Package 'rdyncall'

July 15, 2026

Version 0.10.0

Title Improved Foreign Function Interface and Dynamic Bindings to C Libraries

Depends R (>= 3.0.0)

Description Provides a cross-platform framework for dynamic binding of C libraries using a flexible Foreign Function Interface (FFI).
The FFI supports almost all fundamental C types, multiple calling conventions, symbolic access to foreign C struct/union data types and wrapping of R functions as C callback function pointers.
Dynamic bindings to shared C libraries are data-driven by cross-platform binding specifications using a compact plain text format; the package includes a 'DynPort' binding specification for 'SDL3' generated from current headers with 'porter'.
The package includes a variety of technology demos and OS-specific notes for installation of shared libraries. For the underlying methods and bundled 'DynCall' libraries, see Adler (2012) <doi:10.32614/RJ-2012-004> and Adler and Philipp (2008) <https://dyncall.org>.

License MIT + file LICENSE

URL https://github.com/hongyuanjia/rdyncall, https://dyncall.org

BugReports https://github.com/hongyuanjia/rdyncall/issues

Suggests Rtinycc, tinytest

Config/Needs/website r-lib/asciicast

RoxygenNote 7.3.3

Encoding UTF-8

NeedsCompilation yes

Author Daniel Adler [aut, cph],
Hongyuan Jia [aut, cre, cph],
Tassilo Philipp [ctb, cph],
Olivier Chafik [ctb, cph]

Maintainer Hongyuan Jia <hongyuanjia@cqust.edu.cn>

Repository CRAN

Date/Publication 2026-07-15 08:10:09 UTC

Contents

`ccallback` *Dynamic wrapping of R functions as C callbacks*

Description

Function to wrap R functions as C function pointers.

Usage

```
ccallback(signature, fun, envir = new.env())

callback_status(callback)

callback_is_active(callback)

callback_last_error(callback)

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

Arguments

`signature`	character string specifying the call signature of the C function callback type.
`fun`	R function to be wrapped as a C function pointer.
`envir`	the environment in which to evaluate the call to `fun`.
`callback`	external pointer returned by `ccallback()`.
`x`	object returned by `callback_status()`.
`...`	unused.

Details

Callbacks are user-defined functions that are registered in a foreign library and that are executed at a later time from within that library. Examples include user-interface event handlers that are registered in GUI toolkits, and, comparison functions for custom data types to be passed to generic sort algorithm.

The function `ccallback()` wraps an R function `fun` as a C function pointer and returns an external pointer. The foreign C function type of the wrapped R function is specified by a call signature given by `signature`.

When the C function pointer is called, a global callback handler (implemented in C) is executed first, that dynamically creates an R call expression to `fun` using the arguments, passed from C and converted to R, according to the argument types signature within the call signature specified. See `dyncall()` for details on the format.

Finally, the handler evaluates the R call expression within the environment given by `envir`. On return, the R return value of `fun` is coerced to the C value, according to the return type signature specified in `signature` .

Aggregate by-value callback arguments and returns use the same `<Type>` signature syntax as `dyncall()`. An aggregate argument is passed to the R callback as a raw-backed `cdata` object with `struct` and `typeinfo` attributes. An aggregate return value must be a raw-backed object for the same aggregate type and size. Type or storage mismatches disable the callback and emit a warning.

Aggregate callbacks are supported on the implemented 64-bit x86 and ARM64 dyncallback backends. On unsupported backends, creating a callback whose signature contains `<Type>` fails early.

If an error occurs during the evaluation, the callback will be disabled for further invocations. Use `callback_status()` or `callback_last_error()` to inspect a callback after foreign code has invoked it.

`callback_status()` reports whether a callback is still active, how many times it has been invoked, and why it was disabled. A disabled callback is not re-enabled by these helpers; create a new callback if foreign code needs to call into R again.

`callback_is_active()` returns only the active flag. `callback_last_error()` returns `NULL` until an error disables the callback. Once an error has been recorded, it returns a list with `message`, `class`, and `reason`.

Value

An external pointer to a synthetically generated C function.

`callback_status()` returns a list with class `callback_status`. `callback_is_active()` returns a single logical value. `callback_last_error()` returns `NULL` or a list describing the last error.

Note

The call signature **MUST** match the foreign C callback function type, otherwise an activated callback call from C can lead to a **fatal R, process crash**.

A small amount of memory is allocated with each wrapper. A finalizer function that frees the allocated memory is registered at the external pointer. If the external callback function pointer is registered in a C library, a reference should also be held in R as long as the callback can be activated from a foreign C run-time context, otherwise the garbage collector might call the finalizer and the next invocation of the callback could lead to a **fatal R process crash** as well.

References

Adler, D. (2012) "Foreign Library Interface", *The R Journal*, **4(1)** , 30–40, June 2012. https://journal.r-project.org/articles/RJ-2012-004/

Adler, D., Philipp, T. (2008) *DynCall Project*. https://dyncall.org

See Also

See call signature for details on call signatures, `reg.finalizer()` for details on finalizers.

Examples

```
# Create a function, wrap it to a callback and call it via dyncall:
f <- function(x, y) x + y
cb <- ccallback("ii)i", f)
r <- dyncall(cb, "ii)i", 20, 3)

# Sort vectors directly via 'qsort' C library function using an R callback:
dynbind(c("msvcrt","c","c.so.6"), "qsort(piip)v;")
cb <- ccallback("pp)i", function(px, py) {
    x <- unpack(px, 0, "d")
    y <- unpack(py, 0, "d")
    if (x >  y) return(1) else if (x == y) return(0) else return(-1)
})
x <- rnorm(100)
qsort(x, length(x), 8, cb)
x

cb <- ccallback("i)i", function(x) {
    if (x < 0) stop("negative values are not supported")
    x
})
callback_is_active(cb)
callback_status(cb)
```

`cstruct` *Allocation and handling of foreign C aggregate data types*

Description

Functions for allocation, access and registration of foreign C `struct` and `union` data type.

Usage

```
cstruct(sigs, envir = parent.frame())

cunion(sigs, envir = parent.frame())
```

```
as.ctype(x, type)

cdata(type)

## S3 method for class 'struct'
x$index

## S3 replacement method for class 'struct'
x$index <- value

## S3 method for class 'struct'
print(x, indent = 0, ...)

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

Arguments

sigs	character string that specifies several C struct/union type signatures.
envir	the environment to install S3 type information object(s).
x	external pointer or atomic raw vector of S3 class 'struct'.
type	S3 typeinfo() Object or character string that names the structure type.
index	character string specifying the field name.
value	value to be converted according to struct/union field type given by field index.
indent	indentation level for pretty printing structures.
...	additional arguments to be passed to base::print() method.

Details

References to foreign C data objects are represented by objects of class 'struct'.

Two reference types are supported:

- *External pointers* returned by dyncall() using a call signature with a *typed pointer* return type signature and pointers extracted as a result of unpack() and S3 struct $-operators.
- *Internal objects*, memory-managed by R, are allocated by cdata(): An atomic raw storage object is returned, initialized with length equal to the byte size of the foreign C data type.

In order to access and manipulate the data fields of foreign C aggregate data objects, the $ and $<- S3 operator methods can be used.

S3 objects of class struct have an attribute struct set to the name of a typeinfo object, which provides the run-time type information of a particular foreign C type.

The run-time type information for foreign C struct and union types need to be registered once via cstruct and cunion functions. The C data types are specified by sigs, a signature character string. The formats for both types are described next:

Structure type signatures describe the layout of aggregate struct C data types. Type Signatures are used within the field-types. Fixed-size array fields are written as a type signature followed

by [N], for example C[4] for unsigned char[4] or <Point>[2] for two nested aggregate values. field-names consists of space separated identifier names and should match the number of fields. Integer bitfields are written as name:width in the field name list. Unnamed padding bitfields use :width; zero-width alignment bitfields use :0. Optional layout directives can follow the field names before the final semicolon.

```
struct-name { field-types } field-names [layout-directives] ;
```

Here is an example of a C struct type:

```
struct Rect {
  signed short x, y;
  unsigned short w, h;
};
```

The corresponding structure type signature string is:

```
"Rect{ssSS}x y w h;"
```

Bitfields keep the ordinary type signature in field-types and put the bit width next to the field name:

```
"Flags{IIII}a:1 b:3 :4 c:8;"
```

Packed or manually aligned aggregate layouts can be registered with @packed, @pack(n) and @align(n) directives, where n must be a positive power of two. @packed is equivalent to @pack(1), @pack(n) caps member alignment at n, and @align(n) raises the final aggregate alignment to at least n.

```
"Packed{Cd}c d @packed;"
"Pack4{Cd}c d @pack(4);"
"PackedAligned{Cd}c d @packed @align(8);"
```

Union type signatures describe the components of the union C data type. Type signatures are used within the field-types. Fixed-size array fields use the same [N] suffix as structure fields. field-names consists of space separated identifier names and should match the number of fields. The same layout directives can follow union field names.

```
union-name | field-types } field-names [layout-directives] ;
```

Here is an example of a C union type:

```
union Value {
  int anInt;
  float aFloat;
  struct LongValue aStruct
};
```

The corresponding union type signature string is:

```
"Value|if<LongValue>}anInt aFloat aStruct;"
```

as.ctype() can be used to *cast* a foreign C data reference to a different type. When using an external pointer reference, this can lead quickly to a **fatal R process crash** - like in C.

Value

The functions in this help topic have the following return values:

- `cstruct()` and `cunion()` return `NULL`; they are called for the side effect of registering `typeinfo` objects in `envir`.
- `cdata()` returns an atomic raw vector of S3 class `struct`. The raw bytes store an R-managed C aggregate object, and attributes record its run-time type information.
- `as.ctype()` returns `x` tagged with S3 classes `ctype` and `struct`, together with the associated type-information attributes.
- The `$.struct` method returns the named field converted from its underlying C representation. Fixed-size arrays return a vector or list, and nested aggregate fields return `struct` object(s).
- The `$<-.struct` method returns the modified `struct` object after packing the replacement value into the selected field.
- `print.struct()` and `print.ctype()` return `x` invisibly after printing a field summary.

See Also

`dyncall()` for type signatures and `typeinfo()` for details on run-time type information S3 objects.

Examples

```
# Specify the following foreign type:
# struct Rect {
#     short x, y;
#     unsigned short w, h;
# }
cstruct("Rect{ssSS}x y w h;")
r <- cdata(Rect)
print(r)
r$x <- 40
r$y <- 60
r$w <- 10
r$h <- 15
print(r)
str(r)
```

`dynbind` *Binding C library functions via thin call wrappers*

Description

Function to bind several foreign functions of a C library via installation of thin R call wrappers.

Usage

```
dynbind(
  libnames,
  signature,
  envir = parent.frame(),
  callmode = "default",
  pattern = NULL,
  replace = NULL,
  funcptr = FALSE,
  variadic = FALSE
)

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

Arguments

libnames	character vector or external pointer handle specifying the shared library. Character values that contain a path separator, or name an existing file, are passed directly to dynload(). Other character values are treated as short library names and loaded using dynfind(). External pointer handles returned by dynload() or dynfind() are used directly.
signature	character string specifying the *library signature* that determines the set of foreign function names and types. See details.
envir	the environment to use for installation of call wrappers.
callmode	character string specifying the calling convention, see details.
pattern	NULL or regular expression character string applied to symbolic names.
replace	NULL or replacement character string applied to pattern part of symbolic names.
funcptr	logical, that indicates whether foreign objects refer to functions (FALSE, default) or to function pointer variables (TRUE rarely needed).
variadic	logical, that indicates whether wrappers should call C variadic functions using dyncall_variadic(). Cannot be combined with funcptr = TRUE.
x	S3 dynbind.report object to print.
...	additional arguments to be passed to base::print() methods.

Details

dynbind() makes a set of C functions available to R through installation of thin call wrappers. The set of functions, including the symbolic name and function type, is specified by signature; a character string that encodes a library signature:

The **library signature** is a compact plain-text format to specify a set of function bindings. It consists of function names and corresponding call signature. Function bindings are separated by ";" (semicolon); white spaces (including tab and new line) are allowed before and after semicolon.

```
function-name ( call-signature ; ...
```

Here is an example that specifies three function bindings to the OpenGL library:

```
`"glAccum(If)v ; glClear(I)v ; glClearColor(ffff)v ;"`
```

Symbolic names are resolved using the library specified by `libnames`. Character short library names are loaded using `dynfind()`, character paths are loaded directly using `dynload()`, and external pointer handles returned by `dynload()` or `dynfind()` are used as-is. For each function, a thin call wrapper function is created using the following template:

```
function(...) .dyncall.<MODE> ( <TARGET>, <SIGNATURE>, ... )
```

`<MODE>` is replaced by `callmode` argument, see `dyncall()` for details on calling conventions. `<TARGET>` is replaced by the external pointer, resolved by the `function-name`. `<SIGNATURE>` is replaced by the call signature string contained in `signature`.

The call wrapper is installed in the environment given by `envir`. The assignment name is obtained from the function signature. If `pattern` and `replace` is given, a text replacement is applied to the name before assignment, useful for basic C name space mangling such as exchanging the prefix.

As a special case, `dynbind()` supports binding of pointer-to-function variables, indicated by setting `funcptr` to `TRUE`, in which case `<TARGET>` is replaced with the expression `unpack(<TARGET>,"p",0)` in order to dereference `<TARGET>` as a pointer-to-function variable at call-time.

`variadic = TRUE` creates wrappers for C functions declared with `...`. Generated wrappers accept normal call arguments through `...` and a named `.varargs` argument that describes the run-time vararg signature passed to `dyncall_variadic()`.

Value

The function returns a list with two fields:

`libhandle` External pointer returned by `dynload()`.

`unresolved.symbols`
vector of character strings, the names of unresolved symbols.

As a side effect, for each wrapper, `dynbind()` assigns the `function-name` to the corresponding call wrapper function in the environment given by `envir`.

If no shared library is found, an error is reported.

See Also

`dyncall()` for details on call signatures and calling conventions, `dynfind()` for details on short library names, `unpack()` for details on reading low-level memory (e.g. dereferencing of (function) pointer variables).

Examples

```
info <- dynbind("R", "R_ShowMessage(Z)v; R_rsort(pi)v;")
R_ShowMessage("hello")
```

dyncall *Foreign Function Interface with support for almost all C types*

Description

Functions to call pre-compiled code with support for most C argument and return types.

Usage

```
dyncall(address, signature, ..., callmode = "default")

dyncall_variadic(
  address,
  signature,
  varargs = "",
  ...,
  callmode = c("default", "cdecl")
)

dyncall.cdecl(address, signature, ...)

dyncall.default(address, signature, ...)

dyncall.stdcall(address, signature, ...)

dyncall.thiscall.gcc(address, signature, ...)

dyncall.thiscall.msvc(address, signature, ...)

dyncall.fastcall.gcc(address, signature, ...)

dyncall.fastcall.msvc(address, signature, ...)

dyncall.thiscall(address, signature, ...)

dyncall.fastcall(address, signature, ...)
```

Arguments

address	external pointer to foreign function.
signature	character string specifying the *call signature* that describes the foreign function type. See details.
...	arguments to be passed to the foreign function. Arguments are converted from R to C values according to the *call signature*. See details.
callmode	character string specifying the *calling convention*. This argument has no effect on most platforms, but on Microsoft Windows 32-Bit Intel/x86 platforms. See details.

varargs — character string specifying the type signatures for the arguments passed through the C ... portion of a variadic function. This string contains argument types only, without) and without a return type.

Details

dyncall() offers a flexible Foreign Function Interface (FFI) for the C language with support for calls to arbitrary pre-compiled C function types at run-time. Almost all C fundamental argument- and return types are supported including extended support for pointers. No limitations is given for arity as well. In addition, on the Microsoft Windows 32-Bit Intel/x86 platform, it supports multiple calling conventions to interoperate with System DLLs. Foreign C function types are specified via plain text *type signatures*. The foreign C function type of the target function is known to the FFI in advance, before preparation of the foreign call via plain text *type signature* information. This has several advantages: R arguments do not need to match exactly. Although R lacks some fundamental C value types, they are supported via coercion at this interface (e.g. C float and 64-bit integer). Arity and argument type checks help make this interface type-safe to a certain degree and encourage end-users to use interface from the interpreter prompt for rapid application development.

The foreign function to be called is specified by address, which is an external pointer that is obtained from dynsym() or getNativeSymbolInfo().

signature is a character string that specifies the formal argument-and-return types of the foreign function using a *call signature* string. It should match the function type of the foreign function given by address, otherwise this can lead to a **fatal R process crash**.

The calling convention is specified *explicitly* via function dyncall() using the callmode argument or *implicitly* by using dyncall.* functions. See details below.

The package option rdyncall.callvm.size controls the byte size of the internal dyncall CallVM argument stack. The default is 4096. Set this option before loading **rdyncall**; changing it after package load does not resize already-created CallVM objects.

Arguments passed via ... are converted to C according to signature; see below for details.

dyncall_variadic() calls C functions declared with .... The signature argument describes the fixed parameter prefix and return type, while varargs describes the actual argument types passed through ... at this specific call site. C default promotions are the caller's responsibility; for example, pass promoted variadic float values as double ("d").

Given that the signature matches the foreign function type, the FFI provides a certain level of type-safety to users, when exposing foreign functions via call wrappers such as done in dynbind() and dynport(). Several basic argument type-safety checks are done during preparation of the foreign function call: The arity of formals and actual arguments must match and they must be compatible as well. Otherwise, the foreign function call is aborted with an error before risking a fatal system crash.

Value

Functions return the received C return value converted to an R value. See section "Call Signature" below for details.

Type Signature

Type signatures are used by almost all other signature formats (call, library, structure and union signature) and also by the low-level (un)-packing functions.

The following table gives a list of valid type signatures for all supported C types.

Type Signature	C type	Valid R argument types	R returntype
'B'	bool	raw,logical,integer,double	logical
'c'	char	raw,logical,integer,double	integer
'C'	unsigned char	raw,logical,integer,double	integer
's'	short	raw,logical,integer,double	integer
'S'	unsigned short	raw,logical,integer,double	integer
'i'	int	raw,logical,integer,double	integer
'I'	unsigned int	raw,logical,integer,double	double
'j'	long	raw,logical,integer,double	double
'J'	unsigned long	raw,logical,integer,double	double
'l'	long long	raw,logical,integer,double	double
'L'	unsigned long long	raw,logical,integer,double	double
'f'	float	raw,logical,integer,double	double
'd'	double	raw,logical,integer,double	double
'p'	C pointer	any vector,externalptr,NULL	externalptr
'Z'	char*	character,NULL	character or NULL
'x'	SEXP	any	any
'v'	void	invalid	NULL
'*' ...	C type* (pointer)	compatible vector,externalptr,NULL	externalptr
'*<' *typename* '>'	typename* (pointer)	raw,externalptr	externalptr
'<' *typename* '>'	typename (by value)	raw `struct` from `cdata()`	raw `struct`

Aggregate by-value signatures support `struct` and `union` type information registered with `cstruct()` or `cunion()`. Aggregate arguments and returns are passed through dyncall aggregate descriptors, including nested aggregate and fixed-size array fields that are already represented in the registered typeinfo.

The last typed pointer rows of the table above refer to *typed pointer* signatures. If they appear as a return type signature, the external pointer returned is a S3 `struct` object. See `cdata()` for details. Low-level typed pointer signatures accept `NULL` and external pointers for all base pointer types. Vector-backed convenience arguments are available when the R vector storage matches the C pointee type, for example integer for `*i`/`*I`, numeric for `*d`, `floatraw` for `*f`, character/raw for `*c`/`*C`, and general vectors for `*v`.

Call Signature

Call Signatures are used by `dyncall()` and `ccallback()` to describe foreign C function types. The general form of a call signature is as following:

```
(argument-type)* ) return-type
```

The calling sequence given by the **argument types signature** is specified in direct *left-to-right* order of the formal argument types defined in C. The type signatures are put in sequence without

any white space in between. A closing bracket character ')' marks the end of argument types, followed by a single **return type signature**.

Derived pointer types can be specified as untyped pointers via `'p'` or via prefix `'*'` following the underlying base type (e.g. `'*d'` for `double *`) which is more type-safe. For example, this can prevent users from passing a `numeric` R atomic as `int*` if using `'*i'` instead of `'p'`.

Derived pointer types to aggregate `union` or `struct` types are supported in combination with the framework for handling foreign data types. See `cdata()` for details. Once a C type is registered, the signature **<typename>* can be used to refer to a pointer to an aggregate C object *type**, and *<typename>* can be used to pass a raw `cdata()` aggregate object by value. If typed pointers to aggregate objects are used as a return type and the corresponding type information exists, the returned value can be printed and accessed symbolically.

Here are some examples of C function prototypes and corresponding call signatures:

	C Function Prototype	**Call Signature**
`double`	`sqrt(double);`	`"d)d"`
`double`	`dnorm(double,double,double,int);`	`"dddi)d"`
`void`	`R_isort(int*,int);`	`"pi)v"` or `"*ii)v"`
`void`	`revsort(double*,int*,int);`	`"ppi)v"` or `"*d*ii)v"`
`int`	`SDL_PollEvents(SDL_Event *);`	`"p)i"` or `"*<SDL_Event>)i"`
`SDL_Surface*`	`SDL_SetVideoMode(int,int,int,int);`	`"iiii)p"` or `"iiii)*<SDL_Surface>"`

Calling Convention

Calling Conventions specify *how* sub-routine calls are performed, and, *how* arguments and results are passed, on machine-level. They differ significantly among families of CPU Architectures as well as OS and Compiler implementations.

On most platforms, a single `"default"` C Calling Convention is used. As an exception, on the Microsoft Windows 32-Bit Intel/x86 platform several calling conventions are common. Most of the C libraries still use a `"default"` C (also known as `"cdecl"`) calling convention, but when working with Microsoft System APIs and DLLs, the `"stdcall"` calling convention must be used.

It follows a description of supported Win32 Calling Conventions:

`"cdecl"` Dummy alias to *default*

`"stdcall"` C functions with *stdcall* calling convention. Useful for all Microsoft Windows System Libraries (e.g. KERNEL32.DLL, USER32.DLL, OPENGL32.DLL ...). Third-party libraries usually prefer the default C *cdecl* calling convention.

`"fastcall.msvc"` C functions with *fastcall* calling convention compiled with Microsoft Visual C++ Compiler. Very rare usage.

`"fastcall.gcc"` C functions with *fastcall* calling convention compiled with GNU C Compiler. Very rare usage.

`"thiscall"` C++ member functions.

`"thiscall.gcc"` C++ member functions compiled with GNU C Compiler.

`"thiscall.msvc"` C++ member functions compiled with Microsoft Visual C++ Compiler.

As of the current version of this package and for practical reasons, the `callmode` argument does not have an effect on almost all platforms, except that if R is running on Microsoft Windows 32-Bit Intel/x86 platform, `dyncall` uses the specified calling convention. For example, when loading OpenGL across platforms, `"stdcall"` should be used instead of `"default"`, because on Windows, OpenGL is a System DLL. This is very exceptional, as in most other cases, `"default"` (or `"cdecl"`, the alias) need to be used for normal C shared libraries on Windows.

At this stage of development, support for C++ calls should be considered experimental. Support for Fortran is planed but not yet implemented in dyncall.

Portability

The implementation is based on the *dyncall* library (part of the DynCall project).

The following processor architectures are supported: X86 32- and 64-bit, ARM v4t-v7 oabi/eabi (aapcs) and armhf including support for Thumb ISA, PowerPC 32-bit, MIPS 32- and 64-Bit, SPARC 32- and 64-bit. The library has been built and tested to work on various OSs: Linux, Mac OS X, Windows 32/64-bit, BSDs, Haiku, Nexenta/Open Solaris, Solaris, Minix and Plan9, as well as embedded platforms such as Linux/ARM (OpenMoko, Beagleboard, Gumstix, Efika MX, Raspberry Pi), Nintendo DS (ARM), Sony Playstation Portable (MIPS 32-bit/eabi) and iOS (ARM - armv6 mode ok, armv7 unstable). In the context of R, dyncall has currently no support for PowerPC 64-Bit.

Note

The target address, calling convention and call signature **MUST** match foreign function type, otherwise the invocation could lead to a **fatal R process crash**.

References

Adler, D. (2012) "Foreign Library Interface", *The R Journal*, **4(1)** , 30–40, June 2012. https://journal.r-project.org/articles/RJ-2012-004/

Adler, D., Philipp, T. (2008) *DynCall Project*. https://dyncall.org

Examples

```
libm <- dynfind(c("msvcrt", "m", "m.so.6"))
c_sqrt <- dynsym(libm, "sqrt")
dyncall(c_sqrt, "d)d", 144)
```

`dynfind` *Portable searching and loading of shared libraries*

Description

Function to load shared libraries using a platform-portable interface.

Usage

```
dynfind(libnames, auto.unload = TRUE)

dynfind_explain(libnames, try.load = TRUE)
```

Arguments

libnames	vector of character strings specifying several short library names.
auto.unload	logical: if TRUE then a finalizer is registered that closes the library on garbage collection. See dynload() for details.
try.load	logical: if TRUE, attempt candidates in dynfind() order until one loads. The loaded handle is immediately closed with dynunload().

Details

dynfind() offers a platform-portable naming interface for loading a specific shared library.

dynfind_explain() returns the candidate paths that dynfind() would try, optionally attempts them in the same order, and records the first loadable candidate. It is intended for diagnosing platform-specific library discovery failures without keeping a library handle open.

The naming scheme and standard locations of shared libraries are OS-specific. When loading a shared library dynamically at run-time across platforms via standard interfaces such as dynload() or dyn.load(), a platform-test is usually needed to specify the OS-dependant library file path.

This *library name problem* is encountered via breaking up the library file path into several abstract components:

```
<location> <prefix> <libname> <suffix>
```

By permutation of values in each component and concatenation, a list of possible file paths can be derived. dynfind() goes through this list to try opening a library. On the first success, the search is stopped and the function returns.

Given that the three components location, prefix and suffix are set up properly on a per OS basis, the unique identification of a library is given by libname - the short library name.

For some libraries, multiple 'short library name' are needed to make this mechanism work across all major platforms. For example, to load the Standard C Library across major R platforms:

```
lib <- dynfind(c("msvcrt", "c", "c.so.6"))
```

On Windows MSVCRT.dll would be loaded; libc.dylib on Mac OS X; libc.so.6 on Linux and libc.so on BSD.

Here is a sample list of values for the three other components:

location: /usr/local/lib/, C:/Windows/System32/

prefix: lib (common), empty - common on Windows

suffix: .dll (Windows), .so (ELF), .dylib (macOS) and empty - useful for all platforms

The vector of locations is initialized by the dynamic linker search rules, including environment variables such as PATH on Windows and LD_LIBRARY_PATH on Unix-flavour systems. If the dynamic linker lookup fails, dynfind() also checks library directories that belong to the current R runtime, such as R.home("lib") and R.home("bin"), and common package-manager library locations such as Homebrew (HOMEBREW_PREFIX, /opt/homebrew, /usr/local), MacPorts (MACPORTS_PREFIX, /opt/local), Linuxbrew (/home/linuxbrew/.linuxbrew), Scoop (SCOOP, SCOOP_GLOBAL, ProgramData/scoop), MSYS2 (MINGW_PREFIX, MSYSTEM_PREFIX, C:/msys64), vcpkg (VCPKG_ROOT) and conda (CONDA_PREFIX). On Windows, when dynfind() tries a full DLL path from one of these directories, it temporarily prepends that directory to PATH for the load attempt so that sibling transitive DLL dependencies can be resolved. (The set of hardcoded locations might expand and change within the next minor releases).

The file extension depends on the OS: .dll (Windows), .dylib (macOS), .so (all others).

On Mac OS X, the search for a library includes the 'Frameworks' folders as well. This happens before the normal library search procedure and uses a slightly different naming pattern in a separate search phase:

```
<frameworksLocation> Frameworks/ <libname> .framework/ <libname>
```

The frameworksLocation is a vector of locations such as /System/Library/ and /Library/.

dynfind() loads a library via dynload() passing over the parameter auto.unload.

Value

dynfind() returns an external pointer (library handle), if search was successful. Otherwise, if no library is located, a NULL is returned.

dynfind_explain() returns a data frame with columns libname, source, candidate, exists, loaded, and resolved_path. loaded is NA for candidates that were not attempted because an earlier candidate loaded, or because try.load = FALSE.

See Also

See dynload() for details on the loader interface to the OS-specific dynamic linker.

Examples

```
diag <- dynfind_explain(c("msvcrt", "m", "m.so.6"), try.load = FALSE)
head(diag)
```

---

dynload *Loading of shared libraries and resolving of symbols (Alternative Framework)*

Description

Alternative framework for loading of shared libraries and resolving of symbols. The framework offers *automatic unload management* of shared libraries and provides a direct interface to the dynamic linker of the OS.

Usage

```
dynload(libname, auto.unload = TRUE)

dynunload(libhandle)

dynsym(libhandle, symname, protect.lib = TRUE)

dynpath(libhandle)

dyncount(libhandle)

dynlist(libhandle)
```

Arguments

`libname`	character string giving the pathname to a shared library in OS-specific notation.
`auto.unload`	logical, if `TRUE` a finalizer will be registered that will automatically unload the library.
`libhandle`	external pointer representing a handle to an opened library.
`symname`	character string specifying a symbolic name to be resolved.
`protect.lib`	logical, if `TRUE` resolved external pointers protect library handles from finalization.

Details

`dynload()` loads a shared library into the current R process using the OS-specific dynamic linker interface. The `libname` is passed *as-is* directly to the dynamic linker and thus is given in OS-specific notation - see below for details. On success, a handle to the library represented as an external pointer R objects is returned, otherwise `NULL`. If `auto.unload` is `TRUE`, a finalizer function is registered that will unload the library on garbage collection via `dynunload()`.

`dynsym()` looks up symbol names in loaded libraries and resolves them to memory addresses returned as external pointer R objects. Otherwise `NULL` is returned. If `protect.lib` is `TRUE`, the library handle is *protected* by resolved address external pointers from unloading.

`dynpath()` returns the full path of the loaded library specified by `libhandle`.

`dyncount()` returns the number of symbols in the loaded library specified by `libhandle`.

`dynlist()` returns all symbol names in the loaded library specified by `libhandle`.

`dynunload()` explicitly unreferences the loaded library specified by `libhandle`.

Setting both `auto.unload` and `protect.lib` to `TRUE`, libraries remain loaded as long as resolved symbols are in use, and they get automatic unloaded when no resolved symbols remain.

Dynamic linkers usually hold an internal link count, such that a library can be opened multiple times via `dynload()` - with a balanced number of calls to `dynunload()` that decreases the link count to unload the library again.

Similar functionality is available via `base::dyn.load()` and `base::getNativeSymbolInfo()`, except that path names are filtered and no automatic unloading of libraries is supported.

Value

`dynload` returns an external pointer `libhandle` on success. Otherwise `NULL` is returned, if the library is not found or the linkage failed.

`dynsym` returns an external pointer `address` on success. Otherwise `NULL` is returned, if the address was invalid or the symbol has not been found.

`dynunload` always returns `NULL`.

`dynpath` returns a single string.

`dyncount` returns a single integer.

`dynlist` returns a character vector.

Shared library

Shared libraries are single files that contain compiled code, data and meta-information. The code and data can be loaded and mapped to a process at run-time once. Operating system platforms have slightly different schemes for naming, searching and linking options.

Platform	Binary format	File Extension
Linux, BSD derivates and Sun Solaris	ELF format	`so`
Darwin / Apple macOS	Mach-O format	`dylib`
Microsoft Windows	PE format	`dll`

Library search on Posix platforms (Linux,BSD,Sun Solaris)

The following text is taken from the Linux `dlopen` manual page:

These search rules will only be applied to path names that do not contain an embedded '/'.

- If the `LD_LIBRARY_PATH` environment variable is defined to contain a colon-separated list of directories, then these are searched.
- The cache file `/etc/ld.so.cache` is checked to see whether it contains an entry for filename.
- The directories `/lib` and `/usr/lib` are searched (in that order).

If the library has dependencies on other shared libraries, then these are also automatically loaded by the dynamic linker using the same rules.

Library search on Darwin (Mac OS X) platforms

The following text is taken from the Mac OS X `dlopen` manual page:

`dlopen()` searches for a compatible Mach-O file in the directories specified by a set of environment variables and the process's current working directory. When set, the environment variables must contain a colon-separated list of directory paths, which can be absolute or relative to the current working directory. The environment variables are `$LD_LIBRARY_PATH`, `$DYLD_LIBRARY_PATH`, and `$DYLD_FALLBACK_LIBRARY_PATH`. The first two variables have no default value. The default value of `$DYLD_FALLBACK_LIBRARY_PATH` is `$HOME/lib;/usr/local/lib;/usr/lib`. `dlopen()` searches the directories specified in the environment variables in the order they are listed.

When path doesn't contain a slash character (i.e. it is just a leaf name), dlopen() searches the following until it finds a compatible Mach-O file: $LD_LIBRARY_PATH, $DYLD_LIBRARY_PATH, current working directory, $DYLD_FALLBACK_LIBRARY_PATH.

When path contains a slash (i.e. a full path or a partial path) dlopen() searches the following the following until it finds a compatible Mach-O file: $DYLD_LIBRARY_PATH (with leaf name from path), current working directory (for partial paths), $DYLD_FALLBACK_LIBRARY_PATH (with leaf name from path).

Library search on Microsoft Windows platforms

The following text is taken from the Window SDK Documentation:

If no file name extension is specified ..., the default library extension .dll is appended. However, the file name string can include a trailing point character (.) to indicate that the shared library module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file in the following sequence:

- The directory from which the application loaded.
- The current directory.
- The system directory. Use the GetSystemDirectory Win32 API function to get the path of this directory.
- The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. Windows Me/98/95: This directory does not exist.
- The Windows directory. Use the GetWindowsDirectory Win32 API function to get the path of this directory.
- The directories that are listed in the PATH environment variable.

Windows Server 2003, Windows XP SP1: The default value of HKLM\System\CurrentControlSet\Control\Session Mana
is 1 (current directory is searched after the system and Windows directories).

Windows XP: If HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode is 1, the current directory is searched after the system and Windows directories, but before the directories in the PATH environment variable. The default value is 0 (current directory is searched before the system and Windows directories).

The first directory searched is the one directory containing the image file used to create the calling process. Doing this allows private dynamic-link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable.

The search path can be altered using the SetDllDirectory() function. This solution is recommended instead of using SetCurrentDirectory() or hard-coding the full path to the DLL.

If a path is specified and there is a redirection file for the application, the function searches for the module in the application's directory. If the module exists in the application's directory, the LoadLibrary() function ignores the specified path and loads the module from the application's directory. If the module does not exist in the application's directory, LoadLibrary() loads the module from the specified directory. For more information, see Dynamic Link Library Redirection from the Windows SDK Documentation.

Portability

The implementation is based on the *dynload* library (part of the DynCall project) which has been ported to all major R platforms (ELF (Linux, BSD, Solaris), Mach-O (Mac OS X) and Portable Executable (Win32/64)).

Note

On macOS, `dynlist()` enumerates symbols from system libraries in the dyld shared cache on a best-effort basis using Mach-O export information. For `/usr/lib/lib*.dylib` system aliases, matching `/usr/lib/system/libsystem_*` re-exports are included without expanding the entire libSystem umbrella.

See Also

This facility is used by `dynfind()` and `dynbind()`. Similar functionality is available from `base::dyn.load()` and `base::getNativeSymbolInfo()`.

`dynport` *Dynamic R Bindings to standard and common C libraries*

Description

Functions to turn DCF DynPort files into generated R packages that provide wrappers for C functions, object-like macros, enums and data types.

Usage

```
dynport(
  portname,
  portfile = NULL,
  repo = system.file("dynports", package = "rdyncall"),
  package = NULL,
  lib = dynport_lib(),
  rebuild = FALSE,
  load = TRUE,
  quiet = FALSE
)

dynport_install_package(
  portname,
  portfile = NULL,
  repo = system.file("dynports", package = "rdyncall"),
  package = NULL,
  lib = dynport_lib(),
  rebuild = FALSE,
  load = FALSE,
  quiet = FALSE
```

```
)

dynport_load_into(portfile, envir)

dynport_lib(create = TRUE, add = FALSE)

dynport_clear_lib(lib = dynport_lib(create = FALSE), unload = TRUE)
```

Arguments

portname	the name of a dynport, given as a literal or character string.
portfile	NULL or character string giving a DCF .dynport file to parse.
repo	character string giving the path to the root of the dynport repository.
package	NULL or character string giving the generated R package name. When NULL, the Package field from the DynPort file is prefixed by option rdyncall.dynport.package.prefix.
lib	character string giving the R library path where the generated package is installed. Defaults to dynport_lib().
rebuild	logical. If TRUE, reinstall an existing generated package when the DynPort file contents have changed.
load	logical. If TRUE, load and attach the generated package in the current R session after installation.
quiet	logical. If TRUE, suppress installation and loading output where possible.
envir	environment to populate from a DynPort file.
create	logical. If TRUE, create the default DynPort package library when it does not exist.
add	logical. If TRUE, prepend the DynPort package library to .libPaths().
unload	logical. If TRUE, unload generated DynPort packages from the current session before removing them from lib.

Details

dynport() offers a convenient method for binding entire C libraries to R. This mechanism runs cross-platform and uses dynamic linkage but it implies that the run-time library of a chosen binding need to be pre-installed in the system. Depending on the OS, the run-time libraries may be pre-installed or require manual installation. See rdyncall-demos for OS-specific installation notes for several C libraries.

The binding method is data-driven using platform-portable specifications named *DynPort* files. The current implementation supports DCF (Debian Control File) .dynport files.

When dynport() processes a *DynPort* file, it generates and installs a real R package whose namespace is populated at load time from the DynPort metadata. By default, generated package names use the prefix given by option rdyncall.dynport.package.prefix, which defaults to "dyn.". For example, a DynPort with Package: SDL3 is installed as dyn.SDL3 unless a package name is explicitly supplied.

The package ships the following current-format DCF *DynPort*:

DynPort name/C library	Description
SDL3	Simple DirectMedia Layer 3

The DCF format records the following binding metadata:

- Functions (and pointer-to-function variables) are mapped via `dynbind()` and a description of the C library using a *library signatures*.
- Symbolic names are assigned to its values for object-like macro defines and C enum types.
- Run-time type-information objects for aggregate C data types (struct and union) are registered via `cstruct()` and `cunion()`.

The file path to the *DynPort* file is derived from `portname` per default. This would refer to `"<repo>/<portname>.dynport"` where `repo` defaults to the package's `"dynports/"` sub-folder. If `portfile` is given, then this value is taken as file path.

The bundled SDL3 DynPort is generated from SDL3 headers with porter. For other libraries, generate a DCF `.dynport` file externally and pass it with `portfile`.

Value

`dynport()` invisibly returns the generated package name. `dynport_install_package()` invisibly returns the installed package path with the generated package name stored in attribute `"package"`. `dynport_load_into()` invisibly returns `envir`. `dynport_lib()` returns the DynPort package library path. `dynport_clear_lib()` invisibly returns the paths it removed.

Author(s)

Daniel Adler dadler@uni-goettingen.de

References

Adler, D. (2012) "Foreign Library Interface", *The R Journal*, **4(1)**, 30–40, June 2012. https://journal.r-project.org/articles/RJ-2012-004/

Adler, D., Philipp, T. (2008) *DynCall Project*. https://dyncall.org

Latinga, S. (1998). The Simple DirectMedia Layer Library. https://www.libsdl.org/

Examples

```
if (run_external) {
    portfile <- system.file("dynports", "SDL3.dynport",
        package = "rdyncall", mustWork = TRUE
    )
    lib <- tempfile("rdyncall-dynport-lib")
    generated <- dynport(
        portfile = portfile,
        package = "dyn.SDL3Example",
        lib = lib,
        rebuild = TRUE,
```

```
        quiet = TRUE
    )
    getExportedValue(generated, "SDL_GetPlatform")()
}
```

`is.nullptr` *Utility functions for working with foreign C data types*

Description

Functions for low-level operations on C pointers as well as helper functions and objects to handle C `float` arrays and strings.

Usage

```
is.nullptr(x)

as.externalptr(x)

offset_ptr(x, offset)

is.externalptr(x)

as.floatraw(x)

floatraw2numeric(x)

floatraw(n)

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

ptr2str(x)

strarrayptr(x)

strptr(x)
```

Arguments

`x`	object to test, convert, or pass to a pointer/string helper.
`offset`	integer specifying *byte offset* starting at 0.
`n`	integer specifying the number of single-precision C `float` values to allocate.
`...`	additional arguments to be passed to `base::print()` methods.

Details

`is.nullptr()` tests if the external pointer given by `x` represents a C `NULL` pointer.

`as.externalptr()` returns an external pointer to the data area of atomic vector given by `x`. The external pointer holds an additional reference to the `x` R object to prevent it from garbage collection. `x` must have length greater than zero.

`is.externalptr()` tests if the object given by `x` is an external pointer.

`floatraw()` creates an array with a capacity to store `n` single-precision C `float` values. The array is implemented via a `base::raw()` vector.

`as.floatraw()` coerces a numeric vector into a single-precision C `float` vector. Values given by `x` are converted to C `float` values and stored in the R raw vector via `pack()`. This function is useful when calling foreign functions that expect a C `float` pointer via `dyncall()`.

`floatraw2numeric()` coerces a C `float` (raw) vector to a numeric vector.

`ptr2str()`, `strarrayptr()`, `strptr()` are currently experimental.

`offset_ptr()` creates a new external pointer pointing to `x` plus the non-negative byte `offset`. If `x` is given as an external pointer, the address is increased by the `offset`, or, if `x` is given as a atomic vector, the address of the data (pointing to offset zero) is taken as basis and increased by the `offset`. Atomic vector offsets are checked against the vector byte size. The returned external pointer is protected (as offered by the C function `R_MakeExternalPtr`) by the external pointer `x`.

Value

A logical value is returned by `is.nullptr()` and `is.externalptr()`. `as.externalptr()` and `offset_ptr()` returns an external pointer value. `floatraw()` and `as.floatraw()` return an atomic vector of type `raw` tagged with class `floatraw`. `floatraw2numeric` returns a `numeric` atomic vector.

Examples

```
is.nullptr(NULL)

one <- as.externalptr(1)
is.externalptr(one)

floatraw(1)

floats <- as.floatraw(1:10)
all.equal(floatraw2numeric(floats), 1:10)
```

---

pack *Handling of foreign C fundamental data types*

Description

Functions to unpack/pack (read/write) foreign C data types from/to R atomic vectors and C data objects such as arrays and pointers to structures.

Usage

```
pack(x, offset, sigchar, value)

unpack(x, offset, sigchar)
```

Arguments

x	atomic vector (logical, raw, integer or double) or external pointer.
offset	integer specifying *byte offset* starting at 0.
sigchar	character string specifying the C data type by a type signature.
value	R object value to be coerced and packed to a foreign C data type.

Details

The function pack() converts an R value into a C data type specified by the signature sigchar and it writes the raw C foreign data value at byte position offset into the object x.

The function unpack() extracts a C data type according to the signature sigchar at byte position offset from the object x and converts the C value to an R value and returns it.

Byte offset calculations start at 0 relative to the first byte in an atomic vectors data area. Offsets must be non-missing, non-negative integer scalars.

If x is an atomic vector, a bound check is carried out before read/write access. Otherwise, if x is an external pointer, there is only a C NULL pointer check. Values read from R vectors must have length greater than zero.

Value

unpack() returns a read C data type coerced to an R value.

See Also

dyncall() for details on type signatures.

Examples

```
# transfer double to array of floats and back, compare precision:
n <- 6
input <- rnorm(n)
buf <- raw(n*4)
for (i in 1:n) {
    pack(buf, 4 * (i - 1), "f", input[i])
}

output <- numeric(n)
for (i in 1:n) {
    output[i] <- unpack(buf, 4 * (i - 1), "f")
}
# difference between double and float
difference <- output - input
print(cbind(input, output, difference))
```

rdyncall-demos *rdyncall demos: shared library notes*

Description

The demos of the **rdyncall** package (see `demo(package = "rdyncall")`) exercise direct calls, callbacks, generated DynPort packages and a few real shared libraries.

Some demos only use the R runtime or the platform C runtime. Others require external shared libraries to be installed and discoverable by `dynfind`.

Current demos

Demo	Required external library	Notes
`sqrt`	C math library	Usually provided by the operating system
`factorial`	none	Calls an R callback through a C function pointer
`callbacks`	none	Callback examples
`qsort`	C runtime	Uses the platform `qsort()`
`stdio`	C runtime	Uses platform standard I/O functions
`R_ShowMessage`	R shared library	Uses the loaded R runtime
`glpk`	GLPK	Solves a small linear program
`libxml2`	libxml2	Parses a small XML document
`SDL`	SDL3	Opens an SDL3 window for the snake demo
`SDL_audio`	SDL3	Opens an SDL3 audio/window demo
`SDL_raster`	SDL3 and OpenGL	Opens an SDL3/OpenGL raster demo
`raylib`	raylib	Opens a raylib window; can use `RAYLIB_LIB`

Library discovery

Place shared libraries in a standard system location or update the platform library search path so that `dynfind` can locate them. On Windows this usually means `PATH`; on Unix-like systems it can mean the dynamic linker search path or a package-manager library directory.

Several demos also honor explicit environment variables:

- `SDL3_LIB`: path to an SDL3 shared library.
- `OPENGL_LIB`: path to an OpenGL shared library.
- `RAYLIB_LIB`: path to a raylib shared library.
- `RDYNCALL_RAYLIB_CACHE`: cache directory used by the raylib demo when it downloads a release asset.

Installation hints

Install the development/runtime package names used by your operating system or package manager. Common examples are:

Platform	Examples
macOS with Homebrew	`brew install glpk libxml2 sdl3 raylib`
Debian/Ubuntu	`apt install libglpk40 libxml2 libsdl3-0 libgl1 libglu1-mesa`
Fedora	`dnf install glpk libxml2 SDL3 mesa-libGL mesa-libGLU`
Windows	Install matching DLLs with MSYS2, Scoop, vcpkg or another package manager and put them on PA

The raylib demo can use a locally installed raylib through `RAYLIB_LIB`. If `RAYLIB_LIB` is not set, it attempts to obtain a matching release asset from the raylib GitHub releases and cache it locally.

See Also

`dynfind`, `dynbind`, `dynport`

---

typeinfo *S3 class for run-time type information of foreign C data types*

Description

S3 class for run-time type information of foreign C data types.

Usage

```
typeinfo(
  name,
  type = c("base", "pointer", "struct", "union"),
  size = NA,
  align = NA,
  basetype = NA,
  fields = NA,
  signature = NA
)

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

get_typeinfo(name, envir = parent.frame())
```

Arguments

name	character string specifying the type name.
type	character string specifying the type.
size	integer, size of type in bytes.
align	integer, alignment of type in bytes.
basetype	character string, base type of 'pointer' types.
fields	data frame with name, type, offset and optional array or bitfield layout information that specifies aggregate struct and union types.
signature	character string specifying the struct/union type signature.
x	S3 typeinfo object to print.
...	additional arguments to be passed to base::print() methods.
envir	the environment to look for type object.

Details

Type information objects are created at run-time to describe the concrete layout of foreign C data types on the host machine. While type signatures give an abstract information on e.g. the field types and names of aggregate structure types, these objects store concrete memory size, alignment and layout information about C data types.

Value

List object tagged as S3 class typeinfo with the following named entries

type	Type name.
size	Size in bytes.
align	Alignment in bytes.
fields	Data frame for field information with the following columns:

name	field name
type	type name
offset	byte offset (starts counted from 0)
array_len	fixed array length, or 1 for scalar fields
bit_offset	bitfield offset, or NA for ordinary fields
bit_width	bitfield width, or NA for ordinary fields
storage_offset	bitfield storage-unit byte offset
storage_size	bitfield storage-unit size in bytes

See Also

cstruct() for details on the framework for handling foreign C data types.

Index