parse callback type specification

**Usage**

parse\_callback\_type(type)

**Arguments**

type                  Type string like "callback:double(int,int)"

**Value**

Parsed signature list or NULL

---

```
print.tcc_callback    Print tcc_callback object
```

---

**Description**

Print tcc\_callback object

**Usage**

```
## S3 method for class 'tcc_callback'
print(x, ...)
```

**Arguments**

x	A tcc_callback object
...	Ignored

**Value**

The input tcc\_callback object, invisibly. Called for its side effect of printing the callback signature, thread-safety flag, and validity status.

---

```
print.tcc_compiled    Print tcc_compiled object
```

---

**Description**

Print tcc\_compiled object

**Usage**

```
## S3 method for class 'tcc_compiled'
print(x, ...)
```

**Arguments**

x	A tcc_compiled object
...	Ignored

**Value**

The input tcc\_compiled object, invisibly. Called for its side effect of printing the compilation output mode and the status of compiled callable symbols.

---

print.tcc_cstring	<i>Print a tcc_cstring object</i>
-------------------	-----------------------------------

---

**Description**

Print a tcc\_cstring object

**Usage**

```
## S3 method for class 'tcc_cstring'  
print(x, ...)
```

**Arguments**

x	A tcc_cstring object.
...	Ignored.

**Value**

The input tcc\_cstring object, invisibly. Called for its side effect of printing the current string value.

---

print.tcc_ffi	<i>Print tcc_ffi object</i>
---------------	-----------------------------

---

**Description**

Print tcc\_ffi object

**Usage**

```
## S3 method for class 'tcc_ffi'  
print(x, ...)
```

**Arguments**

x	A tcc_ffi object
...	Ignored

**Value**

The input tcc\_ffi object, invisibly. Called for its side effect of printing the configured output mode, registered symbols, and selected libraries/include paths.

---

tcc_add_file	<i>Add a source file to a libtcc state</i>
--------------	--

---

**Description**

Add a source file to a libtcc state

**Usage**

```
tcc_add_file(state, path)
```

**Arguments**

state	A tcc_state.
path	Path to a C source file.

**Value**

Integer status code (0 = success).

---

tcc_add_include_path	<i>Add an include path to a libtcc state</i>
----------------------	--

---

**Description**

Add an include path to a libtcc state

**Usage**

```
tcc_add_include_path(state, path)
```

**Arguments**

state	A tcc_state.
path	Path to include directory.

**Value**

Integer status code (0 = success).

---

tcc\_add\_library      *Add a library to a libtcc state*

---

**Description**

Add a library to a libtcc state

**Usage**

```
tcc_add_library(state, library)
```

**Arguments**

state	A tcc_state.
library	Library name (e.g., "m" for libm, "R" for libR).

**Value**

Integer status code (0 = success).

---

tcc\_add\_library\_path      *Add a library path to a libtcc state*

---

**Description**

Add a library path to a libtcc state

**Usage**

```
tcc_add_library_path(state, path)
```

**Arguments**

state	A tcc_state.
path	Path to library directory.

**Value**

Integer status code (0 = success).

tcc\_add\_symbol            *Add a symbol to a libtcc state*

---

**Description**

Add a symbol to a libtcc state

**Usage**

```
tcc_add_symbol(state, name, addr)
```

**Arguments**

state	A tcc_state.
name	Symbol name.
addr	External pointer address or symbol value.

**Value**

Integer status code (0 = success).

---

tcc\_add\_sysinclude\_path            *Add a system include path to a libtcc state*

---

**Description**

Add a system include path to a libtcc state

**Usage**

```
tcc_add_sysinclude_path(state, path)
```

**Arguments**

state	A tcc_state.
path	Path to system include directory.

**Value**

Integer status code (0 = success).

tcc\_bind

*Bind symbols with type specifications***Description**

Define symbols with Bun-style type specifications for API mode. This is the core of the declarative FFI API.

**Usage**

```
tcc_bind(ffi, ...)
```

**Arguments**

`ffi` A `tcc_ffi` object

`...` Named list of symbol definitions. Each definition is a list with:

- `args`: List of fixed FFI argument types (e.g., `list("i32", "f64")`)
- `returns`: FFI type for return value (e.g., `"f64", "cstring"`)
- `variadic`: Set TRUE for C varargs functions
- `varargs`: Legacy typed variadic tail (exact/prefix mode)
- `varargs_types`: Allowed scalar FFI types for true variadic tails
- `varargs_min`: Minimum number of trailing varargs
- `varargs_max`: Maximum number of trailing varargs (required for true variadic mode, defaults to `varargs_min`)
- `code`: Optional C code for the symbol (for embedded functions)

Callback arguments should use the form `callback:<signature>` (e.g., `callback:double(double)`). The generated trampoline expects a `tcc_callback_ptr(cb)` to the corresponding user-data parameter in the C API. For thread-safe scheduling, use `callback_async:<signature>` which enqueues the call on the main thread and returns a default value immediately.

**Value**

Updated `tcc_ffi` object (for chaining)

**Examples**

```
ffi <- tcc_ffi() |>
  tcc_bind(
    add = list(args = list("i32", "i32"), returns = "i32"),
    greet = list(args = list("cstring"), returns = "cstring")
  )
```

---

tcc_call_symbol	<i>Call a symbol from a TinyCC state</i>
-----------------	--

---

### Description

With no additional arguments, `tcc_call_symbol()` preserves its historical quick-test behavior: call a zero-argument symbol and box an `int`, `double`, or `void` return value.

### Usage

```
tcc_call_symbol(
  .state,
  .NAME,
  ...,
  return = c("int", "double", "void"),
  NAOK = FALSE
)
```

### Arguments

<code>.state</code>	A <code>tcc_state</code> . Named <code>state =</code> is also accepted for compatibility with earlier releases.
<code>.NAME</code>	Symbol name to call. Named <code>name =</code> is also accepted for compatibility with earlier releases.
<code>...</code>	Optional arguments for <code>.C()</code> -style pointer calls.
<code>return</code>	One of <code>"int"</code> , <code>"double"</code> , or <code>"void"</code> . If <code>...</code> is present, the only supported value is <code>"void"</code> ; when omitted with <code>...</code> , it defaults to <code>"void"</code> .
<code>NAOK</code>	If <code>FALSE</code> , integer/logical NA and non-finite numeric/complex values are rejected before the call, matching <code>.C()</code> 's default safety check. If <code>TRUE</code> , those values are passed through.

### Details

With additional arguments, it uses an R `.C()`-style calling convention: the target C function must be `void`, each atomic or character R argument is copied to guarded mutable call storage and passed by pointer, and the result is a list of the modified argument values. Supported argument mappings follow R's `.C()` interface: raw as `unsigned char *`, integer/logical as `int *`, numeric as `double *` or `float *` when `attr(x, "Csingle")` is true, complex as `Rcomplex *`, character as `char **`, lists as read-only `SEXP *`, and functions/environments/other R objects as read-only `SEXP`. Up to 65 arguments are supported. Guard bytes around copied buffers are checked after the call to catch simple native underwrites and overwrites; character code may edit string contents in place but must not replace `char *` elements in the `char **` array.

Non-atomic R objects are borrowed for the duration of the call only. C code must not mutate them through this interface, and must call `R_PreserveObject()` if it deliberately stores a `SEXP` beyond the call. This is a low-level convenience interface; for typed scalar returns, explicit zero-copy arrays, ownership metadata, and clearer signatures, prefer `tcc_ffi()`.

**Value**

For zero-argument scalar calls, the boxed return value (NULL for void). For .C()-style calls, a list mirroring . . . with any C-side modifications copied back.

---

tcc_callback	<i>Register an R function as a callback</i>
--------------	---

---

**Description**

Wraps an R function so it can be passed as a C function pointer to compiled code. The callback will be invoked via a trampoline that marshals arguments between C and R.

**Usage**

```
tcc_callback(fun, signature, threadsafe = FALSE)
```

**Arguments**

fun	An R function to be called from C
signature	C function signature string (e.g., "double (*)(int, double)")
threadsafe	Whether to enable thread-safe invocation (experimental)

**Details**

Thread safety: callbacks are executed on the R main thread only. Invoking a callback from a worker thread is unsupported and may crash R. The threadsafe flag is currently informational only.

If a callback raises an error, a warning is emitted and a type-appropriate default value is returned.

When binding callbacks with `tcc_bind()`, use a `callback:<signature>` argument type so a synchronous trampoline is generated. The trampoline expects a `void*` user-data pointer as its first argument; pass `tcc_callback_ptr(cb)` as the user-data argument to the C API. For thread-safe usage from worker threads, use `callback_async:<signature>` which schedules the call on the main thread and returns a default value.

Pointer arguments (e.g., `double*`, `int*`) are passed as external pointers. Lengths must be supplied separately if needed.

The return type may be any scalar type supported by the FFI mappings (e.g., `i32`, `f64`, `bool`, `cstring`), or `SEXP` to return an R object directly.

Callback lifetime: callbacks are eventually released by finalizers and package unload. Call `tcc_callback_close()` when you want deterministic invalidation and earlier release of the preserved R function.

**Value**

A `tcc_callback` object (externalptr wrapper)

---

tcc\_callback\_async\_drain

*Drain the async callback queue*

---

### Description

Execute any pending async callbacks immediately on the main R thread. Normally callbacks fire automatically via R's event loop (input handler on POSIX, message pump on Windows), so explicit draining is only needed in test harnesses or tight batch loops that never yield to R's event loop.

### Usage

```
tcc_callback_async_drain()
```

### Details

TCC-compiled C code that is known to be running on the main R thread can call `RC_callback_async_drain_c()` directly instead of returning to R. Functions bound with `callback_async:*` arguments are normally executed on a worker by the generated wrapper while the main thread drains, so user code in that target function should not assume it is itself on the main thread.

### Value

NULL (invisible)

---

tcc\_callback\_async\_schedule

*Schedule a callback to run on the main thread*

---

### Description

Enqueue a callback for main-thread execution. Arguments must be basic scalars or external pointers.

### Usage

```
tcc_callback_async_schedule(callback, args = list())
```

### Arguments

callback	A tcc_callback object
args	List of arguments to pass to the callback

### Value

NULL (invisible)

---

tcc_callback_close	<i>Close/unregister a callback</i>
--------------------	------------------------------------

---

**Description**

Invalidates a callback immediately, releases the preserved R function reference, and cleans up callback resources as early as possible. This is recommended for deterministic lifetime management, but callbacks are also eventually released by finalizers if you simply drop all references.

**Usage**

```
tcc_callback_close(callback)
```

**Arguments**

callback	A tcc_callback object returned by tcc_callback()
----------	--

**Value**

NULL (invisible)

---

tcc_callback_ptr	<i>Get the C-compatible function pointer</i>
------------------	--

---

**Description**

Returns an external pointer that can be passed to compiled C code as user data for trampolines. Keep this handle (and the original tcc\_callback) alive for as long as C may call back. The pointer handle keeps the underlying token storage alive until it is garbage collected. Closing the original callback still invalidates the callback registry entry, so C must not continue invoking it after tcc\_callback\_close().

**Usage**

```
tcc_callback_ptr(callback)
```

**Arguments**

callback	A tcc_callback object
----------	-----------------------

**Details**

Pointer arguments and return values are treated as external pointers. Use tcc\_read\_bytes(), tcc\_read\_u8(), or tcc\_read\_f64() to inspect pointed data when needed.

**Value**

An external pointer (address of the callback token)

---

tcc_callback_valid	<i>Check if callback is still valid</i>
--------------------	---

---

**Description**

Check if callback is still valid

**Usage**

```
tcc_callback_valid(callback)
```

**Arguments**

callback          A tcc\_callback object

**Value**

Logical indicating if callback can be invoked

---

tcc_compile	<i>Compile FFI bindings</i>
-------------	-----------------------------

---

**Description**

Compile the defined symbols into callable functions. This generates C wrapper code and compiles it with TinyCC.

**Usage**

```
tcc_compile(ffl, verbose = FALSE)
```

**Arguments**

ffi                A tcc\_ffl object  
verbose            Print compilation info

**Value**

A tcc\_compiled object with callable functions

---

tcc\_compile\_string     *Compile C code from a character string*

---

**Description**

Compile C code from a character string

**Usage**

```
tcc_compile_string(state, code)
```

**Arguments**

state	A tcc_state.
code	C source code string.

**Value**

Integer status code (0 = success).

---

tcc\_container\_of     *Generate container\_of helper for struct member*

---

**Description**

Creates a function that recovers the parent struct pointer from a pointer to one of its members. This is the classic Linux kernel container\_of macro made accessible from R.

**Usage**

```
tcc_container_of(ffl, struct_name, member_name)
```

**Arguments**

ffl	A tcc_ffl object
struct_name	Struct name
member_name	Member field name to compute offset from

**Value**

Updated tcc\_ffl object

**Examples**

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_struct("student", list(id = "i32", marks = "i32")) |>
  tcc_container_of("student", "marks") # Creates struct_student_from_marks()

## End(Not run)
```

---

tcc_cstring	<i>Create a C-style string pointer</i>
-------------	--

---

**Description**

Convert R character strings to C-style null-terminated string pointers. This handles UTF-8 encoding and null termination automatically.

**Usage**

```
tcc_cstring(str)
```

**Arguments**

str	Character string
-----	------------------

**Value**

An external pointer tagged "rtinycc\_owned" pointing to a malloc'd copy of the string. Freed on garbage collection or via [tcc\\_free\(\)](#).

---

tcc_cstring_object	<i>CString S3 Class</i>
--------------------	-------------------------

---

**Description**

Wrapper around a C string pointer with an optional cached R copy. Ownership follows the underlying external pointer; this wrapper does not add finalizer or freeing behavior on top of that pointer.

**Usage**

```
tcc_cstring_object(ptr, clone = TRUE, owned = FALSE)
```

**Arguments**

ptr	External pointer to C string
clone	Whether to clone the string immediately (safe for R use)
owned	Currently unused. Reserved for future finalizer support.

**Value**

A tcc\_cstring object

---

tcc_data_ptr	<i>Dereference a pointer-to-pointer</i>
--------------	---

---

**Description**

Treats ptr\_ref as a pointer to a pointer and returns the pointed address as an external pointer. This is useful for fields like void\*\* or T\*\*.

**Usage**

```
tcc_data_ptr(ptr_ref)
```

**Arguments**

ptr_ref	External pointer to a pointer value (e.g., address of a field).
---------	---

**Value**

An external pointer tagged "rtinycc\_borrowed". Not owned by Rtinycc and never freed on garbage collection. Do not pass to [tcc\\_free\(\)](#).

---

tcc_enum	<i>Declare enum for FFI helper generation</i>
----------	---

---

**Description**

Generate R-callable helpers for enum constants and type conversions. The enum must be defined in a header.

**Usage**

```
tcc_enum(fffi, name, constants = NULL, export_constants = FALSE)
```

**Arguments**

fffi	A tcc_fffi object
name	Enum name (as defined in C header)
constants	Character vector of constant names to export
export_constants	Whether to export enum constants as R functions

**Value**

Updated tcc\_ffi object

**Examples**

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_header("#include <errors.h>") |>
  tcc_enum("error_code", constants = c("OK", "ERROR"), export_constants = TRUE)

## End(Not run)
```

---

tcc_ffi	<i>Create a new FFI compilation context</i>
---------	---

---

**Description**

Initialize a Bun-style FFI context for API-mode compilation. This is the entry point for the modern FFI API.

**Usage**

```
tcc_ffi()
```

**Value**

A tcc\_ffi object with chaining support

**Examples**

```
ffi <- tcc_ffi()
```

---

tcc_field_addr	<i>Generate field address getter helpers</i>
----------------	--

---

**Description**

Creates functions that return pointers to specific struct fields. Useful for passing field pointers to C functions or for container\_of.

**Usage**

```
tcc_field_addr(ffi, struct_name, fields)
```

**Arguments**

ffi	A tcc_ffi object
struct_name	Struct name
fields	Character vector of field names

**Value**

Updated tcc\_ffi object

**Examples**

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_struct("point", list(x = "f64", y = "f64")) |>
  tcc_field_addr("point", c("x", "y")) # point_x_addr(), point_y_addr()

## End(Not run)
```

---

tcc\_free

*Free owned memory*


---

**Description**

Free memory whose external pointer is tagged "rtinycc\_owned" (e.g. from [tcc\\_malloc\(\)](#) or [tcc\\_cstring\(\)](#)). Errors on struct pointers (use the generated struct\_<name>\_free()) or borrowed pointers from [tcc\\_data\\_ptr\(\)](#).

**Usage**

```
tcc_free(ptr)
```

**Arguments**

ptr	External pointer to free
-----	--------------------------

**Value**

NULL.

---

tcc\_generate\_bindings *Generate bindings from header declarations*

---

## Description

Generate bindings from header declarations

## Usage

```
tcc_generate_bindings(
  ffi = NULL,
  header,
  mapper = tcc_map_c_type_to_ffi,
  functions = TRUE,
  structs = TRUE,
  unions = TRUE,
  enums = TRUE,
  globals = TRUE,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

## Arguments

ffi	A tcc_ffi object. If NULL, a new one is created.
header	Character scalar containing C declarations.
mapper	Function to map C types to FFI types.
functions	Logical; generate tcc_bind() specs for functions.
structs	Logical; generate tcc_struct() helpers.
unions	Logical; generate tcc_union() helpers.
enums	Logical; generate tcc_enum() helpers.
globals	Logical; generate tcc_global() getters/setters.
bitfield_type	FFI type to use for bitfields.
include_bitfields	Whether to include bitfields.

## Value

Updated tcc\_ffi object.

## Examples

```
## Not run:
header <- "double sqrt(double x); struct point { double x; double y; };"
ffi <- tcc_generate_bindings(tcc_ffi(), header)

## End(Not run)
```

---

tcc_get_symbol	<i>Get a symbol pointer from a libtcc state</i>
----------------	---

---

**Description**

Get a symbol pointer from a libtcc state

**Usage**

```
tcc_get_symbol(state, name)
```

**Arguments**

state	A tcc_state.
name	Symbol name to look up.

**Value**

External pointer of class tcc\_symbol.

---

tcc_global	<i>Declare a global variable getter</i>
------------	---

---

**Description**

Register a global C symbol so the compiled object exposes getter/setter functions `global_<name>_get()` and `global_<name>_set()`.

**Usage**

```
tcc_global(ffl, name, type)
```

**Arguments**

ffi	A tcc_ffi object
name	Global symbol name
type	FFI type for the global (scalar types only)

**Details**

Globals are limited to scalar FFI types. Array types are rejected. Scalar conversions follow the same rules as wrapper arguments:

- Integer inputs (i8, i16, i32, u8, u16) must be finite and within range; NA values error.
- Large integer types (i64, u32, u64) are mediated through R numeric (double). Values must be integer-valued and within range; for i64/u64 only exact integers up to  $2^{53}$  are accepted.
- Getter wrappers for i64/u64 warn when the stored value exceeds R's exact integer range for numeric vectors.
- bool rejects NA logicals.

Ownership notes:

- ptr globals store the raw address from an external pointer. If the external pointer owns memory, keep it alive; otherwise the pointer may be freed while the global still points to it.
- cstring globals store a borrowed pointer to R's string data (UTF-8 translation). Do not free it; for C-owned strings prefer a ptr global and manage lifetime explicitly (e.g., with `tcc_cstring()`).

**Value**

Updated `tcc_ffi` object (for chaining)

**Note**

Global helpers are generated inside the compiled TCC unit. Recompiling creates a new instance of the global variable; existing compiled objects continue to refer to their own copy.

**Examples**

```
ffi <- tcc_ffi() |>
  tcc_source("int global_counter = 7;") |>
  tcc_global("global_counter", "i32") |>
  tcc_compile()
ffi$global_global_counter_get()
```

---

tcc\_header

*Add C headers*

---

**Description**

Add C headers

**Usage**

```
tcc_header(ffi, header)
```

**Arguments**

ffi	A tcc_ffi object
header	Header string or include directive

**Value**

Updated tcc\_ffi object (for chaining)

---

tcc_include	<i>Add include path to FFI context</i>
-------------	--

---

**Description**

Add include path to FFI context

**Usage**

```
tcc_include(ffi, path)
```

**Arguments**

ffi	A tcc_ffi object
path	Include directory path

**Value**

Updated tcc\_ffi object (for chaining)

---

tcc_include_paths	<i>TinyCC include search paths</i>
-------------------	------------------------------------

---

**Description**

Returns the include directories used by the bundled TinyCC (top-level include and lib/tcc/include).

**Usage**

```
tcc_include_paths()

tcc_sysinclude_paths()
```

**Value**

A character vector of include directories.

---

tcc_introspect	<i>Enable introspection helpers</i>
----------------	-------------------------------------

---

**Description**

Generates sizeof, alignof, and offsetof helper functions for structs, unions, and enums. Useful for debugging or when you need to know C layout information from R.

**Usage**

```
tcc_introspect(ffl)
```

**Arguments**

ffl	A tcc_ffl object
-----	------------------

**Value**

Updated tcc\_ffl object

---

tcc_library	<i>Add library to link against</i>
-------------	------------------------------------

---

**Description**

Add library to link against

**Usage**

```
tcc_library(ffl, library)
```

**Arguments**

ffl	A tcc_ffl object
library	Library name (e.g., "m", "sqlite3") or a path to a shared library (e.g., "libm.so.6"). When a path or platform library file name is provided, the library directory is added automatically and TinyCC is asked to link that exact file name. This keeps versioned runtime libraries such as libm.so.6 distinct from generic linker names such as m/libm.so.

**Value**

Updated tcc\_ffl object (for chaining)

---

tcc_library_path	<i>Add library path to FFI context</i>
------------------	--

---

**Description**

Add library path to FFI context

**Usage**

```
tcc_library_path(ffi, path)
```

**Arguments**

ffi	A tcc_ffi object
path	Library directory path

**Value**

Updated tcc\_ffi object (for chaining)

---

tcc_link	<i>Link an external shared library with Bun-style FFI bindings</i>
----------	--

---

**Description**

Link a system library (like libsqlite3) and generate type-safe wrappers automatically using TinyCC JIT compilation (API mode). Unlike `dlopen()`, this uses TinyCC to compile bindings that handle type conversion between R and C automatically.

**Usage**

```
tcc_link(  
  path,  
  symbols,  
  headers = NULL,  
  libs = character(0),  
  lib_paths = character(0),  
  include_paths = character(0),  
  user_code = NULL,  
  verbose = FALSE  
)
```

**Arguments**

path	Library short name (e.g., "m", "sqlite3") or full path to the shared library. Short names stay on the normal linker-name path (-l<name>). File names such as libm.so.6 or full paths are resolved through the configured library search paths when needed and linked as exact files rather than collapsed to generic names like m.
symbols	Named list of symbol definitions with: <ul style="list-style-type: none"> <li>• args: List of FFI types for arguments</li> <li>• returns: FFI type for return value</li> </ul>
headers	Optional C headers to include
libs	Library names to link (e.g., "sqlite3")
lib_paths	Additional library search paths
include_paths	Additional include search paths
user_code	Optional custom C code to include in the compilation
verbose	Print debug information

**Value**

A tcc\_compiled object with callable functions

**Examples**

```
## Not run:
# Link SQLite with type-safe bindings
sqlite <- tcc_link(
  "libsqlite3.so",
  symbols = list(
    sqlite3_libversion = list(args = list(), returns = "cstring"),
    sqlite3_open = list(args = list("cstring", "ptr"), returns = "i32")
  ),
  libs = "sqlite3"
)

# Call directly - type conversion happens automatically
sqlite$sqlite3_libversion()

# Example with custom user code for helper functions
math_with_helpers <- tcc_link(
  "m",
  symbols = list(
    sqrt = list(args = list("f64"), returns = "f64"),
    safe_sqrt = list(args = list("f64"), returns = "f64")
  ),
  user_code = "
#include <math.h>

// Helper function that validates input before calling sqrt
double safe_sqrt(double x) {
```

```

        if (x < 0) {
            return NAN;
        }
        return sqrt(x);
    }
    ",
    libs = "m"
)
math_with_helpers$safe_sqrt(16.0)
math_with_helpers$safe_sqrt(-4.0) # Returns NaN for negative input

## End(Not run)

```

---

tcc_list_symbols	<i>List symbols known to a libtcc state</i>
------------------	---

---

### Description

Return the global symbols currently reported by libtcc for a state. This is a best-effort symbol-table inspection helper for compiled/linked TCC states, not a portable exhaustive symbol enumerator, not a DLL export scanner, and not a C signature discovery API. Platform backends may omit symbols that are still resolvable with `tcc_get_symbol()`. For meaningful runtime addresses, call it after `tcc_relocate()`.

### Usage

```
tcc_list_symbols(state)
```

### Arguments

state            A tcc\_state.

### Value

A data frame with columns name and address, where address is a hexadecimal character string.

---

tcc_malloc	<i>Allocate memory buffer</i>
------------	-------------------------------

---

### Description

Allocate a memory buffer of specified size, equivalent to C malloc. Returns an external pointer that can be passed to FFI functions.

### Usage

```
tcc_malloc(size)
```

**Arguments**

size                    Number of bytes to allocate

**Value**

An external pointer tagged "rtinyc\_owned" with an R finalizer. Freed on garbage collection or explicitly via `tcc_free()`.

---

`tcc_map_c_type_to_ffi` *Map a C type string to an Rtinyc FFI type*

---

**Description**

Map a C type string to an Rtinyc FFI type

**Usage**

```
tcc_map_c_type_to_ffi(c_type)
```

**Arguments**

c\_type                    C type string (e.g., "int", "double", "char \*").

**Details**

This mapper is intentionally conservative for pointer types. `char*` is treated as a raw pointer (`ptr`) because C does not guarantee NUL-terminated strings. If you know the API expects a C string, map it explicitly to `cstring` in your custom mapper.

**Value**

A single FFI type string.

**Examples**

```
## Not run:
tcc_map_c_type_to_ffi("int")
tcc_map_c_type_to_ffi("const char *")

## End(Not run)
```

---

tcc_null_ptr	<i>Create a NULL pointer</i>
--------------	------------------------------

---

**Description**

Creates a NULL pointer equivalent for use with FFI functions that expect optional pointer arguments.

**Usage**

```
tcc_null_ptr()
```

**Value**

An external pointer with NULL address

---

tcc_options	<i>Add TinyCC compiler options to FFI context</i>
-------------	---

---

**Description**

Append raw options that are passed to `tcc_set_options()` before compiling generated wrappers (for example `"-O2"` or `"-Wall"`).

**Usage**

```
tcc_options(ffi, options)
```

**Arguments**

ffi	A tcc_ffi object
options	Character vector of option fragments

**Value**

Updated tcc\_ffi object (for chaining)

---

tcc_output	<i>Set output type for FFI compilation</i>
------------	--

---

**Description**

Set output type for FFI compilation

**Usage**

```
tcc_output(ffi, output = c("memory", "dll", "exe"))
```

**Arguments**

ffi	A tcc_ffi object
output	One of "memory", "dll", "exe"

**Value**

Updated tcc\_ffi object (for chaining)

---

tcc_path	<i>Locate the TinyCC executable</i>
----------	-------------------------------------

---

**Description**

Returns the platform-specific tcc binary path (or tcc.exe on Windows), preferring the bundled installation.

**Usage**

```
tcc_path()
```

**Value**

A character scalar path.

---

tcc_prefix	<i>TinyCC paths</i>
------------	---------------------

---

**Description**

Helpers to locate the bundled tinycc installation after the package is installed.

**Usage**

tcc\_prefix()

tcc\_lib\_path()

tcc\_lib\_paths()

tcc\_include\_path()

tcc\_bin\_path()

tcc\_cli()

**Value**

A character scalar path.

---

tcc_ptr_addr	<i>Get pointer address as integer</i>
--------------	---------------------------------------

---

**Description**

Get the numeric address of an external pointer, useful for debugging and when APIs require pointer addresses as integers. Optional hex mode available.

**Usage**

tcc\_ptr\_addr(ptr, hex = FALSE)

**Arguments**

ptr            External pointer

hex            Whether to display in hexadecimal (default: FALSE)

**Value**

Character representation of pointer address (hex if requested, decimal otherwise)

---

`tcc_ptr_free_set_null` *Free the pointed memory and set to NULL*

---

**Description**

Frees the memory pointed to by `ptr_ref` and sets the pointer to NULL. Use this only when the pointed memory is not already owned by another external pointer with its own finalizer.

**Usage**

```
tcc_ptr_free_set_null(ptr_ref)
```

**Arguments**

`ptr_ref`            External pointer to a pointer value.

**Value**

The updated pointer reference (invisibly).

---

`tcc_ptr_is_null`            *Check whether an external pointer is NULL*

---

**Description**

Returns TRUE if the external pointer address is NULL, FALSE otherwise.

**Usage**

```
tcc_ptr_is_null(ptr)
```

**Arguments**

`ptr`                External pointer

**Value**

Logical scalar

---

tcc_ptr_is_owned	<i>Check for the "rtinyc_owned" tag</i>
------------------	---

---

**Description**

Returns TRUE only for pointers created by `tcc_malloc()` or `tcc_cstring()`. Struct pointers (tagged "struct\_<name>") and borrowed pointers return FALSE.

**Usage**

```
tcc_ptr_is_owned(ptr)
```

**Arguments**

ptr	External pointer
-----	------------------

**Value**

Logical scalar

---

tcc_ptr_set	<i>Set a pointer-to-pointer value</i>
-------------	---------------------------------------

---

**Description**

Assigns the address in `ptr_value` to the location pointed to by `ptr_ref`.

**Usage**

```
tcc_ptr_set(ptr_ref, ptr_value)
```

**Arguments**

ptr_ref	External pointer to a pointer value (e.g., address of a field).
ptr_value	External pointer to store.

**Value**

The updated pointer reference (invisibly).

---

tcc_ptr_utils	<i>Pointer and Buffer Utilities for FFI</i>
---------------	---

---

**Description**

Generic helper functions for common FFI operations inspired by Bun's FFI. These utilities handle pointer creation, buffer management, and memory operations that are commonly needed when working with external libraries.

**Value**

No return value. This is a documentation topic that groups the pointer and buffer utility functions described below.

---

tcc_read_bytes	<i>Read raw bytes from a pointer</i>
----------------	--------------------------------------

---

**Description**

Read a fixed number of bytes from an external pointer into a raw vector.

**Usage**

```
tcc_read_bytes(ptr, nbytes)
```

**Arguments**

ptr	External pointer
nbytes	Number of bytes to read

**Value**

Raw vector

---

tcc_read_cstring	<i>Read C-style string from pointer</i>
------------------	---

---

**Description**

Convert a C-style null-terminated string pointer back to R character string. Handles UTF-8 decoding automatically.

**Usage**

```
tcc_read_cstring(
  ptr,
  max_bytes = NULL,
  null_action = c("na", "empty", "error")
)
```

**Arguments**

ptr	External pointer to C string
max_bytes	Optional maximum number of bytes to read (fixed-length read).
null_action	Behavior when ptr is NULL: one of "na", "empty", "error". Only effective when max_bytes is provided; without it a NULL pointer returns "".

**Value**

Character string, or NA/"" for NULL pointers depending on null\_action.

---

tcc_read_f32	<i>Read 32-bit float</i>
--------------	--------------------------

---

**Description**

Read 32-bit float

**Usage**

```
tcc_read_f32(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Numeric scalar (promoted to double).

---

tcc_read_f64	<i>Read 64-bit doubles from a pointer</i>
--------------	---

---

**Description**

Read 64-bit doubles from a pointer

**Usage**

```
tcc_read_f64(ptr, n = NULL, offset = 0L)
```

**Arguments**

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive f64 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Numeric scalar (offset form) or numeric vector (n form).

---

tcc_read_i16	<i>Read signed 16-bit integer</i>
--------------	-----------------------------------

---

**Description**

Read signed 16-bit integer

**Usage**

```
tcc_read_i16(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Integer scalar

---

tcc_read_i32	<i>Read signed 32-bit integers from a pointer</i>
--------------	---

---

**Description**

Read signed 32-bit integers from a pointer

**Usage**

```
tcc_read_i32(ptr, n = NULL, offset = 0L)
```

**Arguments**

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive i32 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Integer scalar (offset form) or integer vector (n form).

---

tcc_read_i64	<i>Read signed 64-bit integer</i>
--------------	-----------------------------------

---

**Description**

Read signed 64-bit integer

**Usage**

```
tcc_read_i64(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Numeric scalar (double, exact up to  $2^{53}$ ).

---

tcc_read_i8	<i>Read signed 8-bit integer</i>
-------------	----------------------------------

---

**Description**

Read signed 8-bit integer

**Usage**

```
tcc_read_i8(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Integer scalar

---

tcc_read_ptr	<i>Read a pointer at byte offset</i>
--------------	--------------------------------------

---

**Description**

Dereferences a void\* at the given byte offset from ptr. Equivalent to \*(void\*\*)(ptr + offset). The returned pointer is tagged "rtinycc\_borrowed" and will not be freed by the garbage collector.

**Usage**

```
tcc_read_ptr(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

External pointer

---

tcc_read_u16	<i>Read unsigned 16-bit integer</i>
--------------	-------------------------------------

---

**Description**

Read unsigned 16-bit integer

**Usage**

```
tcc_read_u16(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Integer scalar

---

tcc_read_u32	<i>Read unsigned 32-bit integer</i>
--------------	-------------------------------------

---

**Description**

Read unsigned 32-bit integer

**Usage**

```
tcc_read_u32(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Numeric scalar (double, exact up to  $2^{32}-1$ ).

---

tcc_read_u64	<i>Read unsigned 64-bit integer</i>
--------------	-------------------------------------

---

**Description**

Read unsigned 64-bit integer

**Usage**

```
tcc_read_u64(ptr, offset = 0L)
```

**Arguments**

ptr	External pointer
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Numeric scalar (double, exact up to  $2^{53}$ ).

---

tcc_read_u8	<i>Read unsigned 8-bit values from a pointer</i>
-------------	--

---

**Description**

Read unsigned 8-bit values from a pointer

**Usage**

```
tcc_read_u8(ptr, n = NULL, offset = 0L)
```

**Arguments**

ptr	External pointer
n	Number of values to read (legacy vectorised interface). If provided, reads n consecutive u8 values starting at byte 0.
offset	Byte offset from ptr (scalar interface). Ignored when n is supplied.

**Value**

Integer scalar (offset form) or integer vector (n form).

---

tcc_recompile	<i>Recompile a tcc_compiled object</i>
---------------	--

---

**Description**

Explicitly recompile from the stored FFI recipe. Useful after deserialization (`readRDS`, `unserialize`) or to force a fresh compilation.

**Usage**

```
tcc_recompile(compiled)
```

**Arguments**

compiled	A <code>tcc_compiled</code> object
----------	------------------------------------

**Value**

The recompiled `tcc_compiled` object (invisibly, same environment)

---

tcc_relocate	<i>Relocate compiled code</i>
--------------	-------------------------------

---

**Description**

Relocate compiled code

**Usage**

```
tcc_relocate(state)
```

**Arguments**

state	A <code>tcc_state</code> .
-------	----------------------------

**Value**

Integer status code (0 = success).

---

tcc_run_cli	<i>Run the tinycc CLI</i>
-------------	---------------------------

---

**Description**

Run the tinycc CLI

**Usage**

```
tcc_run_cli(args = character(), tcc_path = check_cli_exists())
```

**Arguments**

args	Character vector of CLI arguments (e.g., c("-c", file, "-o", out)).
tcc_path	Optional path to the tcc binary; defaults to the bundled CLI.

**Value**

Integer status from system2().

---

tcc_set_options	<i>Apply raw TinyCC options to a libtcc state</i>
-----------------	---

---

**Description**

Passes options directly to tcc\_set\_options() for the given state.

**Usage**

```
tcc_set_options(state, options)
```

**Arguments**

state	A tcc_state.
options	Character scalar of options (for example "-O2 -Wall").

**Value**

Integer status code (0 on success; negative on parse error).

---

tcc_source	<i>Add C source code</i>
------------	--------------------------

---

**Description**

Add C source code

**Usage**

```
tcc_source(ffl, code)
```

**Arguments**

ffl	A tcc_ffl object
code	C source code string

**Value**

Updated tcc\_ffl object (for chaining)

---

tcc_state	<i>Create a libtcc state</i>
-----------	------------------------------

---

**Description**

Initialize a libtcc compilation state, optionally pointing at the bundled include/lib paths.

**Usage**

```
tcc_state(
    output = c("memory", "obj", "dll", "exe", "preprocess"),
    include_path = tcc_include_paths(),
    lib_path = tcc_lib_paths()
)
```

**Arguments**

output	Output type: one of "memory", "obj", "dll", "exe", "preprocess".
include_path	Path(s) to headers; defaults to the bundled include dirs.
lib_path	Path(s) to libraries; defaults to the bundled lib dirs (lib and lib/tcc).

**Value**

An external pointer of class tcc\_state.

---

tcc_struct	<i>Declare struct for FFI helper generation</i>
------------	---

---

### Description

Generate R-callable helpers for struct allocation, field access, and pointer management. The struct must be defined in a header.

### Usage

```
tcc_struct(ffi, name, accessors)
```

### Arguments

ffi	A tcc_ffi object
name	Struct name (as defined in C header)
accessors	Named list of field accessors where names are field names and values are FFI types (e.g., list(x="f64", y="f64")). Named nested struct fields can use "struct:<name>" to generate borrowed nested-view getters and copy-in setters (for example child = "struct:child").

### Value

Updated tcc\_ffi object

### Examples

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_header("#include <point.h>") |>
  tcc_struct("point", list(x = "f64", y = "f64", id = "i32"))

## End(Not run)
```

---

tcc_struct_raw_access	<i>Enable raw byte access for struct</i>
-----------------------	--

---

### Description

Generates helper functions to read/write raw bytes from struct memory. Useful for bitwise operations, debugging, or manual serialization.

### Usage

```
tcc_struct_raw_access(ffi, struct_name)
```

**Arguments**

ffi	A tcc_ffi object
struct_name	Struct name

**Value**

Updated tcc\_ffi object

---

tcc\_symbol\_is\_valid    *Check if a tcc\_symbol external pointer is valid*

---

**Description**

Check if a tcc\_symbol external pointer is valid

**Usage**

```
tcc_symbol_is_valid(ptr)
```

**Arguments**

ptr	External pointer from tcc_get_symbol().
-----	---

**Value**

TRUE if the pointer address is non-null, FALSE otherwise.

---

tcc\_treesitter\_bindings    *Generate bindings from a header*

---

**Description**

Generate bindings from a header

**Usage**

```
tcc_treesitter_bindings(
  header,
  mapper = tcc_map_c_type_to_ffi,
  ffi = NULL,
  functions = TRUE,
  structs = FALSE,
  unions = FALSE,
  enums = FALSE,
```

```

globals = FALSE,
bitfield_type = "u8",
include_bitfields = TRUE
)

```

### Arguments

header	Character scalar containing C declarations.
mapper	Function to map C types to FFI types.
ffi	Optional tcc_ffi object. When provided, returns an updated FFI object with generated bindings.
functions	Logical; generate tcc_bind() specs for functions.
structs	Logical; generate tcc_struct() helpers.
unions	Logical; generate tcc_union() helpers.
enums	Logical; generate tcc_enum() helpers.
globals	Logical; generate tcc_global() getters/setters.
bitfield_type	FFI type to use for bitfields.
include_bitfields	Whether to include bitfields.

### Value

Named list suitable for tcc\_bind() when ffi is NULL, otherwise an updated tcc\_ffi object.

### Examples

```

## Not run:
header <- "double sqrt(double x);"
symbols <- tcc_treesitter_bindings(header)

## End(Not run)

```

---

```
tcc_treesitter_defines
```

*Extract macro defines from a header file*

---

### Description

Extract macro defines from a header file

### Usage

```

tcc_treesitter_defines(
  file,
  use_cpp = TRUE,
  cc = treesitter.c::r_cc(),
  ccflags = NULL
)

```

**Arguments**

file	Path to a header file.
use_cpp	Logical; use the C preprocessor if available.
cc	Compiler string; passed to system2() if use_cpp = TRUE.
ccflags	Additional flags for the compiler.

**Value**

Character vector of macro names defined in file.

**Examples**

```
## Not run:
tcc_treesitter_defines("/usr/include/math.h")

## End(Not run)
```

---

```
tcc_treesitter_enum_bindings
```

*Apply tcc\_enum() bindings from a header*

---

**Description**

Apply tcc\_enum() bindings from a header

**Usage**

```
tcc_treesitter_enum_bindings(ffl, header, constants = NULL)
```

**Arguments**

ffi	A tcc_ffi object.
header	Character scalar containing C declarations.
constants	Named list of enum constants.

**Value**

Updated tcc\_ffi object.

**Examples**

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
ffi <- tcc_ffi()
ffi <- tcc_treesitter_enum_bindings(ffi, header, constants = list(status = c("OK", "ERR")))

## End(Not run)
```

---

```
tcc_treesitter_enum_members
```

*Parse enum members with treesitter.c*

---

### Description

Parse enum members with treesitter.c

### Usage

```
tcc_treesitter_enum_members(header, ...)
```

### Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to <code>treesitter.c::get_enum_members_from_root()</code> .

### Value

A data frame of enum members.

### Examples

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
tcc_treesitter_enum_members(header)

## End(Not run)
```

---

```
tcc_treesitter_enums Parse enum declarations with treesitter.c
```

---

### Description

Parse enum declarations with treesitter.c

### Usage

```
tcc_treesitter_enums(header, ...)
```

### Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to <code>treesitter.c::get_enum_nodes()</code> .

**Value**

A data frame of enum nodes.

**Examples**

```
## Not run:
header <- "enum status { OK = 0, ERR = 1 };"
tcc_treesitter_enums(header)

## End(Not run)
```

---

tcc\_treesitter\_functions

*Parse function declarations with treesitter.c*

---

**Description**

Parse function declarations with treesitter.c

**Usage**

```
tcc_treesitter_functions(header, ...)
```

**Arguments**

header	Character scalar containing C declarations.
...	Additional arguments passed to <code>treesitter.c::get_function_nodes()</code> .

**Value**

A data frame of function nodes.

**Examples**

```
## Not run:
header <- "double sqrt(double x);"
tcc_treesitter_functions(header)

## End(Not run)
```

---

`tcc_treesitter_global_types`*Parse global declarations with types using treesitter.c*

---

**Description**

Parse global declarations with types using treesitter.c

**Usage**

```
tcc_treesitter_global_types(header, ...)
```

**Arguments**

<code>header</code>	Character scalar containing C declarations.
<code>...</code>	Additional arguments passed to <code>treesitter.c::get_globals_with_types_from_root()</code> .

**Value**

A data frame of global names and C types.

**Examples**

```
## Not run:  
header <- "int global_counter;"  
tcc_treesitter_global_types(header)  
  
## End(Not run)
```

---

`tcc_treesitter_globals`*Parse global declarations with treesitter.c*

---

**Description**

Parse global declarations with treesitter.c

**Usage**

```
tcc_treesitter_globals(header, ...)
```

**Arguments**

<code>header</code>	Character scalar containing C declarations.
<code>...</code>	Additional arguments passed to <code>treesitter.c::get_globals_from_root()</code> .

**Value**

A data frame of global names.

**Examples**

```
## Not run:
header <- "int global_counter;"
tcc_treesitter_globals(header)

## End(Not run)
```

---

```
tcc_treesitter_struct_accessors
```

*Generate tcc\_struct() accessors from header structs*

---

**Description**

Generate tcc\_struct() accessors from header structs

**Usage**

```
tcc_treesitter_struct_accessors(
  header,
  mapper = tcc_map_c_type_to_ffi,
  bitfield_type = "u8",
  include_bitfields = TRUE
)
```

**Arguments**

header	Character scalar containing C declarations.
mapper	Function to map C types to FFI types.
bitfield_type	FFI type to use for bitfields.
include_bitfields	Whether to include bitfields.

**Value**

Named list of accessors by struct name. Bitfields are returned as lists with type, bitfield = TRUE, and width. Named nested struct fields are returned as "struct:<name>"; ambiguous or anonymous nested structs fall back to "ptr".

**Examples**

```
## Not run:
header <- "struct point { double x; double y; };"
tcc_treesitter_struct_accessors(header)

## End(Not run)
```

---

tcc\_treesitter\_struct\_bindings

*Apply tcc\_struct() bindings from a header*

---

### Description

Apply tcc\_struct() bindings from a header

### Usage

```
tcc_treesitter_struct_bindings(ffi, header, ...)
```

### Arguments

ffi	A tcc_ffi object.
header	Character scalar containing C declarations.
...	Passed to tcc_treesitter_struct_accessors().

### Value

Updated tcc\_ffi object.

### Examples

```
## Not run:
header <- "struct point { double x; double y; };"
ffi <- tcc_ffi()
ffi <- tcc_treesitter_struct_bindings(ffi, header)

## End(Not run)
```

---

tcc\_treesitter\_struct\_members

*Parse struct members (including bitfields) with treesitter.c*

---

### Description

Parse struct members (including bitfields) with treesitter.c

### Usage

```
tcc_treesitter_struct_members(header, ...)
```

### Arguments

header	Character scalar containing C declarations.
...	Additional arguments passed to treesitter.c::get_struct_members().

**Value**

A data frame of struct members.

**Examples**

```
## Not run:  
header <- "struct point { double x; double y; };"  
tcc_treesitter_struct_members(header)  
  
## End(Not run)
```

---

tcc\_treesitter\_structs

*Parse struct declarations with treesitter.c*

---

**Description**

Parse struct declarations with treesitter.c

**Usage**

```
tcc_treesitter_structs(header, ...)
```

**Arguments**

header	Character scalar containing C declarations.
...	Additional arguments passed to <code>treesitter.c::get_struct_nodes()</code> .

**Value**

A data frame of struct nodes.

**Examples**

```
## Not run:  
header <- "struct point { double x; double y; };"  
tcc_treesitter_structs(header)  
  
## End(Not run)
```

---

`tcc_treesitter_union_accessors`*Generate tcc\_union() accessors from header unions*

---

## Description

Generate tcc\_union() accessors from header unions

## Usage

```
tcc_treesitter_union_accessors(  
  header,  
  mapper = tcc_map_c_type_to_ffi,  
  bitfield_type = "u8",  
  include_bitfields = TRUE  
)
```

## Arguments

header	Character scalar containing C declarations.
mapper	Function to map C types to FFI types.
bitfield_type	FFI type to use for bitfields.
include_bitfields	Whether to include bitfields.

## Value

Named list of accessors by union name. Bitfields are returned as lists with type, bitfield = TRUE, and width. Nested struct members are returned as list(type = "struct", struct\_name = <name>) when the struct name is available, otherwise list(type = "struct").

## Examples

```
## Not run:  
header <- "union data { int i; double d; };"  
tcc_treesitter_union_accessors(header)  
  
## End(Not run)
```

---

`tcc_treesitter_union_bindings`*Apply tcc\_union() bindings from a header*

---

**Description**

Apply tcc\_union() bindings from a header

**Usage**

```
tcc_treesitter_union_bindings(ffi, header, ...)
```

**Arguments**

<code>ffi</code>	A tcc_ffi object.
<code>header</code>	Character scalar containing C declarations.
<code>...</code>	Passed to tcc_treesitter_union_accessors().

**Value**

Updated tcc\_ffi object.

**Examples**

```
## Not run:  
header <- "union data { int i; double d; };"  
ffi <- tcc_ffi()  
ffi <- tcc_treesitter_union_bindings(ffi, header)  
  
## End(Not run)
```

---

`tcc_treesitter_union_members`*Parse union members with treesitter.c*

---

**Description**

Parse union members with treesitter.c

**Usage**

```
tcc_treesitter_union_members(header, ...)
```

**Arguments**

<code>header</code>	Character scalar containing C declarations.
<code>...</code>	Additional arguments passed to treesitter.c::get_union_members_from_root().

**Value**

A data frame of union members.

**Examples**

```
## Not run:  
header <- "union data { int i; double d; };"  
tcc_treesitter_union_members(header)  
  
## End(Not run)
```

---

tcc\_treesitter\_unions *Parse union declarations with treesitter.c*

---

**Description**

Parse union declarations with treesitter.c

**Usage**

```
tcc_treesitter_unions(header, ...)
```

**Arguments**

header	Character scalar containing C declarations.
...	Additional arguments passed to <code>treesitter.c::get_union_nodes()</code> .

**Value**

A data frame of union nodes.

**Examples**

```
## Not run:  
header <- "union data { int i; double d; };"  
tcc_treesitter_unions(header)  
  
## End(Not run)
```

---

tcc_union	<i>Declare union for FFI helper generation</i>
-----------	--

---

**Description**

Generate R-callable helpers for union allocation and member access. The union must be defined in a header.

**Usage**

```
tcc_union(ffl, name, members, active = NULL)
```

**Arguments**

ffi	A tcc_ffi object
name	Union name (as defined in C header)
members	Named list of union members with FFI types
active	Default active member for accessors

**Value**

Updated tcc\_ffi object

**Examples**

```
## Not run:
ffi <- tcc_ffi() |>
  tcc_union("data_variant",
    members = list(as_int = "i32", as_float = "f32"),
    active = "as_int"
  )

## End(Not run)
```

---

tcc_write_bytes	<i>Write raw bytes to a pointer</i>
-----------------	-------------------------------------

---

**Description**

Write a raw vector into memory pointed to by an external pointer.

**Usage**

```
tcc_write_bytes(ptr, raw)
```

**Arguments**

ptr	External pointer
raw	Raw vector to write

**Value**

NULL.

---

tcc_write_f32	<i>Write a 32-bit float</i>
---------------	-----------------------------

---

**Description**

Write a 32-bit float

**Usage**

```
tcc_write_f32(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one 32-bit floating-point value into native memory at ptr + offset.

---

tcc_write_f64	<i>Write a 64-bit double</i>
---------------	------------------------------

---

**Description**

Write a 64-bit double

**Usage**

```
tcc_write_f64(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one 64-bit floating-point value into native memory at ptr + offset.

---

tcc_write_i16	<i>Write a signed 16-bit integer</i>
---------------	--------------------------------------

---

**Description**

Write a signed 16-bit integer

**Usage**

```
tcc_write_i16(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one signed 16-bit value into native memory at ptr + offset.

---

tcc_write_i32	<i>Write a signed 32-bit integer</i>
---------------	--------------------------------------

---

**Description**

Write a signed 32-bit integer

**Usage**

```
tcc_write_i32(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one signed 32-bit value into native memory at ptr + offset.

tcc\_write\_i64      *Write a signed 64-bit integer*

---

**Description**

Write a signed 64-bit integer

**Usage**

```
tcc_write_i64(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one signed 64-bit value into native memory at ptr + offset.

---

tcc\_write\_i8      *Write a signed 8-bit integer*

---

**Description**

Write a signed 8-bit integer

**Usage**

```
tcc_write_i8(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly).

---

tcc_write_ptr	<i>Write a pointer at byte offset</i>
---------------	---------------------------------------

---

**Description**

Write a pointer at byte offset

**Usage**

```
tcc_write_ptr(ptr, offset, value)
```

**Arguments**

ptr	External pointer (destination buffer)
offset	Byte offset
value	External pointer to write

**Value**

NULL (invisibly).

---

tcc_write_u16	<i>Write an unsigned 16-bit integer</i>
---------------	---

---

**Description**

Write an unsigned 16-bit integer

**Usage**

```
tcc_write_u16(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one unsigned 16-bit value into native memory at ptr + offset.

---

tcc_write_u32	<i>Write an unsigned 32-bit integer</i>
---------------	---

---

**Description**

Write an unsigned 32-bit integer

**Usage**

```
tcc_write_u32(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one unsigned 32-bit value into native memory at ptr + offset.

---

tcc_write_u64	<i>Write an unsigned 64-bit integer</i>
---------------	---

---

**Description**

Write an unsigned 64-bit integer

**Usage**

```
tcc_write_u64(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one unsigned 64-bit value into native memory at ptr + offset.

---

tcc_write_u8	<i>Write an unsigned 8-bit integer</i>
--------------	--

---

**Description**

Write an unsigned 8-bit integer

**Usage**

```
tcc_write_u8(ptr, offset, value)
```

**Arguments**

ptr	External pointer
offset	Byte offset
value	Integer value to write

**Value**

NULL (invisibly). Called for its side effect of writing one unsigned 8-bit value into native memory at ptr + offset.

# Index

`$.tcc_compiled`, 4  
`as.character.tcc_cstring`, 5  
`blas_lapack_info`, 5  
`generate_trampoline`, 6  
`get_external_ptr_addr`, 6  
`is_callback_type`, 7  
`parse_callback_type`, 7  
`print.tcc_callback`, 8  
`print.tcc_compiled`, 8  
`print.tcc_cstring`, 9  
`print.tcc_ffi`, 9  
`tcc_add_file`, 10  
`tcc_add_include_path`, 10  
`tcc_add_library`, 11  
`tcc_add_library_path`, 11  
`tcc_add_symbol`, 12  
`tcc_add_sysinclude_path`, 12  
`tcc_bin_path(tcc_prefix)`, 35  
`tcc_bind`, 13  
`tcc_call_symbol`, 14  
`tcc_callback`, 15  
`tcc_callback_async_drain`, 16  
`tcc_callback_async_schedule`, 16  
`tcc_callback_close`, 17  
`tcc_callback_ptr`, 17  
`tcc_callback_valid`, 18  
`tcc_cli(tcc_prefix)`, 35  
`tcc_compile`, 18  
`tcc_compile_string`, 19  
`tcc_container_of`, 19  
`tcc_cstring`, 20  
`tcc_cstring()`, 23, 37  
`tcc_cstring_object`, 20  
`tcc_data_ptr`, 21  
`tcc_data_ptr()`, 23  
`tcc_enum`, 21  
`tcc_ffi`, 22  
`tcc_ffi()`, 14  
`tcc_field_addr`, 22  
`tcc_free`, 23  
`tcc_free()`, 20, 21, 32  
`tcc_generate_bindings`, 24  
`tcc_get_symbol`, 25  
`tcc_get_symbol()`, 31  
`tcc_global`, 25  
`tcc_header`, 26  
`tcc_include`, 27  
`tcc_include_path(tcc_prefix)`, 35  
`tcc_include_paths`, 27  
`tcc_introspect`, 28  
`tcc_lib_path(tcc_prefix)`, 35  
`tcc_lib_paths(tcc_prefix)`, 35  
`tcc_library`, 28  
`tcc_library_path`, 29  
`tcc_link`, 29  
`tcc_list_symbols`, 31  
`tcc_malloc`, 31  
`tcc_malloc()`, 23, 37  
`tcc_map_c_type_to_ffi`, 32  
`tcc_null_ptr`, 33  
`tcc_options`, 33  
`tcc_output`, 34  
`tcc_path`, 34  
`tcc_prefix`, 35  
`tcc_ptr_addr`, 35  
`tcc_ptr_free_set_null`, 36  
`tcc_ptr_is_null`, 36  
`tcc_ptr_is_owned`, 37  
`tcc_ptr_set`, 37  
`tcc_ptr_utils`, 38  
`tcc_read_bytes`, 38  
`tcc_read_cstring`, 39  
`tcc_read_f32`, 39  
`tcc_read_f64`, 40

tcc\_read\_i16, [40](#)  
tcc\_read\_i32, [41](#)  
tcc\_read\_i64, [41](#)  
tcc\_read\_i8, [42](#)  
tcc\_read\_ptr, [42](#)  
tcc\_read\_u16, [43](#)  
tcc\_read\_u32, [43](#)  
tcc\_read\_u64, [44](#)  
tcc\_read\_u8, [44](#)  
tcc\_recompile, [45](#)  
tcc\_relocate, [45](#)  
tcc\_relocate(), [31](#)  
tcc\_run\_cli, [46](#)  
tcc\_set\_options, [46](#)  
tcc\_source, [47](#)  
tcc\_state, [47](#)  
tcc\_struct, [48](#)  
tcc\_struct\_raw\_access, [48](#)  
tcc\_symbol\_is\_valid, [49](#)  
tcc\_sysinclude\_paths  
    (tcc\_include\_paths), [27](#)  
tcc\_treesitter\_bindings, [49](#)  
tcc\_treesitter\_defines, [50](#)  
tcc\_treesitter\_enum\_bindings, [51](#)  
tcc\_treesitter\_enum\_members, [52](#)  
tcc\_treesitter\_enums, [52](#)  
tcc\_treesitter\_functions, [53](#)  
tcc\_treesitter\_global\_types, [54](#)  
tcc\_treesitter\_globals, [54](#)  
tcc\_treesitter\_struct\_accessors, [55](#)  
tcc\_treesitter\_struct\_bindings, [56](#)  
tcc\_treesitter\_struct\_members, [56](#)  
tcc\_treesitter\_structs, [57](#)  
tcc\_treesitter\_union\_accessors, [58](#)  
tcc\_treesitter\_union\_bindings, [59](#)  
tcc\_treesitter\_union\_members, [59](#)  
tcc\_treesitter\_unions, [60](#)  
tcc\_union, [61](#)  
tcc\_write\_bytes, [61](#)  
tcc\_write\_f32, [62](#)  
tcc\_write\_f64, [62](#)  
tcc\_write\_i16, [63](#)  
tcc\_write\_i32, [63](#)  
tcc\_write\_i64, [64](#)  
tcc\_write\_i8, [64](#)  
tcc\_write\_ptr, [65](#)  
tcc\_write\_u16, [65](#)  
tcc\_write\_u32, [66](#)  
tcc\_write\_u64, [66](#)  
tcc\_write\_u8, [67](#)