Package: RSimpleFFI (via r-universe)

May 11, 2026

Version 1.2.0.90000

Title Simple Foreign Function Interface using 'S7' and 'libffi'

Description Simple Foreign Function Interface for 'R' using 'libffi' and 'S7' classes. Supports calling 'C' functions with type conversion and struct handling. Includes standard 'C' types (int8, int16, int32, int64, uint variants), platform types (size_t, bool), floating point types, and complex struct types. Header parsing uses 'Rtinycc' for 'TinyCC'-backed preprocessing and enables automatic generation of 'R' bindings from 'C' header files, simplifying package development for 'C' libraries.

License GPL-3

Depends R (>= 4.4.0)

Imports S7, Rtinycc, treesitter, treesitter.c

Suggests tinytest, bench

Remotes sounkou-bioinfo/Rtinycc, sounkou-bioinfo/treesitter.c

SystemRequirements GNU make, pkg-config (for libffi configuration on RTools)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

URL https://github.com/sounkou-bioinfo/RSimpleFFI, https://sounkou-bioinfo.github.io/RSimpleFFI/

BugReports https://github.com/sounkou-bioinfo/RSimpleFFI/issues

Config/pak/sysreqs make pkg-config

Repository https://sounkou-bioinfo.r-universe.dev

Date/Publication 2026-05-11 16:17:44 UTC

RemoteUrl https://github.com/sounkou-bioinfo/RSimpleFFI

RemoteRef HEAD

RemoteSha 668d9dd3877495502e4bc1469e01e1d61509049f

Contents

`ArrayType` *FFI Array Type*

Description

FFI Array Type

Usage

```
ArrayType(
  name = character(0),
  size = integer(0),
  ref = NULL,
  element_type = FFIType(),
  length = integer(0)
)
```

Arguments

name	Character name of the type
size	Integer size in bytes
ref	External pointer to ffi_type
element_type	FFIType of array elements
length	Integer length of array

Value

An ArrayType object

---

bindgen_r_api *Generate FFI bindings for R's C API*

Description

Parses R's header files (Rinternals.h, R.h, Rmath.h) and generates R bindings that allow calling R's internal C functions directly via FFI.

Usage

```
bindgen_r_api(
  output_file = NULL,
  headers = c("Rinternals.h", "R.h", "Rmath.h"),
  include_path = R.home("include"),
  load_r_lib = .is_windows,
  verbose = FALSE
)
```

Arguments

output_file	Path to write the generated R bindings. If NULL, returns the parsed results without writing.
headers	Character vector of header names to parse. Default includes "Rinternals.h", "R.h", and "Rmath.h".

include_path	Path to R's include directory. Defaults to R.home("include").
load_r_lib	Logical. If TRUE (default on Windows), attempt to load the R shared library to ensure symbols are available.
verbose	Logical. If TRUE, print progress messages.

Details

On Unix-like systems (Linux, macOS), R's symbols are typically available in the running process. On Windows, the R.dll may need to be explicitly loaded. Set load_r_lib = TRUE to handle this automatically.

The generated bindings create wrapper functions that use ffi_function() to call the underlying C functions. Each wrapper includes roxygen2 documentation with parameter types and return values.

Value

Invisibly returns a list with parsed results for each header:

structs	Named list of struct definitions
functions	Named list of function signatures
typedefs	Named list of typedef mappings
enums	Named list of enum definitions

Available Headers

Rinternals.h Core R internals: SEXP manipulation, memory management, type checking functions (Rf_isInteger, Rf_length, etc.)

R.h Main R header, includes standard utilities

Rmath.h Statistical distribution functions (dnorm, pnorm, qnorm, gamma, beta, etc.)

See Also

ffi_parse_header(), generate_r_bindings(), ffi_function()

Examples

```
## Not run:
# Generate bindings to a file
bindgen_r_api(output_file = "r_api_bindings.R")

# Parse without writing (for inspection)
result <- bindgen_r_api(verbose = TRUE)
names(result$Rinternals$functions)

# Generate only Rmath bindings
bindgen_r_api(
    output_file = "rmath_bindings.R",
    headers = "Rmath.h"
)
```

```
# After sourcing generated bindings:
# source("r_api_bindings.R")
# r_Rf_dnorm4(0, 0, 1, 0L)  # same as dnorm(0, 0, 1)

## End(Not run)
```

`bindgen_r_api_summary` *Get summary of R API bindings*

Description

Quick summary of what's available in R's C API headers, including all headers in the main include directory and the R_ext subdirectory.

Usage

```
bindgen_r_api_summary(include_path = R.home("include"))
```

Arguments

`include_path`	Path to R's include directory

Value

Data frame with header info including name, exists flag, size, and category

Examples

```
## Not run:
bindgen_r_api_summary()

## End(Not run)
```

`CIF` *FFI Call Interface (CIF)*

Description

FFI Call Interface (CIF)

Usage

```
CIF(return_type = FFIType(), arg_types = list(), ref = NULL)
```

Arguments

return_type	FFIType for return value
arg_types	List of FFIType objects for arguments
ref	External pointer to ffi_cif

Value

An CIF object

---

create_builtin_type *Create built-in FFI type*

Description

Create built-in FFI type

create_builtin_type

Usage

```
create_builtin_type(name, ...)
```

Arguments

name	Character name of built-in type
...	Additional arguments (not used)

Value

An FFIType object

FFIType object for bool

---

data_ptr *Get data pointer from R vector with GC protection*

Description

Returns an external pointer to the underlying data (INTEGER, REAL, etc.) while ensuring the object won't be garbage collected.

Usage

```
data_ptr(x)
```

Arguments

x	An R vector (integer, double, complex, character, raw, or list)

Details

Note: For ALTREP objects (like `1:10`), use `data_ptr_ro()` instead, which properly handles deferred materialization.

Value

External pointer to the data, with finalizer to release protection

Examples

```
## Not run:
x <- c(1.0, 2.0, 3.0) # Regular vector, not ALTREP
ptr <- data_ptr(x)
# ptr points to the double* array
# Safe to pass to C functions expecting double*

## End(Not run)
```

`data_ptr_ro` *Get read-only data pointer from R vector with GC protection*

Description

Like `data_ptr()` but attempts read-only access first (for ALTREP support). If the ALTREP implementation doesn't provide direct access, falls back to materializing the data (which may allocate memory).

Usage

```
data_ptr_ro(x)
```

Arguments

x	An R vector

Value

External pointer to the data

dll_ffi_symbol *Create FFI function from dynamically loaded function*

Description

This creates an FFI function wrapper for dynamically loaded native C functions. Uses direct address access like Rffi for maximum compatibility.

Usage

```
dll_ffi_symbol(symbol_name, return_type, ..., package = NULL, na_check = TRUE)
```

Arguments

symbol_name	Name of the symbol
return_type	Return type specifimessageion
...	Argument type specifimessageions
package	Package name (optional)
na_check	Logical; if TRUE (default), check for NA values and error if found. Set to FALSE to skip NA checking for better performance (at your own risk).

Value

FFI function object that can be called directly

dll_info *Get information about a loaded library*

Description

Get information about a loaded library

Usage

```
dll_info(handle)
```

Arguments

handle	Library handle (path)

Value

List with library information

dll_is_loaded *Check if a symbol is loaded*

Description

Check if a symbol is loaded

Usage

```
dll_is_loaded(symbol_name, package = NULL)
```

Arguments

symbol_name	Name of the symbol to check
package	Package name (optional)

Value

TRUE if symbol is loaded, FALSE otherwise

dll_list_loaded *List loaded libraries*

Description

List loaded libraries

Usage

```
dll_list_loaded()
```

Value

Character vector of loaded library paths

`dll_load` *Load a shared library/DLL*

Description

This function compiles C code using R's configured compiler and loads it. Uses the same compiler configuration that R was built with.

Usage

```
dll_load(filename, now = TRUE, local = TRUE, verbose = FALSE)

dll_unload(handle, verbose = FALSE)

dll_symbol(symbol_name, package = NULL)

dll_compile_and_load(
  code,
  name = "temp_dll",
  includes = NULL,
  libs = NULL,
  verbose = FALSE,
  cflags = NULL,
  compilation_directory = tempfile("dll_compile_")
)

dll_load_system(lib_name, verbose = FALSE)

dll_load_r(verbose = FALSE)
```

Arguments

`filename`	Path to the shared library
`now`	Whether to resolve all symbols immediately (default TRUE)
`local`	Keep symbols local to avoid namespace pollution (default TRUE)
`verbose`	Print loading information (default FALSE)
`handle`	Library handle (path) returned by dll_load()
`symbol_name`	Name of the symbol to find
`package`	Package name where symbol is registered (optional)
`code`	Character vector of C code to compile
`name`	Base name for the compiled library (default "temp_dll")
`includes`	Additional include directories
`libs`	Additional libraries to link
`cflags`	Additional compiler flags (e.g., "-O2", "-O3")

`compilation_directory`
: Directory to use for compilation (default temp dir)

`lib_name`
: Name of system library (e.g., libc.so.6, libm.dylib, kernel32.dll)

Value

Library handle (character string of loaded library path)

dyn.unload result invisibly

Symbol information including address as external pointer

Library handle that can be used with dll_* functions

Library handle or NULL if not found

Library handle or NULL if not found

EnumType *FFI Enumeration Type*

Description

FFI Enumeration Type

Usage

```
EnumType(
  name = character(0),
  size = integer(0),
  ref = NULL,
  values = integer(0),
  underlying_type = FFIType()
)
```

Arguments

`name`
: Character name of the type

`size`
: Integer size in bytes

`ref`
: External pointer to ffi_type

`values`
: Named integer vector of enum values

`underlying_type`
: FFIType for the underlying integer type

escape_r_name *Escape R name with backticks if needed*

Description

Escape R name with backticks if needed

Usage

```
escape_r_name(name)
```

Arguments

name	Variable name to escape

Value

Escaped name if needed, original otherwise

ffi_all_offsets *Get all field offsets for a struct*

Description

Returns a named integer vector with byte offsets for all fields. For packed structs, uses the pack alignment setting.

Usage

```
ffi_all_offsets(struct_type, use_pack = TRUE)
```

Arguments

struct_type	StructType object
use_pack	Logical, whether to use the struct's pack setting. Default TRUE.

Value

Named integer vector of offsets

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_double())
ffi_all_offsets(Point)
# x y
# 0 8

# Packed struct
Packed <- ffi_struct(a = ffi_uint8(), b = ffi_int32(), .pack = 1)
ffi_all_offsets(Packed)
# a b
# 0 1

## End(Not run)
```

---

ffi_alloc *Allocate a buffer for a given FFI type*

Description

Allocates a buffer for n elements of the given FFIType (e.g., int, double, etc). Returns an external pointer tagged with the type.

Usage

```
ffi_alloc(type, ...)
```

Arguments

type	FFIType object
...	Additional arguments

Value

External pointer to buffer

ffi_alloc_buffer *Allocate a raw memory buffer (external pointer, auto-finalized)*

Description

Allocates a buffer of the given size (in bytes) and returns an external pointer. The memory is automatically freed when the pointer is garbage collected.

Usage

```
ffi_alloc_buffer(size)
```

Arguments

size	Number of bytes to allocate

Value

External pointer to buffer

ffi_array_type *Create an FFI array type*

Description

Create an FFI array type

Usage

```
ffi_array_type(element_type, length)
```

Arguments

element_type	FFIType of array elements
length	Integer length of array

Value

An ArrayType object

ffi_bool *Bool FFI type*

Description

Bool FFI type

Usage

```
ffi_bool()
```

Value

FFIType object for bool

ffi_call *Make FFI function call*

Description

Call a C function through the FFI interface.

Usage

```
ffi_call(cif, symbol, ...)
```

Arguments

cif	CIF object defining the call interface
symbol	NativeSymbol or character name of function
...	Arguments to pass to the function (including na_check)

Details

The method implementations accept an additional na_check argument (logical, default TRUE). When TRUE, the function checks for NA values in arguments and errors if found. Set to FALSE to skip NA checking for better performance (at your own risk).

Error Handling Limitations:

Important: libffi provides no error handling for the actual C function call. If the called C function crashes (segmentation fault, illegal instruction, abort, etc.), R itself will crash. This is a fundamental limitation of FFI

- there is no portable way to catch such errors in C code.

Before making FFI calls, ensure:

- The function pointer is valid (not NULL, points to executable code)
- All pointer arguments are valid (use `ffi_is_null` to check)
- Array/buffer sizes are correct - buffer overruns cause undefined behavior
- The CIF signature exactly matches the C function's signature
- Struct layouts match between R types and C (check alignment/padding)

For debugging crashes:

- Run R under a debugger: `R -d gdb`
- Enable core dumps: `ulimit -c unlimited`
- Use address sanitizers when building the library being called

Value

The return value from the C function, converted to an R type

See Also

`ffi_is_null` for checking pointer validity

`ffi_char` *Char FFI type*

Description

Char FFI type

Usage

```
ffi_char()
```

Value

FFIType char type

`ffi_cif` *Prepare FFI call interface Prepare FFI call interface*

Description

Prepare FFI call interface Prepare FFI call interface

Usage

```
ffi_cif(return_type, ...)
```

Arguments

`return_type`	FFIType for return value
`...`	FFIType objects for arguments

`ffi_cif_var` *Prepare FFI call interface for variadic functions*

Description

Creates a CIF for calling C functions with variable arguments (varargs). Unlike regular CIFs, variadic CIFs must specify the types of ALL arguments for each specific call, including the variadic ones.

Usage

```
ffi_cif_var(return_type, nfixedargs, ...)
```

Arguments

`return_type`	FFIType for return value
`nfixedargs`	Number of fixed arguments (before the ...)
`...`	FFIType objects for ALL arguments (fixed + variadic)

Details

Due to C calling conventions, variadic arguments undergo "default argument promotions": float becomes double, and small integers (char, short) become int. You must use ffi_int() or ffi_double() for variadic arguments, not smaller types.

Value

CIF object

Examples

```
## Not run:
# Call a varargs function: test_varargs_sum(int nargs, ...)
# First argument (nargs) is fixed, rest are variadic integers
sym <- ffi_symbol("test_varargs_sum")

# Call with 3 variadic int arguments
cif <- ffi_cif_var(ffi_double(),
  nfixedargs = 1L,
  ffi_int(), ffi_int(), ffi_int(), ffi_int()
)
result <- ffi_call(cif, sym, 3L, 10L, 20L, 30L) # returns 60

## End(Not run)
```

`ffi_closure` *Create an FFI closure from an R function*

Description

Wraps an R function so it can be used as a callback from C code. The closure has a CIF that describes its signature (return type and argument types). When C code calls through the closure's function pointer, the R function is invoked with converted arguments.

Usage

```
ffi_closure(r_function, return_type, ...)
```

Arguments

`r_function`	An R function to wrap as a callback
`return_type`	FFIType for return value
`...`	FFIType objects for arguments

Details

The R function must accept the same number of arguments as specified in the type signature. Arguments are converted from C types to R types before calling, and the return value is converted back to C.

Important: You must keep a reference to the FFIClosure object for as long as C code might call through it. If the closure is garbage collected, calling through its function pointer will crash.

Value

An FFIClosure object

See Also

ffi_closure_pointer() to get the callable function pointer

Examples

```
## Not run:
# Create a comparison function for qsort
cmp_fn <- function(a, b) {
  as.integer(a - b)
}

# Wrap it as a C callback: int (*)(int*, int*)
cmp_closure <- ffi_closure(
  cmp_fn,
  ffi_int(), # return type
  ffi_pointer(), ffi_pointer() # argument types (pointers to int)
```

```
)

# Get the function pointer to pass to C
cmp_ptr <- ffi_closure_pointer(cmp_closure)

## End(Not run)
```

`ffi_closure_pointer` *Get the function pointer for an FFI closure*

Description

Returns the executable function pointer that can be passed to C functions expecting a callback.

Usage

```
ffi_closure_pointer(closure)
```

Arguments

`closure`	An FFIClosure object

Details

The returned pointer can be passed to C functions via `ffi_call()`. It will invoke the R function when called.

Value

External pointer to the callable function

See Also

ffi_closure() to create closures

ffi_closures_supported *Check if closures are supported on this platform*

Description

Not all platforms support FFI closures. Use this function to check before attempting to create closures.

Usage

```
ffi_closures_supported()
```

Value

Logical; TRUE if closures are supported

ffi_copy_array *Copy array from native memory*

Description

Copy array from native memory

Usage

```
ffi_copy_array(ptr, length, element_type)
```

Arguments

ptr	External pointer to array
length	Integer length of array
element_type	FFIType of array elements

ffi_copy_array_type *Copy array from native memory (ArrayType version)*

Description

Copy array from native memory (ArrayType version)

Usage

```
ffi_copy_array_type(ptr, array_type)
```

Arguments

ptr	External pointer to array
array_type	ArrayType object

ffi_create_bitfield_accessors *Create accessor functions for a bit-field structure*

Description

Generates getter and setter functions for a C structure with bit-fields, allowing easy manipulation of packed bit-field values.

Usage

```
ffi_create_bitfield_accessors(field_specs, base_type = ffi_uint32())
```

Arguments

field_specs	Named list where names are field names and values are bit widths (integers)
base_type	FFI type for the packed representation (default: ffi_uint32())

Value

List with pack, unpack, get, and set functions

Examples

```
# Define a bit-field structure
# C equivalent:
# struct Flags {
#   unsigned int enabled : 1;
#   unsigned int mode : 3;
#   unsigned int priority : 4;
# };

flags_accessors <- ffi_create_bitfield_accessors(
  list(enabled = 1L, mode = 3L, priority = 4L)
)

# Pack values
packed <- flags_accessors$pack(list(enabled = 1L, mode = 5L, priority = 12L))

# Unpack to list
flags_accessors$unpack(packed)
# $enabled:  1
# $mode:     5
# $priority: 12

# Get a single field
flags_accessors$get(packed, "mode") # 5

# Set a single field
new_packed <- flags_accessors$set(packed, "mode", 7L)
flags_accessors$get(new_packed, "mode") # 7
```

---

ffi_create_helpers *Create High-Level API Helpers for Struct*

Description

This is the main user-facing function for API mode. It generates C code to compute field offsets using the compiler, compiles it, and returns a helper object with constructor and field metadata.

Usage

```
ffi_create_helpers(struct_name, field_types, include_dirs = NULL)
```

Arguments

struct_name	Character string naming the struct (e.g., "Point2D")
field_types	Named list of FFIType objects for each field (e.g., list(x = ffi_int(), y = ffi_int()))
include_dirs	Optional character vector of include directories for compilation

Value

An rffi_struct_helpers S3 object with:

$new() Constructor function that allocates a new struct

$fields Named list of field metadata (offset + FFIType)

$get(ptr, field) Get field value from struct pointer

$set(ptr, field, value) Set field value in struct pointer

$lib The compiled library object (for cleanup)

Examples

```
## Not run:
# Define struct with field types
helpers <- ffi_create_helpers(
  "Point2D",
  list(x = ffi_int(), y = ffi_int())
)

# Create instance
pt <- helpers$new()

# Set/get fields
helpers$set(pt, "x", 42L)
helpers$set(pt, "y", 100L)
helpers$get(pt, "x")  # 42L
helpers$get(pt, "y")  # 100L

# Access field metadata
helpers$fields$x$offset  # 0
helpers$fields$y$offset  # 4

## End(Not run)
```

---

ffi_deref_pointer *Dereference a pointer*

Description

Reads the pointer value stored at an address. This is useful for accessing global variables in shared libraries that are pointers (like R_GlobalEnv, R_NilValue, etc.).

Usage

```
ffi_deref_pointer(ptr)
```

Arguments

ptr	External pointer to the address to dereference

Value

External pointer containing the value at the address

Examples

```
## Not run:
# Get R_GlobalEnv from libR.so
addr <- getNativeSymbolInfo("R_GlobalEnv")$address
globalenv_sexp <- ffi_deref_pointer(addr)

## End(Not run)
```

---

ffi_double *double FFI type*

Description

double FFI type

Usage

```
ffi_double()
```

Value

FFIType object for double

---

ffi_enum *Create FFI enumeration type*

Description

Create FFI enumeration type

Usage

```
ffi_enum(..., underlying_type = ffi_int())
```

Arguments

`...`	Named integer values representing enum constants
`underlying_type`	FFIType for underlying integer type (default: ffi_int())

Value

EnumType object

ffi_enum_to_int *Convert enum name to integer value*

Description

Look up the integer value for a named enum constant.

Usage

```
ffi_enum_to_int(enum_type, name)
```

Arguments

enum_type	EnumType object
name	Character name of enum constant

Value

Integer value

Examples

```
## Not run:
Color <- ffi_enum(RED = 0L, GREEN = 1L, BLUE = 2L)
ffi_enum_to_int(Color, "GREEN") # 1L

## End(Not run)
```

ffi_extract_bit_field *Extract a single bit-field from a packed value*

Description

Extracts a single bit-field value from a packed integer at a specified bit offset and width.

Usage

```
ffi_extract_bit_field(packed_value, bit_offset, bit_width)
```

Arguments

packed_value	Integer value containing packed bit-fields
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Extracted integer value

Examples

```
# Extract 3-bit mode field at bit offset 1 from value 0x65
ffi_extract_bit_field(0x65L, 1L, 3L) # 5

# Extract 4-bit priority field at bit offset 4
ffi_extract_bit_field(0x65L, 4L, 4L) # 6
```

---

ffi_extract_bits64 *Extract a single bit-field from a 64-bit packed value*

Description

Extract a single bit-field from a 64-bit packed value

Usage

```
ffi_extract_bits64(packed_value, bit_offset, bit_width)
```

Arguments

packed_value	Packed value (as double for 64-bit range)
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Extracted value as double (for 64-bit range)

Examples

```
packed <- ffi_pack_bits64(c(1L, 5L, 12L), c(1L, 3L, 4L))
ffi_extract_bits64(packed, 1L, 3L) # 5 (mode field)
```

ffi_extract_signed_bit_field

Extract a signed bit-field from a packed value

Description

Extracts a single bit-field and sign-extends it based on the high bit. This is the 32-bit version using pure R code.

Usage

```
ffi_extract_signed_bit_field(packed_value, bit_offset, bit_width)
```

Arguments

packed_value	Integer value containing packed bit-fields
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Extracted signed integer value

Examples

```
# A 4-bit value of 13 (0xD) represents -3 in signed 4-bit
packed <- ffi_pack_bits(c(13L), c(4L))
ffi_extract_signed_bit_field(packed, 0L, 4L) # -3

# 3-bit value of 7 represents -1 in signed 3-bit
packed2 <- ffi_pack_bits(c(7L), c(3L))
ffi_extract_signed_bit_field(packed2, 0L, 3L) # -1
```

ffi_extract_signed_bits64

Extract a signed bit-field from a 64-bit packed value

Description

Extracts a bit-field and sign-extends it based on the high bit. Useful for signed integer fields in C structures.

Usage

```
ffi_extract_signed_bits64(packed_value, bit_offset, bit_width)
```

Arguments

packed_value	Packed value (as double for 64-bit range)
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Extracted signed value as double

Examples

```
# Pack a negative value in 4-bit signed field (-3 = 0xD in 4 bits)
packed <- ffi_pack_bits64(c(13L), c(4L)) # 0xD = -3 as signed 4-bit
ffi_extract_signed_bits64(packed, 0L, 4L) # -3
```

---

ffi_field_info *Get field information from a struct type*

Description

Returns metadata about a specific field, including its byte offset, size, and type information.

Usage

```
ffi_field_info(struct_type, field)
```

Arguments

struct_type	StructType object
field	Character field name or integer field index (1-based)

Value

FieldInfo object with offset, size, alignment info

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_double())
ffi_field_info(Point, "x")
# <FieldInfo 'x' type=int, offset=0, size=4>
ffi_field_info(Point, "y")
# <FieldInfo 'y' type=double, offset=8, size=8>  (offset 8 due to alignment)

## End(Not run)
```

ffi_fill_typed_buffer *Fill a typed buffer from an R vector (int or double)*

Description

Fill a typed buffer from an R vector (int or double)

Usage

```
ffi_fill_typed_buffer(ptr, values, type)
```

Arguments

ptr	External pointer to buffer
values	Integer or double vector
type	FFIType object

ffi_float *float FFI type*

Description

float FFI type

Usage

```
ffi_float()
```

Value

FFIType object for float

`ffi_free` *Free memory pointed to by an external pointer*

Description

Explicitly frees memory that was allocated by C code and returned as a pointer. Use this when you know the pointer was allocated with malloc/calloc and it's your responsibility to free it.

Usage

```
ffi_free(ptr)
```

Arguments

`ptr`	External pointer to free

Details

When to use this:

- Pointers returned from C functions that allocate memory (e.g., `strdup`, `malloc`)
- When C documentation says "caller must free"

When NOT to use this:

- Pointers allocated via `ffi_alloc()` (auto-freed by R's GC)
- Static or global pointers from C
- Pointers into existing structures
- Pointers that C will free itself

Calling `ffi_free()` on an already-freed pointer or invalid pointer will cause undefined behavior (likely crash).

Value

NULL invisibly

Examples

```
## Not run:
# C function that allocates and returns a string
strdup_fn <- ffi_function("strdup", ffi_pointer(), ffi_string())
ptr <- strdup_fn("hello")
# ... use ptr ...
ffi_free(ptr) # We must free because strdup allocates

## End(Not run)
```

ffi_function *Create a reusable FFI function wrapper*

Description

Create a reusable FFI function wrapper

Usage

```
ffi_function(name, return_type, ..., library = NULL, na_check = TRUE)
```

Arguments

name	Character name of the function
return_type	FFIType for return value
...	FFIType objects for arguments
library	Character name of library (optional)
na_check	Logical; if TRUE (default), check for NA values and error if found. Set to FALSE to skip NA checking for better performance (at your own risk).

ffi_get_element *Get element from struct array*

Description

Returns a pointer to the i-th struct in a contiguous array of structs. The returned pointer shares memory with the original array.

Usage

```
ffi_get_element(ptr, index, struct_type)
```

Arguments

ptr	External pointer to struct array (from ffi_alloc with n > 1)
index	1-based index of element to get
struct_type	StructType describing the element type

Value

External pointer to the element (no finalizer - parent owns memory)

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_int())
points <- ffi_alloc(Point, 10L) # array of 10 Points
p3 <- ffi_get_element(points, 3L, Point)
ffi_set_field(p3, "x", 100L, Point)

## End(Not run)
```

---

ffi_get_field *Get field value from FFI structure*

Description

Get field value from FFI structure

Usage

```
ffi_get_field(ptr, field, struct_type, ...)
```

Arguments

ptr	External pointer to structure
field	Character field name or integer field index
struct_type	StructType object
...	Not used; required for S7 generic dispatch.

Value

Field value

---

ffi_int *int FFI type*

Description

int FFI type

Usage

```
ffi_int()
```

Value

FFIType object for int

ffi_int_to_enum *Convert integer value to enum name*

Description

Look up the enum constant name for an integer value.

Usage

```
ffi_int_to_enum(enum_type, value)
```

Arguments

enum_type	EnumType object
value	Integer value

Value

Character name of enum constant (or NA if not found)

Examples

```
## Not run:
Color <- ffi_enum(RED = 0L, GREEN = 1L, BLUE = 2L)
ffi_int_to_enum(Color, 1L) # "GREEN"

## End(Not run)
```

ffi_int16 *Int16 FFI type*

Description

Int16 FFI type

Usage

```
ffi_int16()
```

Value

FFIType object for int16

ffi_int32 *Int32 FFI type*

Description

Int32 FFI type

Usage

```
ffi_int32()
```

Value

FFIType object for int32

ffi_int64 *Int64 FFI type*

Description

Int64 FFI type

Usage

```
ffi_int64()
```

Value

FFIType object for int64

ffi_int8 *Int8 FFI type*

Description

Int8 FFI type

Usage

```
ffi_int8()
```

Value

FFIType object for int8

ffi_is_null *Check if external pointer is NULL*

Description

Check if external pointer is NULL

Usage

```
ffi_is_null(ptr)
```

Arguments

ptr	External pointer

Value

Logical

ffi_loaded_libs *Get information about loaded native libraries*

Description

Get information about loaded native libraries

Usage

```
ffi_loaded_libs()
```

ffi_long *long FFI type*

Description

long FFI type

Usage

```
ffi_long()
```

Value

FFIType object for long

ffi_longdouble *longdouble FFI type*

Description

longdouble FFI type

Usage

```
ffi_longdouble()
```

Value

FFIType object for longdouble

ffi_longlong *long long FFI type*

Description

long long FFI type

Usage

```
ffi_longlong()
```

Value

FFIType object for longlong

ffi_null_pointer *Create a NULL pointer*

Description

Create a NULL pointer

Usage

```
ffi_null_pointer()
```

Value

External pointer to NULL

`ffi_offsetof` *Get byte offset of a field in a structure*

Description

Returns the byte offset of a field within a structure, accounting for alignment requirements. Similar to C's offsetof() macro.

Usage

```
ffi_offsetof(struct_type, field, use_pack = TRUE)
```

Arguments

`struct_type`	StructType object
`field`	Character field name or integer field index (1-based)
`use_pack`	Logical, whether to use the struct's pack setting. Default TRUE. Set to FALSE to get libffi's natural alignment offset even for packed structs.

Details

For packed structures (created with `pack` parameter), this function computes the offset using the specified packing alignment rather than natural alignment.

Value

Integer byte offset

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_double())
ffi_offsetof(Point, "x") # 0
ffi_offsetof(Point, "y") # 8 (aligned to 8-byte boundary)

# Packed struct example
Packed <- ffi_struct(a = ffi_uint8(), b = ffi_int32(), .pack = 1)
ffi_offsetof(Packed, "b") # 1 (no padding with .pack=1)

## End(Not run)
```

`ffi_pack_bits` *Pack bit-fields into an integer*

Description

Packs multiple values into a single integer according to specified bit widths. This is useful when working with C structures that use bit-fields, which are not directly supported by libffi.

Usage

```
ffi_pack_bits(values, widths, base_type = ffi_uint32())
```

Arguments

`values`	Integer vector of values to pack
`widths`	Integer vector of bit widths for each value
`base_type`	FFI type for the result (default: ffi_uint32())

Details

Values are packed from LSB to MSB (least significant bit to most significant bit). Each value is masked to its specified width and shifted into position.

Value

Packed integer value

Examples

```
# Pack three bit-fields: enabled (1 bit), mode (3 bits), priority (4 bits)
packed <- ffi_pack_bits(c(1L, 5L, 12L), c(1L, 3L, 4L))
# Result: 0b1100101 = 0x65 = 101

# Verify by unpacking
ffi_unpack_bits(packed, c(1L, 3L, 4L))
# [1]  1  5 12
```

ffi_pack_bits64 *Pack bit-fields into a 64-bit value*

Description

Packs multiple values into a single 64-bit integer (returned as double) according to specified bit widths. This version uses C code for full 64-bit support.

Usage

```
ffi_pack_bits64(values, widths)
```

Arguments

values	Integer vector of values to pack
widths	Integer vector of bit widths for each value

Details

Values are packed from LSB to MSB (least significant bit to most significant bit). Each value is masked to its specified width and shifted into position. Uses C implementation for full 64-bit support (R integers are only 32-bit).

Value

Packed value as double (for 64-bit range)

Examples

```
# Pack three bit-fields: enabled (1 bit), mode (3 bits), priority (4 bits)
packed <- ffi_pack_bits64(c(1L, 5L, 12L), c(1L, 3L, 4L))

# Works with values > 32 bits total
large_packed <- ffi_pack_bits64(c(1, 0x7FFFFFFF), c(1L, 31L))
```

ffi_parse_header *Parse C header file and create structured result*

Description

Uses tree-sitter for robust AST-based parsing of C headers.

Usage

```
ffi_parse_header(header_file, includes = NULL)
```

Arguments

header_file	Path to C header file
includes	Additional include directories

Value

List with parsed components (file, defines, structs, unions, enums, functions, typedefs)

---

ffi_pointer *pointer FFI type*

Description

pointer FFI type

Usage

```
ffi_pointer()
```

Value

FFIType object for pointer

---

ffi_print_struct *Pretty print struct contents*

Description

Pretty print struct contents

Usage

```
ffi_print_struct(ptr, struct_type)
```

Arguments

ptr	External pointer to struct
struct_type	StructType object

ffi_raw *Char FFI type*

Description

Char FFI type

Usage

```
ffi_raw()
```

Value

FFIType object for char

ffi_read_global *Read a global variable from a shared library*

Description

Reads a typed value from a global symbol address. This is a typed version of ffi_deref_pointer that handles type conversion.

Usage

```
ffi_read_global(ptr, type)
```

Arguments

ptr	External pointer to the global variable address
type	FFIType describing the type of the global variable

Value

The value at the address, converted to an appropriate R type

ffi_set_bit_field *Set a single bit-field in a packed value*

Description

Updates a single bit-field in a packed integer at a specified bit offset and width, returning the modified packed value.

Usage

```
ffi_set_bit_field(packed_value, new_value, bit_offset, bit_width)
```

Arguments

packed_value	Integer value containing packed bit-fields
new_value	New value for the bit-field
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Modified packed integer value

Examples

```
# Set 3-bit mode field at bit offset 1 to value 7
ffi_set_bit_field(0x65L, 7L, 1L, 3L) # 0x6F

# Set 1-bit enabled field at bit offset 0 to 0
ffi_set_bit_field(0x65L, 0L, 0L, 1L) # 0x64
```

ffi_set_bits64 *Set a single bit-field in a 64-bit packed value*

Description

Set a single bit-field in a 64-bit packed value

Usage

```
ffi_set_bits64(packed_value, new_value, bit_offset, bit_width)
```

Arguments

packed_value	Packed value (as double for 64-bit range)
new_value	New value for the bit-field
bit_offset	Bit offset from LSB (0-based)
bit_width	Number of bits in the field

Value

Modified packed value as double

Examples

```
packed <- ffi_pack_bits64(c(1L, 5L, 12L), c(1L, 3L, 4L))
new_packed <- ffi_set_bits64(packed, 7L, 1L, 3L) # Set mode to 7
ffi_extract_bits64(new_packed, 1L, 3L) # 7
```

---

ffi_set_field *Set field value in FFI structure*

Description

Set field value in FFI structure

Usage

```
ffi_set_field(ptr, field, value, struct_type, ...)
```

Arguments

ptr	External pointer to structure
field	Character field name or integer field index
value	Value to set
struct_type	StructType object
...	Not used; required for S7 generic dispatch.

Value

Updated pointer

ffi_short *short FFI type*

Description

short FFI type

Usage

```
ffi_short()
```

Value

FFIType object for short

ffi_size_t *Size_t FFI type*

Description

Size_t FFI type

Usage

```
ffi_size_t()
```

Value

FFIType object for size_t

ffi_sizeof *Get size of FFI type in bytes*

Description

Get size of FFI type in bytes

Usage

```
ffi_sizeof(type, ...)
```

Arguments

type	FFIType object
...	Additional arguments (not used)

Value

Size in bytes

`ffi_ssize_t` *ssize_t FFI type*

Description

ssize_t FFI type

Usage

```
ffi_ssize_t()
```

Value

FFIType object for ssize_t

`ffi_string` *String FFI type*

Description

String FFI type

Usage

```
ffi_string()
```

Value

FFIType object for string

`ffi_struct` *Create FFI structure type*

Description

Creates an FFI structure type from named field types. By default, libffi uses natural alignment (each field aligned to its size). Use the `.pack` parameter to specify tighter packing similar to `#pragma pack(n)` in C.

Usage

```
ffi_struct(..., .pack = NULL)
```

Arguments

`...`	Named FFIType objects representing struct fields
`.pack`	Integer specifying packing alignment (1, 2, 4, 8, or 16), or NULL for default/natural alignment. When .pack=1, fields are byte-aligned (no padding). The dot prefix prevents collision with C struct field names.

Value

StructType object

Packing

The `.pack` parameter affects `ffi_offsetof()`, `ffi_sizeof()`, `ffi_get_field()`, and `ffi_set_field()`.

Packed Structs By Value

Packed structs cannot be passed by value - GCC compiles functions taking `__attribute__((packed))` structs to expect arguments on the stack, but libffi passes via registers. Use pointers instead.

Examples

```
# Natural alignment (default)
Point <- ffi_struct(x = ffi_int(), y = ffi_int())

# Packed struct (1-byte alignment)
PackedData <- ffi_struct(
  flag = ffi_uint8(),
  value = ffi_int32(),
  .pack = 1
)

# Check sizes
ffi_sizeof(Point) # Natural size
ffi_sizeof(PackedData) # Packed size (smaller)
```

```
# Note: Packed structs cannot be passed by value to C functions.
# Use pointers instead:
# ffi_function("some_func", ffi_void(), ffi_pointer())
```

`ffi_struct_array_from_list` *Allocate array of structs from R list*

Description

Allocate array of structs from R list

Usage

```
ffi_struct_array_from_list(struct_type, values)
```

Arguments

`struct_type`	StructType object
`values`	List of named lists, one per struct

Value

External pointer to allocated struct array

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_int())
points <- ffi_struct_array_from_list(Point, list(
  list(x = 0L, y = 0L),
  list(x = 10L, y = 20L),
  list(x = 30L, y = 40L)
))

## End(Not run)
```

ffi_struct_from_list *Create and initialize a struct from R list*

Description

Create and initialize a struct from R list

Usage

```
ffi_struct_from_list(struct_type, values)
```

Arguments

struct_type	StructType object
values	Named list of field values

Value

External pointer to allocated and initialized struct

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_int())
pt <- ffi_struct_from_list(Point, list(x = 10L, y = 20L))

## End(Not run)
```

ffi_struct_to_list *Convert struct to R list*

Description

Convert struct to R list

Usage

```
ffi_struct_to_list(ptr, struct_type)
```

Arguments

ptr	External pointer to struct
struct_type	StructType object

Value

Named list of field values

Examples

```
## Not run:
Point <- ffi_struct(x = ffi_int(), y = ffi_int())
pt <- ffi_alloc(Point)
ffi_set_field(pt, "x", 42L, Point)
ffi_set_field(pt, "y", 100L, Point)
as.list(pt, Point) # list(x = 42L, y = 100L)

## End(Not run)
```

---

ffi_symbol *Get native symbol reference*

Description

Get native symbol reference

Usage

```
ffi_symbol(name, library = NULL)
```

Arguments

name	Character name of the symbol
library	Character name of library (optional)

---

ffi_symbol_from_address *Create native symbol from direct address*

Description

Create native symbol from direct address

Usage

```
ffi_symbol_from_address(address, name = "anonymous")
```

Arguments

address	External pointer to the symbol address
name	Character name of the symbol (for reference)

ffi_uchar *Uchar FFI type*

Description

Uchar FFI type

Usage

```
ffi_uchar()
```

Value

FFIType object for uchar

ffi_uint *uint FFI type*

Description

uint FFI type

Usage

```
ffi_uint()
```

Value

FFIType object for uint

ffi_uint16 *Uint16 FFI type*

Description

Uint16 FFI type

Usage

```
ffi_uint16()
```

Value

FFIType object for uint16

ffi_uint32 *Uint32 FFI type*

Description

Uint32 FFI type

Usage

```
ffi_uint32()
```

Value

FFIType object for uint32

ffi_uint64 *Uint64 FFI type*

Description

Uint64 FFI type

Usage

```
ffi_uint64()
```

Value

FFIType object for uint64

ffi_uint8 *Uint8 FFI type*

Description

Uint8 FFI type

Usage

```
ffi_uint8()
```

Value

FFIType object for uint8

ffi_ulong *ulong FFI type*

Description

ulong FFI type

Usage

```
ffi_ulong()
```

Value

FFIType object for ulong

ffi_ulonglong *ulonglong FFI type*

Description

ulonglong FFI type

Usage

```
ffi_ulonglong()
```

Value

FFIType object for ulonglong

ffi_union *Create FFI union type*

Description

Creates a union type where all fields share the same memory location. The union's size is the size of its largest member.

Usage

```
ffi_union(..., .pack = NULL)
```

Arguments

...	Named FFIType objects representing union fields
.pack	Integer packing alignment (1, 2, 4, 8, or 16). When specified, the union's alignment is reduced to min(natural_alignment, .pack). This affects placement when the union is used as a struct member. Default NULL uses natural alignment.

Value

UnionType object

Examples

```
# Normal union
U <- ffi_union(c = ffi_char(), i = ffi_int())

# Packed union (alignment = 1)
PackedU <- ffi_union(c = ffi_char(), i = ffi_int(), .pack = 1)

# Packed union in a struct - offset of next field is affected
S <- ffi_struct(u = PackedU, after = ffi_char())

# Note: Packed unions cannot be passed by value to C functions.
# Use pointers instead.
```

---

ffi_unpack_bits *Unpack bit-fields from an integer*

Description

Extracts multiple values from a packed integer according to specified bit widths. This is the inverse operation of `ffi_pack_bits`.

Usage

```
ffi_unpack_bits(packed_value, widths)
```

Arguments

packed_value	Integer value containing packed bit-fields
widths	Integer vector of bit widths for each field

Details

Values are unpacked from LSB to MSB (least significant bit to most significant bit). Each value is extracted by shifting and masking according to its width.

Value

Integer vector of unpacked values

Examples

```
# Unpack a value with three bit-fields
values <- ffi_unpack_bits(0x65L, c(1L, 3L, 4L))
values # [1]  1  5 12

# Round-trip test
packed <- ffi_pack_bits(c(1L, 5L, 12L), c(1L, 3L, 4L))
identical(ffi_unpack_bits(packed, c(1L, 3L, 4L)), c(1L, 5L, 12L))
```

`ffi_unpack_bits64` *Unpack bit-fields from a 64-bit value*

Description

Extracts multiple values from a packed 64-bit value according to specified bit widths.

Usage

```
ffi_unpack_bits64(packed_value, widths)
```

Arguments

`packed_value`	Packed value (as double for 64-bit range)
`widths`	Integer vector of bit widths for each field

Value

Integer vector of unpacked values

Examples

```
packed <- ffi_pack_bits64(c(1L, 5L, 12L), c(1L, 3L, 4L))
ffi_unpack_bits64(packed, c(1L, 3L, 4L)) # c(1, 5, 12)
```

ffi_ushort *Ushort FFI type*

Description

Ushort FFI type

Usage

```
ffi_ushort()
```

Value

FFIType object for ushort

ffi_validate_call *Validate FFI call prerequisites*

Description

Performs comprehensive validation of FFI call inputs before making the call. This helps diagnose issues that would otherwise cause crashes.

Usage

```
ffi_validate_call(cif, symbol, args = list(), verbose = FALSE)
```

Arguments

cif	CIF object defining the call interface
symbol	NativeSymbol object for the function
args	List of arguments to pass
verbose	Logical; if TRUE, print diagnostic information

Details

This function checks:

- CIF and symbol pointers are not NULL
- Argument count matches CIF specification
- No NA values in arguments (unless explicitly allowed)
- Pointer arguments are not NULL (when applicable)

Note that even with all checks passing, crashes can still occur if:

- The C function signature doesn't match the CIF
- Pointer arguments point to invalid memory
- Buffer sizes are incorrect
- The C function itself has bugs

Value

A list with validation results:

valid Logical; TRUE if all checks pass

errors Character vector of error messages (empty if valid)

warnings Character vector of warning messages

Examples

```
## Not run:
cif <- ffi_cif(ffi_int(), ffi_int(), ffi_int())
sym <- ffi_symbol("add_ints")
result <- ffi_validate_call(cif, sym, list(1L, 2L))
if (result$valid) {
  ffi_call(cif, sym, 1L, 2L)
}

## End(Not run)
```

---

ffi_void *void FFI type*

Description

void FFI type

Usage

```
ffi_void()
```

Value

FFIType object for void

ffi_wchar_t *Wide char FFI type*

Description

Wide char FFI type

Usage

```
ffi_wchar_t()
```

Value

FFIType object for wchar_t

FFIClosure *FFI Closure - R function as C callback*

Description

A closure wraps an R function so it can be used as a callback from C code. The closure has an associated CIF that describes the function signature.

Usage

```
FFIClosure(
  r_function = function() NULL,
  cif = CIF(),
  ref = NULL,
  func_ptr = NULL
)
```

Arguments

r_function	The R function to wrap
cif	CIF object describing the callback signature
ref	External pointer to the closure
func_ptr	External pointer to the executable function

Value

An FFIClosure object

FFIType *FFI Type representation*

Description

FFI Type representation

Usage

```
FFIType(name = character(0), size = integer(0), ref = NULL)
```

Arguments

name	Character name of the type
size	Integer size in bytes
ref	External pointer to ffi_type

Value

An FFIType object

FieldInfo *Field Information Class*

Description

Represents metadata about a single field in a structure. Field Information Class

Usage

```
FieldInfo(
  name = character(0),
  type = FFIType(),
  offset = integer(0),
  size = integer(0),
  index = integer(0)
)
```

Arguments

name	Character name of the field
type	FFIType of the field
offset	Integer byte offset within structure
size	Integer size of field in bytes
index	Integer 1-based field index

Details

Contains metadata about a struct field including its name, type, byte offset within the structure, size, and index.

Value

A FieldInfo object

---

generate_enum_definition

Generate R enum definition from parsed enum

Description

Generate R enum definition from parsed enum

Usage

```
generate_enum_definition(enum_name, enum_values)
```

Arguments

enum_name	Name of the enum
enum_values	Named integer vector of enum values

Value

Character vector with R code

---

generate_function_wrapper

Generate R function wrapper from parsed function

Description

Generate R function wrapper from parsed function

Usage

```
generate_function_wrapper(func_def, typedefs = NULL)
```

Arguments

func_def	Function definition (row from functions data.frame)
typedefs	Named character vector of typedefs (optional). Used to resolve typedef'd types like SEXPTYPE to their underlying FFI types.

Value

Character vector with R code

`generate_package_from_headers`

Generate complete package from header files Generate complete package from header files

---

Description

Creates all necessary R files for a package wrapping a C library. Generates a proper R package structure with DESCRIPTION, NAMESPACE, and R code in the R/ subfolder. Uses templates from inst/templates/.

Usage

```
generate_package_from_headers(
  header_files,
  package_name,
  library_name,
  output_dir = package_name,
  library_path = NULL,
  use_system_lib = TRUE,
  include_helpers = TRUE,
  use_api_mode = FALSE,
  authors_r = NULL,
  title = NULL,
  description = NULL
)
```

Arguments

`header_files`	Character vector of header file paths
`package_name`	Name of the R package
`library_name`	Name of the shared library
`output_dir`	Directory to create the package (package root)
`library_path`	Optional: full path to shared library (for custom installs)
`use_system_lib`	Logical: search system library paths
`include_helpers`	Logical: include allocation helper functions
`use_api_mode`	Logical: use API mode (compile struct helpers into package). When TRUE, generates src/ directory with init.c and struct_helpers.c for compiled struct accessors. When FALSE (default), uses ABI mode with runtime offset calculation. API mode is required for structs with bitfields.

authors_r	Authors@R field for DESCRIPTION (R code string). Default creates a placeholder person().
title	Package title (default: auto-generated)
description	Package description (default: auto-generated)

Value

Invisibly returns list of generated files

Examples

```
## Not run:
# ABI mode (default) - runtime offset calculation
generate_package_from_headers(
  header_files = "mylib.h",
  package_name = "MyRPackage",
  library_name = "mylib",
  use_api_mode = FALSE
)

# API mode - compiled struct helpers (better for bitfields)
generate_package_from_headers(
  header_files = c("mylib.h", "mylib_utils.h"),
  package_name = "MyRPackage",
  library_name = "mylib",
  output_dir = "MyRPackage",
  use_api_mode = TRUE,
  use_system_lib = TRUE,
  include_helpers = TRUE,
  authors_r = 'person("John", "Doe", email = "john@example.com", role = c("aut", "cre"))',
  title = "FFI Bindings to mylib",
  description = "Auto-generated FFI bindings for mylib with compiled struct helpers."
)

## End(Not run)
```

---

generate_package_init *Generate .onLoad/.onUnload for package*

Description

Creates the zzz.R file content for loading external libraries

Usage

```
generate_package_init(
  library_name,
  package_name,
  library_path = NULL,
```

```
  use_system_lib = TRUE
)
```

Arguments

library_name	Name of the shared library (e.g., "mylib")
package_name	Name of the R package
library_path	Optional: specific path to library, or NULL for system search
use_system_lib	Logical: search system library paths

Value

Character string with zzz.R content

Examples

```
## Not run:
# Generate for system library
code <- generate_package_init("mylib", "MyRPackage", use_system_lib = TRUE)
writeLines(code, "R/zzz.R")

# Generate for bundled library
code <- generate_package_init("mylib", "MyRPackage", use_system_lib = FALSE)

## End(Not run)
```

---

generate_r_bindings *Generate R bindings from parsed header*

Description

Generate R bindings from parsed header

Usage

```
generate_r_bindings(parsed_header, output_file = NULL, verbose = FALSE)
```

Arguments

parsed_header	Parsed header object from ffi_parse_header()
output_file	Optional file to write code to
verbose	If TRUE, print progress messages

Value

Character vector with all generated R code

generate_struct_definition

Generate R struct definition from parsed struct

Description

Generate R struct definition from parsed struct

Usage

```
generate_struct_definition(struct_name, struct_def, typedefs = NULL)
```

Arguments

struct_name	Name of the struct
struct_def	Struct definition from parsed header
typedefs	Optional data frame of typedefs to resolve type aliases

Value

Character vector with R code

generate_struct_helpers

Generate struct helper functions (allocator, from_list, to_list)

Description

Creates convenience functions for working with a struct type:

- new_<struct>(): Allocate a new struct, optionally initialize from values
- <struct>_to_list(): Convert struct pointer to R list

Usage

```
generate_struct_helpers(struct_name, field_names)
```

Arguments

struct_name	Name of the struct (should match the ffi_struct variable name)
field_names	Character vector of field names

Value

Character string with R code for helper functions

generate_typedef_definition

Generate R typedef alias from parsed typedef

Description

Creates R code that maps a typedef alias to its underlying FFI type. Handles simple type aliases (e.g., typedef int my_int) and struct typedefs.

Usage

```
generate_typedef_definition(
  alias_name,
  base_type,
  known_structs = character(),
  known_typedefs = character()
)
```

Arguments

alias_name	Name of the typedef alias
base_type	The underlying C type string
known_structs	Character vector of known struct names (for struct type resolution)
known_typedefs	Named character vector of already-processed typedefs (for chained resolution)

Value

Character string with R code, or NULL if type cannot be mapped

generate_union_definition

Generate R union definition from parsed union

Description

Generate R union definition from parsed union

Usage

```
generate_union_definition(union_name, union_def)
```

Arguments

union_name	Name of the union
union_def	Union definition (list of fields)

Value

Character vector with R code

`get_pointer_type` *Get pointer type tag*

Description

Retrieves the type tag from an external pointer.

Usage

```
get_pointer_type(ptr)
```

Arguments

`ptr`	External pointer

Value

Character string with the type name

`is_null_pointer` *Check if pointer is NULL*

Description

Check if pointer is NULL

Usage

```
is_null_pointer(ptr)
```

Arguments

`ptr`	External pointer to check

is_protected_ptr *Check if an object is a protected SEXP pointer*

Description

Check if an object is a protected SEXP pointer

Usage

```
is_protected_ptr(ptr)
```

Arguments

ptr	An external pointer

Value

TRUE if ptr was created by sexp_ptr() or data_ptr()

libffi_version *Get libffi version string*

Description

Get libffi version string

Usage

```
libffi_version()
```

make_typed_pointer *Create typed external pointer*

Description

Creates an external pointer with a specific type tag for better type safety, similar to Rffi's approach.

Usage

```
make_typed_pointer(ptr, type_name)
```

Arguments

ptr	External pointer
type_name	Character string describing the pointer type

Value

External pointer with type tag

NativeSymbol *Native Symbol Reference*

Description

Native Symbol Reference

Usage

```
NativeSymbol(name = character(0), address = NULL, library = character(0))
```

Arguments

name	Character name of the symbol
address	External pointer to the symbol
library	Character name of library (optional)

Value

A NativeSymbol object

---

pointer_to_string *Convert pointer to string safely*

Description

Explicitly converts an external pointer to a character string. Use this instead of relying on automatic conversion heuristics.

Usage

```
pointer_to_string(ptr)
```

Arguments

ptr	External pointer that points to a null-terminated string

Value

Character vector of length 1, or NULL if pointer is NULL

`pointer_to_string_safe` *Convert pointer to string safely*

Description

Convert pointer to string safely

Usage

```
pointer_to_string_safe(ptr)
```

Arguments

`ptr`	External pointer to convert

Value

Character string or NULL if pointer is NULL

`print.rffi_compiled_lib` *Print method for compiled library*

Description

Print method for compiled library

Usage

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

ptr_to_sexp *Get the R object from a protected SEXP pointer*

Description

Retrieves the original R object from a pointer created by sexp_ptr().

Usage

```
ptr_to_sexp(ptr)
```

Arguments

ptr	A protected SEXP pointer

Value

The original R object

release_ptr *Manually release a protected pointer*

Description

Normally not needed - the finalizer handles this automatically. Use only if you need to release protection early.

Usage

```
release_ptr(ptr)
```

Arguments

ptr	A protected pointer from sexp_ptr() or data_ptr()

Value

NULL invisibly

sexp_helpers *SEXP Pointer Helpers for Safe FFI Usage*

Description

Functions for safely extracting pointers from R objects while ensuring the R object remains protected from garbage collection.

Details

When passing R objects to C functions via FFI, we need to:

1. Extract the underlying pointer (SEXP or data pointer)
2. Prevent R from garbage collecting the object while we use the pointer
3. Release the protection when we're done

These helpers use R's `R_PreserveObject`/`R_ReleaseObject` mechanism to prevent GC. The returned external pointer includes a finalizer that automatically releases the protection when the pointer is no longer used.

sexp_ptr *Get SEXP pointer from R object with GC protection*

Description

Returns an external pointer to the SEXP (R object header) while ensuring the object won't be garbage collected as long as the pointer exists.

Usage

```
sexp_ptr(x)
```

Arguments

x	Any R object

Value

External pointer to the SEXP, with finalizer to release protection

Examples

```
## Not run:
x <- c(1L, 2L, 3L) # Use c() not 1:3 to avoid ALTREP
ptr <- sexp_ptr(x)
# ptr is now safe to pass to C functions expecting SEXP
# When ptr is garbage collected, the protection is released

## End(Not run)
```

StructType *FFI Structure Type*

Description

FFI Structure Type

Usage

```
StructType(
  name = character(0),
  size = integer(0),
  ref = NULL,
  fields = character(0),
  field_types = list(),
  pack = NULL,
  has_packed_change = logical(0)
)
```

Arguments

name	Character name of the type
size	Integer size in bytes
ref	External pointer to ffi_type
fields	Character vector of field names
field_types	List of FFIType objects for each field
pack	Integer packing alignment (NULL for default/natural alignment)
has_packed_change	Logical indicating if packing changes field offsets from natural alignment. When TRUE, the struct cannot be passed by value to C functions (only pointers work) because libffi doesn't support packed structs.

tcc_available *Check if TCC is available*

Description

Check if TCC is available

Usage

```
tcc_available()
```

Value

Logical indicating if TCC is available

tcc_binary_path *Get path to embedded TCC binary*

Description

Get path to embedded TCC binary

Usage

```
tcc_binary_path()
```

Value

Path to tcc executable in installed package

tcc_extract_defines *Extract #define macros from C header file or preprocessed lines*

Description

Extract #define macros from C header file or preprocessed lines

Usage

```
tcc_extract_defines(header_file = NULL, preprocessed_lines = NULL)
```

Arguments

header_file	Path to C header file (optional if preprocessed_lines provided)
preprocessed_lines	Character vector from tcc_preprocess() (optional)

Value

Named list of macro definitions

tcc_preprocess *Preprocess C header file using embedded TCC*

Description

Preprocess C header file using embedded TCC

Usage

```
tcc_preprocess(header_file, includes = NULL, keep_defines = FALSE)
```

Arguments

header_file	Path to C header file
includes	Additional include directories
keep_defines	Keep #define directives (not supported by -E alone)

Value

Character vector of preprocessed lines

tcc_run *Compile and run C code using embedded TCC*

Description

Compile and run C code using embedded TCC

Usage

```
tcc_run(code, args = character())
```

Arguments

code	C source code as string
args	Arguments to pass to compiled program

Value

Output from program

`UnionType` *FFI Union Type*

Description

FFI Union Type

Usage

```
UnionType(
  name = character(0),
  size = integer(0),
  ref = NULL,
  fields = character(0),
  field_types = list(),
  pack = NULL,
  has_packed_change = logical(0)
)
```

Arguments

`name`	Character name of the type
`size`	Integer size in bytes
`ref`	External pointer to ffi_type
`fields`	Character vector of field names
`field_types`	List of FFIType objects for each field
`pack`	Integer packing alignment (NULL for default/natural alignment)
`has_packed_change`	Logical indicating if packing changes alignment from natural. When TRUE, the union cannot be passed by value to C functions (only pointers work) because libffi doesn't support packed unions.

Index